



PKCS #11 Specification Version 3.1

OASIS Standard

23 July 2023

This stage:

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

Previous stage:

<https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/cs01/pkcs11-spec-v3.1-cs01.pdf> (Authoritative)
<https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/cs01/pkcs11-spec-v3.1-cs01.html>
<https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/cs01/pkcs11-spec-v3.1-cs01.docx>

Latest stage:

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

Technical Committee:

OASIS PKCS 11 TC

Chairs:

Robert Relyea (rrelyea@redhat.com), Red Hat
Greg Scott (greg.scott@cryptsoft.com), Cryptsoft Pty Ltd

Editors:

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

Additional artifacts:

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

- pkcs11.h: <https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/os/include/pkcs11-v3.1/pkcs11.h>
- pkcs11f.h: <https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/os/include/pkcs11-v3.1/pkcs11f.h>
- pkcs11t.h: <https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/os/include/pkcs11-v3.1/pkcs11t.h>

Related work:

This specification replaces or supersedes:

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

This specification is related to:

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

Abstract:

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

Status:

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

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

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

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

Key words:

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

Citation format:

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

[PKCS11-Spec-v3.1]

PKCS #11 Specification Version 3.1. Edited by Dieter Bong and Tony Cox. 23 July 2023. OASIS Standard. <https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/os/pkcs11-spec-v3.1-os.html>. Latest stage: <https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/pkcs11-spec-v3.1.html>.

Notices:

Copyright © OASIS Open 2023. 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 Appendix D below.

Table of Contents

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

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

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

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

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

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

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

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

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

| | |
|--|-----|
| 6.28.5 SHA3-224 key derivation..... | 286 |
| 6.28.6 SHA3-224 HMAC key generation | 286 |
| 6.29 SHA3-256 | 287 |
| 6.29.1 Definitions | 287 |
| 6.29.2 SHA3-256 digest | 287 |
| 6.29.3 General-length SHA3-256-HMAC | 287 |
| 6.29.4 SHA3-256-HMAC | 288 |
| 6.29.5 SHA3-256 key derivation..... | 288 |
| 6.29.6 SHA3-256 HMAC key generation | 288 |
| 6.30 SHA3-384 | 288 |
| 6.30.1 Definitions | 289 |
| 6.30.2 SHA3-384 digest | 289 |
| 6.30.3 General-length SHA3-384-HMAC | 289 |
| 6.30.4 SHA3-384-HMAC | 290 |
| 6.30.5 SHA3-384 key derivation..... | 290 |
| 6.30.6 SHA3-384 HMAC key generation | 290 |
| 6.31 SHA3-512 | 290 |
| 6.31.1 Definitions | 290 |
| 6.31.2 SHA3-512 digest | 291 |
| 6.31.3 General-length SHA3-512-HMAC | 291 |
| 6.31.4 SHA3-512-HMAC | 291 |
| 6.31.5 SHA3-512 key derivation..... | 291 |
| 6.31.6 SHA3-512 HMAC key generation | 291 |
| 6.32 SHAKE..... | 292 |
| 6.32.1 Definitions | 292 |
| 6.32.2 SHAKE Key Derivation..... | 292 |
| 6.33 BLAKE2B-160..... | 293 |
| 6.33.1 Definitions | 293 |
| 6.33.2 BLAKE2B-160 digest..... | 293 |
| 6.33.3 General-length BLAKE2B-160-HMAC | 293 |
| 6.33.4 BLAKE2B-160-HMAC | 294 |
| 6.33.5 BLAKE2B-160 key derivation..... | 294 |
| 6.33.6 BLAKE2B-160 HMAC key generation | 294 |
| 6.34 BLAKE2B-256..... | 294 |
| 6.34.1 Definitions | 295 |
| 6.34.2 BLAKE2B-256 digest..... | 295 |
| 6.34.3 General-length BLAKE2B-256-HMAC | 295 |
| 6.34.4 BLAKE2B-256-HMAC | 296 |
| 6.34.5 BLAKE2B-256 key derivation..... | 296 |
| 6.34.6 BLAKE2B-256 HMAC key generation | 296 |
| 6.35 BLAKE2B-384..... | 296 |
| 6.35.1 Definitions | 297 |
| 6.35.2 BLAKE2B-384 digest..... | 297 |
| 6.35.3 General-length BLAKE2B-384-HMAC | 297 |
| 6.35.4 BLAKE2B-384-HMAC | 298 |
| 6.35.5 BLAKE2B-384 key derivation | 298 |

| | |
|--|-----|
| 6.35.6 BLAKE2B-384 HMAC key generation..... | 298 |
| 6.36 BLAKE2B-512..... | 298 |
| 6.36.1 Definitions | 299 |
| 6.36.2 BLAKE2B-512 digest..... | 299 |
| 6.36.3 General-length BLAKE2B-512-HMAC | 299 |
| 6.36.4 BLAKE2B-512-HMAC | 300 |
| 6.36.5 BLAKE2B-512 key derivation..... | 300 |
| 6.36.6 BLAKE2B-512 HMAC key generation..... | 300 |
| 6.37 PKCS #5 and PKCS #5-style password-based encryption (PBE)..... | 300 |
| 6.37.1 Definitions | 301 |
| 6.37.2 Password-based encryption/authentication mechanism parameters..... | 301 |
| 6.37.3 PKCS #5 PBKDF2 key generation mechanism parameters | 302 |
| 6.37.4 PKCS #5 PBKD2 key generation | 304 |
| 6.38 PKCS #12 password-based encryption/authentication mechanisms | 304 |
| 6.38.1 SHA-1-PBE for 3-key triple-DES-CBC | 305 |
| 6.38.2 SHA-1-PBE for 2-key triple-DES-CBC | 305 |
| 6.38.3 SHA-1-PBA for SHA-1-HMAC..... | 305 |
| 6.39 SSL | 305 |
| 6.39.1 Definitions | 306 |
| 6.39.2 SSL mechanism parameters | 306 |
| 6.39.3 Pre-master key generation | 308 |
| 6.39.4 Master key derivation | 308 |
| 6.39.5 Master key derivation for Diffie-Hellman | 309 |
| 6.39.6 Key and MAC derivation..... | 310 |
| 6.39.7 MD5 MACing in SSL 3.0 | 311 |
| 6.39.8 SHA-1 MACing in SSL 3.0 | 311 |
| 6.40 TLS 1.2 Mechanisms | 311 |
| 6.40.1 Definitions | 312 |
| 6.40.2 TLS 1.2 mechanism parameters | 312 |
| 6.40.3 TLS MAC | 315 |
| 6.40.4 Master key derivation | 315 |
| 6.40.5 Master key derivation for Diffie-Hellman | 316 |
| 6.40.6 Key and MAC derivation..... | 317 |
| 6.40.7 CKM_TLS12_KEY_SAFE_DERIVE..... | 317 |
| 6.40.8 Generic Key Derivation using the TLS PRF | 318 |
| 6.40.9 Generic Key Derivation using the TLS12 PRF..... | 318 |
| 6.41 WTLS | 319 |
| 6.41.1 Definitions | 319 |
| 6.41.2 WTLS mechanism parameters | 320 |
| 6.41.3 Pre master secret key generation for RSA key exchange suite..... | 322 |
| 6.41.4 Master secret key derivation | 323 |
| 6.41.5 Master secret key derivation for Diffie-Hellman and Elliptic Curve Cryptography | 323 |
| 6.41.6 WTLS PRF (pseudorandom function) | 324 |
| 6.41.7 Server Key and MAC derivation | 324 |
| 6.41.8 Client key and MAC derivation | 325 |
| 6.42 SP 800-108 Key Derivation | 326 |

| | |
|--|-----|
| 6.42.1 Definitions | 326 |
| 6.42.2 Mechanism Parameters | 327 |
| 6.42.3 Counter Mode KDF | 332 |
| 6.42.4 Feedback Mode KDF | 332 |
| 6.42.5 Double Pipeline Mode KDF | 333 |
| 6.42.6 Deriving Additional Keys | 334 |
| 6.42.7 Key Derivation Attribute Rules | 335 |
| 6.42.8 Constructing PRF Input Data | 335 |
| 6.42.8.1 Sample Counter Mode KDF | 335 |
| 6.42.8.2 Sample SCP03 Counter Mode KDF | 336 |
| 6.42.8.3 Sample Feedback Mode KDF | 337 |
| 6.42.8.4 Sample Double-Pipeline Mode KDF | 339 |
| 6.43 Miscellaneous simple key derivation mechanisms | 340 |
| 6.43.1 Definitions | 340 |
| 6.43.2 Parameters for miscellaneous simple key derivation mechanisms | 340 |
| 6.43.3 Concatenation of a base key and another key | 341 |
| 6.43.4 Concatenation of a base key and data | 341 |
| 6.43.5 Concatenation of data and a base key | 342 |
| 6.43.6 XORing of a key and data | 343 |
| 6.43.7 Extraction of one key from another key | 344 |
| 6.44 CMS | 344 |
| 6.44.1 Definitions | 345 |
| 6.44.2 CMS Signature Mechanism Objects | 345 |
| 6.44.3 CMS mechanism parameters | 345 |
| 6.44.4 CMS signatures | 346 |
| 6.45 Blowfish | 347 |
| 6.45.1 Definitions | 348 |
| 6.45.2 BLOWFISH secret key objects | 348 |
| 6.45.3 Blowfish key generation | 349 |
| 6.45.4 Blowfish-CBC | 349 |
| 6.45.5 Blowfish-CBC with PKCS padding | 349 |
| 6.46 Twofish | 350 |
| 6.46.1 Definitions | 350 |
| 6.46.2 Twofish secret key objects | 350 |
| 6.46.3 Twofish key generation | 351 |
| 6.46.4 Twofish -CBC | 351 |
| 6.46.5 Twofish-CBC with PKCS padding | 351 |
| 6.47 CAMELLIA | 351 |
| 6.47.1 Definitions | 352 |
| 6.47.2 Camellia secret key objects | 352 |
| 6.47.3 Camellia key generation | 353 |
| 6.47.4 Camellia-ECB | 353 |
| 6.47.5 Camellia-CBC | 354 |
| 6.47.6 Camellia-CBC with PKCS padding | 354 |
| 6.47.7 CAMELLIA with Counter mechanism parameters | 355 |
| 6.47.8 General-length Camellia-MAC | 356 |
| 6.47.9 Camellia-MAC | 356 |

| | |
|--|-----|
| 6.48 Key derivation by data encryption - Camellia | 357 |
| 6.48.1 Definitions | 357 |
| 6.48.2 Mechanism Parameters | 357 |
| 6.49 ARIA..... | 357 |
| 6.49.1 Definitions..... | 358 |
| 6.49.2 Aria secret key objects | 358 |
| 6.49.3 ARIA key generation | 359 |
| 6.49.4 ARIA-ECB..... | 359 |
| 6.49.5 ARIA-CBC | 359 |
| 6.49.6 ARIA-CBC with PKCS padding | 360 |
| 6.49.7 General-length ARIA-MAC | 361 |
| 6.49.8 ARIA-MAC | 361 |
| 6.50 Key derivation by data encryption - ARIA..... | 361 |
| 6.50.1 Definitions..... | 362 |
| 6.50.2 Mechanism Parameters | 362 |
| 6.51 SEED | 362 |
| 6.51.1 Definitions..... | 363 |
| 6.51.2 SEED secret key objects..... | 363 |
| 6.51.3 SEED key generation | 364 |
| 6.51.4 SEED-ECB | 364 |
| 6.51.5 SEED-CBC | 364 |
| 6.51.6 SEED-CBC with PKCS padding..... | 364 |
| 6.51.7 General-length SEED-MAC..... | 365 |
| 6.51.8 SEED-MAC..... | 365 |
| 6.52 Key derivation by data encryption - SEED | 365 |
| 6.52.1 Definitions..... | 365 |
| 6.52.2 Mechanism Parameters | 365 |
| 6.53 OTP..... | 366 |
| 6.53.1 Usage overview..... | 366 |
| 6.53.2 Case 1: Generation of OTP values | 366 |
| 6.53.3 Case 2: Verification of provided OTP values | 367 |
| 6.53.4 Case 3: Generation of OTP keys | 367 |
| 6.53.5 OTP objects..... | 368 |
| 6.53.5.1 Key objects | 368 |
| 6.53.6 OTP-related notifications..... | 371 |
| 6.53.7 OTP mechanisms | 371 |
| 6.53.7.1 OTP mechanism parameters | 371 |
| 6.53.8 RSA SecurID | 375 |
| 6.53.8.1 RSA SecurID secret key objects | 375 |
| 6.53.8.2 RSA SecurID key generation | 376 |
| 6.53.8.3 SecurID OTP generation and validation..... | 376 |
| 6.53.8.4 Return values..... | 377 |
| 6.53.9 OATH HOTP..... | 377 |
| 6.53.9.1 OATH HOTP secret key objects | 377 |
| 6.53.9.2 HOTP key generation | 378 |
| 6.53.9.3 HOTP OTP generation and validation..... | 378 |
| 6.53.10 ActivIdentity ACTI | 378 |

| | |
|---|-----|
| 6.53.10.1 ACTI secret key objects | 378 |
| 6.53.10.2 ACTI key generation | 379 |
| 6.53.10.3 ACTI OTP generation and validation | 379 |
| 6.54 CT-KIP | 380 |
| 6.54.1 Principles of Operation | 380 |
| 6.54.2 Mechanisms | 380 |
| 6.54.3 Definitions | 381 |
| 6.54.4 CT-KIP Mechanism parameters | 381 |
| 6.54.5 CT-KIP key derivation | 381 |
| 6.54.6 CT-KIP key wrap and key unwrap | 382 |
| 6.54.7 CT-KIP signature generation | 382 |
| 6.55 GOST 28147-89 | 382 |
| 6.55.1 Definitions | 382 |
| 6.55.2 GOST 28147-89 secret key objects | 383 |
| 6.55.3 GOST 28147-89 domain parameter objects | 383 |
| 6.55.4 GOST 28147-89 key generation | 384 |
| 6.55.5 GOST 28147-89-ECB | 385 |
| 6.55.6 GOST 28147-89 encryption mode except ECB | 385 |
| 6.55.7 GOST 28147-89-MAC | 386 |
| 6.55.8 GOST 28147-89 keys wrapping/unwrapping with GOST 28147-89 | 386 |
| 6.56 GOST R 34.11-94 | 387 |
| 6.56.1 Definitions | 387 |
| 6.56.2 GOST R 34.11-94 domain parameter objects | 387 |
| 6.56.3 GOST R 34.11-94 digest | 388 |
| 6.56.4 GOST R 34.11-94 HMAC | 389 |
| 6.57 GOST R 34.10-2001 | 389 |
| 6.57.1 Definitions | 389 |
| 6.57.2 GOST R 34.10-2001 public key objects | 390 |
| 6.57.3 GOST R 34.10-2001 private key objects | 391 |
| 6.57.4 GOST R 34.10-2001 domain parameter objects | 393 |
| 6.57.5 GOST R 34.10-2001 mechanism parameters | 394 |
| 6.57.6 GOST R 34.10-2001 key pair generation | 395 |
| 6.57.7 GOST R 34.10-2001 without hashing | 395 |
| 6.57.8 GOST R 34.10-2001 with GOST R 34.11-94 | 396 |
| 6.57.9 GOST 28147-89 keys wrapping/unwrapping with GOST R 34.10-2001 | 396 |
| 6.57.10 Common key derivation with assistance of GOST R 34.10-2001 keys | 396 |
| 6.58 ChaCha20 | 397 |
| 6.58.1 Definitions | 397 |
| 6.58.2 ChaCha20 secret key objects | 397 |
| 6.58.3 ChaCha20 mechanism parameters | 398 |
| 6.58.4 ChaCha20 key generation | 398 |
| 6.58.5 ChaCha20 mechanism | 398 |
| 6.59 Salsa20 | 399 |
| 6.59.1 Definitions | 400 |
| 6.59.2 Salsa20 secret key objects | 400 |
| 6.59.3 Salsa20 mechanism parameters | 401 |

| | |
|---|-----|
| 6.59.4 Salsa20 key generation | 401 |
| 6.59.5 Salsa20 mechanism | 401 |
| 6.60 Poly1305 | 402 |
| 6.60.1 Definitions | 402 |
| 6.60.2 Poly1305 secret key objects | 402 |
| 6.60.3 Poly1305 mechanism | 403 |
| 6.61 ChaCha20/Poly1305 and Salsa20/Poly1305 Authenticated Encryption / Decryption | 403 |
| 6.61.1 Definitions | 404 |
| 6.61.2 Usage | 404 |
| 6.61.3 ChaCha20/Poly1305 and Salsa20/Poly1305 Mechanism parameters | 406 |
| 6.62 HKDF Mechanisms | 407 |
| 6.62.1 Definitions | 407 |
| 6.62.2 HKDF mechanism parameters | 407 |
| 6.62.3 HKDF derive | 408 |
| 6.62.4 HKDF Data | 409 |
| 6.62.5 HKDF Key gen | 409 |
| 6.63 NULL Mechanism | 409 |
| 6.63.1 Definitions | 409 |
| 6.63.2 CKM_NULL mechanism parameters | 409 |
| 6.64 IKE Mechanisms | 409 |
| 6.64.1 Definitions | 410 |
| 6.64.2 IKE mechanism parameters | 410 |
| 6.64.3 IKE PRF DERIVE | 412 |
| 6.64.4 IKEv1 PRF DERIVE | 413 |
| 6.64.5 IKEv2 PRF PLUS DERIVE | 413 |
| 6.64.6 IKEv1 Extended Derive | 414 |
| 6.65 HSS | 414 |
| 6.65.1 Definitions | 415 |
| 6.65.2 HSS public key objects | 415 |
| 6.65.3 HSS private key objects | 416 |
| 6.65.4 HSS key pair generation | 418 |
| 6.65.5 HSS without hashing | 418 |
| 7 PKCS #11 Implementation Conformance | 420 |
| 7.1 PKCS#11 Consumer Implementation Conformance | 420 |
| 7.2 PKCS#11 Provider Implementation Conformance | 420 |
| Appendix A. Acknowledgments | 421 |
| Appendix B. Manifest constants | 423 |
| Appendix C. Revision History | 424 |
| Appendix D. Notices | 425 |

1 Introduction

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

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

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

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

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

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

1.1 Definitions

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

| | | |
|----|--------------------|--|
| 24 | AES | Advanced Encryption Standard, as defined in FIPS PUB 197. |
| 25 | API | Application programming interface. |
| 26 | Application | Any computer program that calls the Cryptoki interface. |
| 27 | ASN.1 | Abstract Syntax Notation One, as defined in X.680. |
| 28 | Attribute | A characteristic of an object. |
| 29 | BER | Basic Encoding Rules, as defined in X.690. |
| 30 | BLOWFISH | The Blowfish Encryption Algorithm of Bruce Schneier, www.schneier.com . |
| 32 | CAMELLIA | The Camellia encryption algorithm, as defined in RFC 3713. |
| 33 | CBC | Cipher-Block Chaining mode, as defined in FIPS PUB 81. |
| 34 | Certificate | A signed message binding a subject name and a public key, or a subject name and a set of attributes. |
| 36 | CDMF | Commercial Data Masking Facility, a block encipherment method specified by International Business Machines Corporation and based on DES. |
| 39 | CMAC | Cipher-based Message Authenticate Code as defined in [NIST sp800-38b] and [RFC 4493]. |
| 41 | CMS | Cryptographic Message Syntax (see RFC 5652) |

| | | |
|----|-----------------------------|--|
| 42 | Cryptographic Device | A device storing cryptographic information and possibly performing cryptographic functions. May be implemented as a smart card, smart disk, PCMCIA card, or with some other technology, including software-only. |
| 43 | | |
| 44 | | |
| 45 | | |
| 46 | Cryptoki | The Cryptographic Token Interface defined in this standard. |
| 47 | Cryptoki library | A library that implements the functions specified in this standard. |
| 48 | CT-KIP | Cryptographic Token Key Initialization Protocol (as defined in [CT-KIP]) |
| 49 | | |
| 50 | DER | Distinguished Encoding Rules, as defined in X.690. |
| 51 | DES | Data Encryption Standard, as defined in FIPS PUB 46-3. |
| 52 | DSA | Digital Signature Algorithm, as defined in FIPS PUB 186-4. |
| 53 | EC | Elliptic Curve |
| 54 | ECB | Electronic Codebook mode, as defined in FIPS PUB 81. |
| 55 | ECDH | Elliptic Curve Diffie-Hellman. |
| 56 | ECDSA | Elliptic Curve DSA, as in ANSI X9.62. |
| 57 | ECMQV | Elliptic Curve Menezes-Qu-Vanstone |
| 58 | GOST 28147-89 | The encryption algorithm, as defined in Part 2 [GOST 28147-89] and [RFC 4357] [RFC 4490], and RFC [4491]. |
| 59 | | |
| 60 | GOST R 34.11-94 | Hash algorithm, as defined in [GOST R 34.11-94] and [RFC 4357], [RFC 4490], and [RFC 4491]. |
| 61 | | |
| 62 | GOST R 34.10-2001 | The digital signature algorithm, as defined in [GOST R 34.10-2001] and [RFC 4357], [RFC 4490], and [RFC 4491]. |
| 63 | | |
| 64 | IV | Initialization Vector. |
| 65 | MAC | Message Authentication Code. |
| 66 | Mechanism | A process for implementing a cryptographic operation. |
| 67 | MQV | Menezes-Qu-Vanstone |
| 68 | OAEP | Optimal Asymmetric Encryption Padding for RSA. |
| 69 | Object | An item that is stored on a token. May be data, a certificate, or a key. |
| 70 | | |
| 71 | PIN | Personal Identification Number. |
| 72 | PKCS | Public-Key Cryptography Standards. |
| 73 | PRF | Pseudo random function. |
| 74 | PTD | Personal Trusted Device, as defined in MeT-PTD |
| 75 | RSA | The RSA public-key cryptosystem. |
| 76 | Reader | The means by which information is exchanged with a device. |
| 77 | Session | A logical connection between an application and a token. |
| 78 | SHA-1 | The (revised) Secure Hash Algorithm with a 160-bit message digest, as defined in FIPS PUB 180-2. |
| 79 | | |
| 80 | SHA-224 | The Secure Hash Algorithm with a 224-bit message digest, as defined in RFC 3874. Also defined in FIPS PUB 180-2 with Change Notice 1. |
| 81 | | |
| 82 | | |

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

1.2 Symbols and abbreviations

102 The following symbols are used in this standard:

103 *Table 1, Symbols*

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

104 The following prefixes are used in this standard:

105 *Table 2, Prefixes*

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

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

106

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

108

```

109  /* an unsigned 8-bit value */
110  typedef unsigned char CK_BYTE;
111
112  /* an unsigned 8-bit character */
113  typedef CK_BYTE CK_CHAR;
114
115  /* an 8-bit UTF-8 character */
116  typedef CK_BYTE CK_UTF8CHAR;
117
118  /* a BYTE-sized Boolean flag */
119  typedef CK_BYTE CK_BBOOL;
120
121  /* an unsigned value, at least 32 bits long */
122  typedef unsigned long int CK ULONG;
123
124  /* a signed value, the same size as a CK ULONG */
125  typedef long int CK_LONG;
126
127  /* at least 32 bits; each bit is a Boolean flag */
128  typedef CK ULONG CK_FLAGS;
129

```

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

```

132  CK_BYTE_PTR      /* Pointer to a CK_BYTE */
133  CK_CHAR_PTR      /* Pointer to a CK_CHAR */
134  CK_UTF8CHAR_PTR  /* Pointer to a CK_UTF8CHAR */
135  CK ULONG_PTR     /* Pointer to a CK ULONG */
136  CK VOID_PTR      /* Pointer to a void */
137

```

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

```

139  CK VOID_PTR_PTR  /* Pointer to a CK VOID_PTR */
140

```

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

```

142  NULL_PTR         /* A NULL pointer */
143

```

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

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

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

151 *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 | " " |

152 The **CK_UTF8CHAR** data type holds UTF-8 encoded Unicode characters as specified in RFC2279. UTF-
153 8 allows internationalization while maintaining backward compatibility with the Local String definition of
154 PKCS #11 version 2.01.

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

```
159 #define CK_FALSE 0
160 #define CK_TRUE 1
```

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

```
164 #ifndef CK_DISABLE_TRUE_FALSE
165 #ifndef FALSE
166 #define FALSE CK_FALSE
167 #endif
168
169 #ifndef TRUE
170 #define TRUE CK_TRUE
171 #endif
172 #endif
```

174 1.3 Normative References

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

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

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

183 **[CDMF]** Johnson, D.B. "The Commercial Data Masking Facility (CDMF) data privacy algorithm", March
184 1994.
185 URL: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5389557>

- 186 [CHACHA] D. Bernstein, "ChaCha, a variant of Salsa20", January 2008.
187 URL: <http://cr.yp.to/chacha/chacha-20080128.pdf>
- 188 [DH] W. Diffie, M. Hellman. "New Directions in Cryptography", November 1976.
189 URL: <http://www-ee.stanford.edu/~hellman/publications/24.pdf>
- 190 [FIPS PUB 46-3] NIST. "FIPS 46-3: Data Encryption Standard", October 1999.
191 URL: <http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf>
- 192 [FIPS PUB 81] NIST. "FIPS 81: DES Modes of Operation", December 1980.
193 URL: <http://csrc.nist.gov/publications/fips/fips81/fips81.htm>
- 194 [FIPS PUB 186-4] NIST. "FIPS 186-4: Digital Signature Standard", July 2013.
195 URL: <http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf>
- 196 [FIPS SP 800-56A] NIST. "Special Publication 800-56A Revision 2: Recommendation for Pair-Wise
197 Key Establishment Schemes Using Discrete Logarithm Cryptography" May 2013.
198 URL: <http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar2.pdf>
- 199 [FIPS SP 800-108] NIST. "Special Publication 800-108 (Revised): Recommendation for Key
200 Derivation Using Pseudorandom Functions", October 2009.
201 URL: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-108.pdf>
- 202 [GOST] V. Dolmatov, A. Degtyarev. "GOST R. 34.11-2012: Hash Function", August 2013.
203 URL: <https://datatracker.ietf.org/doc/html/rfc6986>
- 204 [MD2] B. Kaliski. RSA Laboratories. "The MD2 Message-Digest Algorithm", April 1992.
205 URL: <https://www.ietf.org/rfc/rfc1319.txt>
- 206 [MD5] RSA Data Security. R. Rivest. "The MD5 Message-Digest Algorithm", April 1992.
207 URL: <https://www.ietf.org/rfc/rfc1321.txt>
- 208 [NIST 802-208] NIST "Special Publication 800-208: Recommendation for Stateful Hash-Based Signature
209 Schemes", October 2020.
210 URL: <https://csrc.nist.gov/publications/detail/sp/800-208/final>
- 211 [OAEP] M. Bellare, P. Rogaway. "Optimal Asymmetric Encryption – How to Encrypt with RSA", November
212 1995.
- 213 [PKCS11-Hist] *PKCS #11 Cryptographic Token Interface Historical Mechanisms Specification Version
214 3.0. Edited by Chris Zimman and Dieter Bong. Latest stage. [https://docs.oasis-open.org/pkcs11/pkcs11-hist-v3.0.html](https://docs.oasis-open.org/pkcs11/pkcs11-
215 hist/v3.0/pkcs11-hist-v3.0.html)*
- 216 [PKCS11-Prof] *PKCS #11 Profiles Version 3.1. Edited by Tim Hudson. Latest stage: [https://docs.oasis-open.org/pkcs11/pkcs11-profiles-v3.1.html](https://docs.oasis-
217 open.org/pkcs11/pkcs11-profiles/v3.1/pkcs11-profiles-v3.1.html).*
- 218 [PKCS #1] K. Moriarty, B. Kaliski, J. Jonsson, A. Rusch. RFC 8017 "PKCS #1: RSA Cryptography
219 Specifications Version 2.2", November 2016
220 URL: <https://www.rfc-editor.org/rfc/pdfrfc/rfc8017.txt.pdf>
- 221 [PKCS #3] RSA Laboratories. "Diffie-Hellman Key-Agreement Standard. v1.4", November 1993.
222 URL: https://www.teletrust.de/fileadmin/files/oid/oid_pkcs-3v1-4.pdf
- 223 [PKCS #5] K. Moriarty, B. Kaliski, A. Rusch. RFC 8018. "PKCS #5: Password-Based Cryptography
224 Specification Version 2.1", January 2017
225 URL: <https://www.rfc-editor.org/rfc/pdfrfc/rfc8018.txt.pdf>
- 226 [PKCS #7] B. Kaliski. "PKCS #7 Cryptographic Message Syntax Version 1.5", March 1998
227 URL: <https://www.rfc-editor.org/rfc/pdfrfc/rfc2315.txt.pdf>
- 228 [PKCS #8] B. Kaliski. RFC 5208 "Public-Key Cryptography Standards (PKCS) #8: Private-Key
229 Information Syntax Specification Version 1.2", May 2008, obsoleted by RFC 5258 S.Turner "Asymmetric
230 Key Packages", August 2010
231 URL: <https://www.rfc-editor.org/rfc/pdfrfc/rfc5958.txt.pdf>

- 232 [PKCS #12] K. Moriarty, M. Nystrom, S. Parkinson, A. Rusch, M. Scott. "PKCS #12 Personal
233 Information Exchange Syntax v1.1", July 2014.
234 URL: <https://www.rfc-editor.org/rfc/pdfrfc/rfc7292.txt.pdf>
- 235 [POLY1305] D.J. Bernstein. "The Poly1305-AES message-authentication code", January 2005.
236 URL: <https://cr.yp.to/mac/poly1305-20050329.pdf>
- 237 [RFC 2409] D. Harkins, D.Carrel. RFC 2409: "The Internet Key Exchange (IKE)", November 1998.
238 URL: <https://tools.ietf.org/html/rfc2409>
- 239 [RFC 2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC
240 2119, March 1997.
241 URL: <http://www.ietf.org/rfc/rfc2119.txt>.
- 242 [RFC 2279] F. Yergeau. RFC 2279: "UTF-8, a transformation format of ISO 10646 Alis
243 Technologies", January 1998.
244 URL: <http://www.ietf.org/rfc/rfc2279.txt>
- 245 [RFC 2534] Masinter, L., Wing, D., Mutz, A., and K. Holtman. RFC 2534: "Media Features for Display,
246 Print, and Fax". March 1999.
247 URL: <http://www.ietf.org/rfc/rfc2534.txt>
- 248 [RFC 5652] R. Housley. RFC 5652: "Cryptographic Message Syntax", September 2009. URL:
249 <http://www.ietf.org/rfc/rfc5652.txt>
- 250 [RFC 5707] Rescorla, E., "The Keying Material Exporters for Transport Layer Security (TLS)", RFC
251 5705, March 2010.
252 URL: <http://www.ietf.org/rfc/rfc5705.txt>
- 253 [RFC 5996] C. Kaufman, P. Hoffman, Y. Nir, P. Eronen. RFC 5996: "Internet Key Exchange Protocol
254 Version 2 (IKEv2)", September 2010.
255 URL: <https://tools.ietf.org/html/rfc5996>
- 256 [RFC 8554] D. McGrew, m. Curcio, S. Fluhrer. RFC 8554: "Leighton-Micali Hash-Based Signatures",
257 April 2019.
258 URL: <https://tools.ietf.org/html/rfc8554>
- 259 [RIPEMD] H. Dobbertin, A. Bosselaers, B. Preneel. "The hash function RIPEMD-160", February
260 2012.
261 URL: <http://homes.esat.kuleuven.be/~bosselaer/ripemd160.html>
- 262 [SALSA] D. Bernstein, "ChaCha, a variant of Salsa20", January 2008.
263 URL: <http://cr.yp.to/chacha/chacha-20080128.pdf>
- 264 [SHA-1] NIST. FIPS 180-4: "Secure Hash Standard", March 2012.
265 URL: <http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf>
- 266 [SHA-2] NIST. FIPS 180-4: "Secure Hash Standard", March 2012.
267 URL: <http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf>
- 268 [TLS] [RFC2246] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, January 1999.
269 URL: <http://www.ietf.org/rfc/rfc2246.txt>, superseded by [RFC4346] Dierks, T. and E. Rescorla, "The
270 Transport Layer Security (TLS) Protocol Version 1.1", RFC 4346, April 2006.
271 URL: <http://www.ietf.org/rfc/rfc4346.txt>, which was superseded by [TLS12].
- 272 [TLS12] [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol
273 Version 1.2", RFC 5246, August 2008.
274 URL: <http://www.ietf.org/rfc/rfc5246.txt>
- 275 [TWOFISH] B. Schneier, J. Kelsey, D. Whiting, C. Hall, N. Ferguson. "Twofish: A 128-Bit Block
276 Cipher", June 1998.
277 URL: <https://www.schneier.com/academic/twofish/>
- 278 [X.500] ITU-T. Information Technology — Open Systems Interconnection — The Directory: Overview of
279 Concepts, Models and Services. February 2001. Identical to ISO/IEC 9594-1

- 280 [X.509] ITU-T. Information Technology — Open Systems Interconnection — The Directory: Public-key
281 and Attribute Certificate Frameworks. March 2000. Identical to ISO/IEC 9594-8
- 282 [X.680] ITU-T. Information Technology — Abstract Syntax Notation One (ASN.1): Specification of Basic
283 Notation. July 2002. Identical to ISO/IEC 8824-1
- 284 [X.690] ITU-T. Information Technology — ASN.1 Encoding Rules: Specification of Basic Encoding Rules
285 (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER). July 2002. Identical
286 to ISO/IEC 8825-1

287

288 1.4 Non-Normative References

- 289 [CAP-1.2] Common Alerting Protocol Version 1.2. 01 July 2010. OASIS Standard.
290 URL: <http://docs.oasis-open.org/emergency/cap/v1.2/CAP-v1.2-os.html>
- 291 [AES KEYWRAP] National Institute of Standards and Technology, NIST Special Publication 800-
292 38F, Recommendation for Block Cipher Modes of Operation: Methods for Key Wrapping, December
293 2012, <http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-38F.pdf>
- 294 [ANSI C] ANSI/ISO. American National Standard for Programming Languages – C. 1990.
- 295 [ANSI X9.31] Accredited Standards Committee X9. Digital Signatures Using Reversible Public Key
296 Cryptography for the Financial Services Industry (rDSA). 1998.
- 297 [ANSI X9.42] Accredited Standards Committee X9. Public Key Cryptography for the Financial Services
298 Industry: Agreement of Symmetric Keys Using Discrete Logarithm Cryptography. 2003.
- 299 [ANSI X9.62] Accredited Standards Committee X9. Public Key Cryptography for the Financial Services
300 Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA). 1998.
- 301 [ANSI X9.63] Accredited Standards Committee X9. Public Key Cryptography for the Financial Services
302 Industry: Key Agreement and Key Transport Using Elliptic Curve Cryptography. 2001.
303 URL: <http://webstore.ansi.org/RecordDetail.aspx?sku=X9.63-2011>
- 304 [BRAINPOOL] ECC Brainpool Standard Curves and Curve Generation, v1.0, 19.10.2005
305 URL: <http://www.ecc-brainpool.org>
- 306 [CC/PP] W3C. Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies.
307 World Wide Web Consortium, January 2004.
308 URL: <http://www.w3.org/TR/CCPP-struct-vocab/>
- 309 [CDPD] Ameritech Mobile Communications et al. Cellular Digital Packet Data System Specifications: Part
310 406: Airlink Security. 1993.
- 311 [CT-KIP] RSA Laboratories. Cryptographic Token Key Initialization Protocol. Version 1.0,
312 December 2005.
- 313 [GCS-API] X/Open Company Ltd. Generic Cryptographic Service API (GCS-API), Base - Draft 2.
314 February 14, 1995.
- 315 [ISO/IEC 7816-1] ISO. Information Technology — Identification Cards — Integrated Circuit(s) with
316 Contacts — Part 1: Physical Characteristics. 1998.
- 317 [ISO/IEC 7816-4] ISO. Information Technology — Identification Cards — Integrated Circuit(s) with
318 Contacts — Part 4: Interindustry Commands for Interchange. 1995.
- 319 [ISO/IEC 8824-1] ISO. Information Technology-- Abstract Syntax Notation One (ASN.1):
320 Specification of Basic Notation. 2002.
- 321 [ISO/IEC 8825-1] ISO. Information Technology—ASN.1 Encoding Rules: Specification of Basic
322 Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER).
323 2002.

- 324 **[ISO/IEC 9594-1]** ISO. Information Technology — Open Systems Interconnection — The Directory:
325 Overview of Concepts, Models and Services. 2001.
- 326 **[ISO/IEC 9594-8]** ISO. Information Technology — Open Systems Interconnection — The Directory:
327 Public-key and Attribute Certificate Frameworks. 2001
- 328 **[ISO/IEC 9796-2]** ISO. Information Technology — Security Techniques — Digital Signature
329 Scheme Giving Message Recovery — Part 2: Integer factorization based mechanisms. 2002.
- 330 **[Java MIDP]** Java Community Process. Mobile Information Device Profile for Java 2 Micro Edition.
331 November 2002.
332 URL: <http://jcp.org/jsr/detail/118.jsp>
- 333 **[LEGIFRANCE]**Avis relatif aux paramètres de courbes elliptiques définis par l'Etat français (Publication of
334 Elliptic Curve parameters by the French state)
335 URL: <https://www.legifrance.gouv.fr/affichTexte.do?cidTexte=JORFTEXT000024668816>
- 336 **[MeT-PTD]** MeT. MeT PTD Definition – Personal Trusted Device Definition, Version 1.0, February
337 2003.
338 URL: <http://www.mobiletransaction.org>
- 339 **[NIST AES CTS]** National Institute of Standards and Technology, Addendum to NIST Special
340 Publication 800-38A, "Recommendation for Block Cipher Modes of Operation: Three Variants of
341 Ciphertext Stealing for CBC Mode"
342 URL: http://csrc.nist.gov/publications/nistpubs/800-38a/addendum-to-nist_sp800-38A.pdf
- 343 **[PCMCIA]** Personal Computer Memory Card International Association. *PC Card Standard*,
344 Release 2.1., July 1993.
- 345 **[RFC 2865]** Rigney et al, "Remote Authentication Dial In User Service (RADIUS)", IETF RFC2865,
346 June 2000.
347 URL: <http://www.ietf.org/rfc/rfc2865.txt>.
- 348 **[RFC 3686]** Housley, "Using Advanced Encryption Standard (AES) Counter Mode With IPsec
349 Encapsulating Security Payload (ESP)," IETF RFC 3686, January 2004.
350 URL: <http://www.ietf.org/rfc/rfc3686.txt>.
- 351 **[RFC 3717]** Matsui, et al, "A Description of the Camellia Encryption Algorithm," IETF RFC 3717, April
352 2004.
353 URL: <http://www.ietf.org/rfc/rfc3713.txt>.
- 354 **[RFC 3610]** Whiting, D., Housley, R., and N. Ferguson, "Counter with CBC-MAC (CCM)", IETF RFC
355 3610, September 2003.
356 URL: <http://www.ietf.org/rfc/rfc3610.txt>
- 357 **[RFC 3874]** Smit et al, "A 224-bit One-way Hash Function: SHA-224," IETF RFC 3874, June 2004.
358 URL: <http://www.ietf.org/rfc/rfc3874.txt>.
- 359 **[RFC 3748]** Aboba et al, "Extensible Authentication Protocol (EAP)", IETF RFC 3748, June 2004.
360 URL: <http://www.ietf.org/rfc/rfc3748.txt>.
- 361 **[RFC 4269]** South Korean Information Security Agency (KISA) "The SEED Encryption Algorithm",
362 December 2005.
363 URL: <https://ftp.rfc-editor.org/in-notes/rfc4269.txt>
- 364 **[RFC 4309]** Housley, R., "Using Advanced Encryption Standard (AES) CCM Mode with IPsec
365 Encapsulating Security Payload (ESP)," IETF RFC 4309, December 2005.
366 URL: <http://www.ietf.org/rfc/rfc4309.txt>
- 367 **[RFC 4357]** V. Popov, I. Kurepkin, S. Leontiev "Additional Cryptographic Algorithms for Use with
368 GOST 28147-89, GOST R 34.10-94, GOST R 34.10-2001, and GOST R 34.11-94 Algorithms", January
369 2006.
370 URL: <http://www.ietf.org/rfc/rfc4357.txt>

- 371 [RFC 4490] S. Leontiev, Ed. G. Chudov, Ed. "Using the GOST 28147-89, GOST R 34.11-94, GOST
372 R 34.10-94, and GOST R 34.10-2001 Algorithms with Cryptographic Message Syntax (CMS)", May 2006.
373 URL: <http://www.ietf.org/rfc/rfc4490.txt>
- 374 [RFC 4491] S. Leontiev, Ed., D. Shefanovski, Ed., "Using the GOST R 34.10-94, GOST R 34.10-
375 2001, and GOST R 34.11-94 Algorithms with the Internet X.509 Public Key Infrastructure Certificate and
376 CRL Profile", May 2006.
377 URL: <http://www.ietf.org/rfc/rfc4491.txt>
- 378 [RFC 4493] J. Song et al. *RFC 4493: The AES-CMAC Algorithm*. June 2006.
379 URL: <http://www.ietf.org/rfc/rfc4493.txt>
- 380 [RFC 5705] Rescorla, E., "The Keying Material Exporters for Transport Layer Security (TLS)", RFC
381 5705, March 2010.
382 URL: <http://www.ietf.org/rfc/rfc5705.txt>
- 383 [RFC 5869] H. Krawczyk, P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function
384 (HKDF)", May 2010
385 URL: <http://www.ietf.org/rfc/rfc5869.txt>
- 386 [RFC 7539] Y Nir, A. Langley. *RFC 7539: ChaCha20 and Poly1305 for IETF Protocols*, May 2015
387 URL: <https://tools.ietf.org/rfc/rfc7539.txt>
- 388 [RFC 7748] Aboba et al, "Elliptic Curves for Security", IETF RFC 7748, January 2016
389 URL: <https://tools.ietf.org/html/rfc7748>
- 390 [RFC 8032] Aboba et al, "Edwards-Curve Digital Signature Algorithm (EdDSA)", IETF RFC 8032,
391 January 2017
392 URL: <https://tools.ietf.org/html/rfc8032>
- 393 [SEC 1] Standards for Efficient Cryptography Group (SECG). *Standards for Efficient Cryptography (SEC)*
394 1: Elliptic Curve Cryptography. Version 1.0, September 20, 2000.
- 395 [SEC 2] Standards for Efficient Cryptography Group (SECG). Standards for Efficient Cryptography (SEC)
396 2: Recommended Elliptic Curve Domain Parameters. Version 1.0, September 20, 2000.
- 397 [WTLS] WAP. Wireless Transport Layer Security Version — WAP-261-WTLS-20010406-a. April 2001.
398 URL: <http://openmobilealliance.org/tech/affiliates/wap/wap-261-wtls-20010406-a.pdf>
- 399 [XEDDSA] The XEdDSA and VXEdDSA Signature Schemes - Revision 1, 2016-10-20, Trevor Perrin
400 (editor)
401 URL: <https://signal.org/docs/specifications/xeddsa/>
- 402 [X.500] ITU-T. Information Technology — Open Systems Interconnection — The Directory: Overview of
403 Concepts, Models and Services. February 2001. Identical to ISO/IEC 9594-1
- 404 [X.509] ITU-T. Information Technology — Open Systems Interconnection — The Directory: Public-key
405 and Attribute Certificate Frameworks. March 2000. Identical to ISO/IEC 9594-8
- 406 [X.680] ITU-T. Information Technology — Abstract Syntax Notation One (ASN.1): Specification of Basic
407 Notation. July 2002. Identical to ISO/IEC 8824-1
- 408 [X.690] ITU-T. Information Technology — ASN.1 Encoding Rules: Specification of Basic Encoding Rules
409 (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER). July 2002. Identical
410 to ISO/IEC 8825-1

411 2 Platform- and compiler-dependent directives for C 412 or C++

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

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

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

420 2.1 Structure packing

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

423 2.2 Pointer-related macros

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

426 ◆ CK_PTR

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

```
429    typedef CK_BYTE CK_PTR CK_BYTE_PTR;
```

430 ◆ CK_DECLARE_FUNCTION

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

```
434    CK_DECLARE_FUNCTION(CK_RV, C_Initialize) (
435     CK_VOID_PTR pReserved
436 );
```

437 ◆ CK_DECLARE_FUNCTION_POINTER

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

```
444    CK_DECLARE_FUNCTION_POINTER(CK_RV, myC_Initialize) (
445     CK_VOID_PTR pReserved
446 );
447
```

448 or:

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

```
450     CK_VOID_PTR pReserved  
451 ) ;  
452 myC_InitializeType myC_Initialize;
```

453 ◆ **CK_CALLBACK_FUNCTION**

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

```
461 CK_CALLBACK_FUNCTION(CK_RV, myCallback) (args);  
462
```

463 or:

```
464 typedef CK_CALLBACK_FUNCTION(CK_RV, myCallbackType) (args);  
465 myCallbackType myCallback;
```

466 ◆ **NULL_PTR**

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

469 3 General data types

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

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

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

480 3.1 General information

481 Cryptoki represents general information with the following types:

482 ♦ CK_VERSION; CK_VERSION_PTR

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

```
486     typedef struct CK_VERSION {
487         CK_BYTE major;
488         CK_BYTE minor;
489     } CK_VERSION;
```

491 The fields of the structure have the following meanings:

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

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

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

496 *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 |

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

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

500 ♦ CK_INFO; CK_INFO_PTR

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

```
502     typedef struct CK_INFO {
503         CK_VERSION cryptokiVersion;
504         CK_UTF8CHAR manufacturerID[32];
505         CK_FLAGS flags;
506         CK_UTF8CHAR libraryDescription[32];
507         CK_VERSION libraryVersion;
508     } CK_INFO;
```

510 The fields of the structure have the following meanings:

| | | |
|-----|---------------------------|---|
| 511 | <i>cryptokiVersion</i> | Cryptoki interface version number, for compatibility with future revisions of this interface |
| 512 | <i>manufacturerID</i> | ID of the Cryptoki library manufacturer. MUST be padded with the blank character (' '). Should <i>not</i> be null-terminated. |
| 513 | <i>flags</i> | bit flags reserved for future versions. MUST be zero for this version |
| 514 | <i>libraryDescription</i> | character-string description of the library. MUST be padded with the blank character (' '). Should <i>not</i> be null-terminated. |
| 515 | <i>libraryVersion</i> | Cryptoki library version number |

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

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

522 ♦ CK_NOTIFICATION

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

```
525     typedef CK ULONG CK_NOTIFICATION;
```

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

```
528     CKN_SURRENDER
```

530 The notifications have the following meanings:

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

536 3.2 Slot and token types

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

538 ♦ CK_SLOT_ID; CK_SLOT_ID_PTR

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

```
540     typedef CK ULONG CK_SLOT_ID;
```

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

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

546 ◆ **CK_SLOT_INFO; CK_SLOT_INFO_PTR**

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

```
548     typedef struct CK_SLOT_INFO {  
549         CK_UTF8CHAR slotDescription[64];  
550         CK_UTF8CHAR manufacturerID[32];  
551         CK_FLAGS flags;  
552         CK_VERSION hardwareVersion;  
553         CK_VERSION firmwareVersion;  
554     } CK_SLOT_INFO;
```

556 The fields of the structure have the following meanings:

557 *slotDescription* character-string description of the slot. MUST be padded with the
558 blank character (' '). MUST NOT be null-terminated.

559 *manufacturerID* ID of the slot manufacturer. MUST be padded with the blank
560 character (' '). MUST NOT be null-terminated.

561 *flags* bits flags that provide capabilities of the slot. The flags are defined
562 below

563 *hardwareVersion* version number of the slot's hardware

564 *firmwareVersion* version number of the slot's firmware

565 The following table defines the *flags* field:

566 *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" |

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

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

571 ◆ **CK_TOKEN_INFO; CK_TOKEN_INFO_PTR**

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

```
573     typedef struct CK_TOKEN_INFO {  
574         CK_UTF8CHAR label[32];  
575         CK_UTF8CHAR manufacturerID[32];  
576         CK_UTF8CHAR model[16];  
577         CK_CHAR serialNumber[16];  
578         CK_FLAGS flags;  
579         CK ULONG ulMaxSessionCount;  
580         CK ULONG ulSessionCount;
```

```

581     CK ULONG ulMaxRwSessionCount;
582     CK ULONG ulRwSessionCount;
583     CK ULONG ulMaxPinLen;
584     CK ULONG ulMinPinLen;
585     CK ULONG ulTotalPublicMemory;
586     CK ULONG ulFreePublicMemory;
587     CK ULONG ulTotalPrivateMemory;
588     CK ULONG ulFreePrivateMemory;
589     CK VERSION hardwareVersion;
590     CK VERSION firmwareVersion;
591     CK CHAR utcTime[16];
592 } CK_TOKEN_INFO;
593

```

594 The fields of the structure have the following meanings:

| | | |
|-----|-----------------------------|---|
| 595 | <i>label</i> | application-defined label, assigned during token initialization. MUST be padded with the blank character (' '). MUST NOT be null-terminated. |
| 596 | | |
| 597 | | |
| 598 | <i>manufacturerID</i> | ID of the device manufacturer. MUST be padded with the blank character (' '). MUST NOT be null-terminated. |
| 599 | | |
| 600 | <i>model</i> | model of the device. MUST be padded with the blank character (' '). MUST NOT be null-terminated. |
| 601 | | |
| 602 | <i>serialNumber</i> | character-string serial number of the device. MUST be padded with the blank character (' '). MUST NOT be null-terminated. |
| 603 | | |
| 604 | <i>flags</i> | bit flags indicating capabilities and status of the device as defined below |
| 605 | | |
| 606 | <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) |
| 607 | | |
| 608 | | |
| 609 | <i>ulSessionCount</i> | number of sessions that this application currently has open with the token (see CK_TOKEN_INFO Note below) |
| 610 | | |
| 611 | <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) |
| 612 | | |
| 613 | | |
| 614 | <i>ulRwSessionCount</i> | number of read/write sessions that this application currently has open with the token (see CK_TOKEN_INFO Note below) |
| 615 | | |
| 616 | <i>ulMaxPinLen</i> | maximum length in bytes of the PIN |
| 617 | <i>ulMinPinLen</i> | minimum length in bytes of the PIN |
| 618 | <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) |
| 619 | | |
| 620 | <i>ulFreePublicMemory</i> | the amount of free (unused) memory on the token in bytes for public objects (see CK_TOKEN_INFO Note below) |
| 621 | | |
| 622 | <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) |
| 623 | | |
| 624 | <i>ulFreePrivateMemory</i> | the amount of free (unused) memory on the token in bytes for private objects (see CK_TOKEN_INFO Note below) |
| 625 | | |
| 626 | <i>hardwareVersion</i> | version number of hardware |
| 627 | <i>firmwareVersion</i> | version number of firmware |

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

634 The following table defines the *flags* field:

635 *Table 6, Token Information Flags*

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

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

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

- 639 • Creating/modifying/deleting any object on the token.
 640 • Creating/modifying/deleting a token object on the token.
 641 • Changing the SO's PIN.
 642 • Changing the normal user's PIN.

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

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

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

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

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

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

```
670     CK_TOKEN_INFO info;
671     .
672     .
673     if ((CK_LONG) info.ulMaxSessionCount
674         == CK_UNAVAILABLE_INFORMATION) {
675         /* Token refuses to give value of ulMaxSessionCount */
676         .
677         .
678     } else if (info.ulMaxSessionCount == CK_EFFECTIVELY_INFINITE) {
679         /* Application can open as many sessions as it wants */
680         .
681         .
682     } else {
683         /* ulMaxSessionCount really does contain what it should */
684         .
685         .
686     }
```

688 CK_TOKEN_INFO_PTR is a pointer to a CK_TOKEN_INFO.

689 **3.3 Session types**

690 Cryptoki represents session information with the following types:

691 ◆ **CK_SESSION_HANDLE; CK_SESSION_HANDLE_PTR**

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

```
693     typedef CK ULONG CK_SESSION_HANDLE;
694 
```

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

```
697     CK_INVALID_HANDLE
698 
```

699 CK_SESSION_HANDLE_PTR is a pointer to a CK_SESSION_HANDLE.

700 ◆ **CK_USER_TYPE**

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

```
703     typedef CK ULONG CK_USER_TYPE;
```

704

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

706 CKU_SO
707 CKU_USER
708 CKU_CONTEXT_SPECIFIC

709 ◆ **CK_STATE**

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

711 `typedef CK ULONG CK_STATE;`
712

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

714 CKS_RO_PUBLIC_SESSION
715 CKS_RO_USER_FUNCTIONS
716 CKS_RW_PUBLIC_SESSION
717 CKS_RW_USER_FUNCTIONS
718 CKS_RW_SO_FUNCTIONS

719 ◆ **CK_SESSION_INFO; CK_SESSION_INFO_PTR**

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

721 `typedef struct CK_SESSION_INFO {`
722 `CK_SLOT_ID slotID;`
723 `CK_STATE state;`
724 `CK_FLAGS flags;`
725 `CK ULONG ulDeviceError;`
726 `} CK_SESSION_INFO;`
727

728

729 The fields of the structure have the following meanings:

730 `slotID` ID of the slot that interfaces with the token
731 `state` the state of the session
732 `flags` bit flags that define the type of session; the flags are defined below
733 `ulDeviceError` an error code defined by the cryptographic device. Used for errors
734 not covered by Cryptoki.

735 The following table defines the `flags` field:

736 *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 |

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

738 **3.4 Object types**

739 Cryptoki represents object information with the following types:

740 ◆ **CK_OBJECT_HANDLE; CK_OBJECT_HANDLE_PTR**

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

```
742     typedef CK ULONG CK_OBJECT_HANDLE;
```

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

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

```
752     CK_INVALID_HANDLE
```

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

755 ◆ **CK_OBJECT_CLASS; CK_OBJECT_CLASS_PTR**

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

```
758     typedef CK ULONG CK_OBJECT_CLASS;
```

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

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

```
763     CKO_VENDOR_DEFINED
```

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

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

768 ◆ **CK_HW_FEATURE_TYPE**

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

```
771     typedef CK ULONG CK_HW_FEATURE_TYPE;
```

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

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

```
776     CKH_VENDOR_DEFINED
```

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

780 ◆ **CK_KEY_TYPE**

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

```
782     typedef CK ULONG CK_KEY_TYPE;
```

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

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

```
787     CKK_VENDOR_DEFINED
```

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

791 ◆ **CK_CERTIFICATE_TYPE**

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

```
793     typedef CK ULONG CK_CERTIFICATE_TYPE;
```

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

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

```
798     CKC_VENDOR_DEFINED
```

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

802 ◆ **CK_CERTIFICATE_CATEGORY**

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

```
804     typedef CK ULONG CK_CERTIFICATE_CATEGORY;
```

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

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

807 ◆ **CK_ATTRIBUTE_TYPE**

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

```
809     typedef CK ULONG CK_ATTRIBUTE_TYPE;
```

811 Attributes are defined with the objects and mechanisms that use them. Attributes are specified on an
812 object as a list of type, length value items. These are often specified as an attribute template.
813 Vendor defined values for this type may also be specified.

```
814     CKA_VENDOR_DEFINED
```

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

818 ◆ **CK_ATTRIBUTE; CK_ATTRIBUTE_PTR**

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

```
821     typedef struct CK_ATTRIBUTE {  
822         CK_ATTRIBUTE_TYPE type;  
823         CK_VOID_PTR pValue;  
824         CK ULONG ulValueLen;  
825     } CK_ATTRIBUTE;
```

827 The fields of the structure have the following meanings:

828 *type* the attribute type

829 *pValue* pointer to the value of the attribute

830 *ulValueLen* length in bytes of the value

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

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

840

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

842 ◆ **CK_DATE**

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

```
844     typedef struct CK_DATE {  
845         CK_CHAR year[4];  
846         CK_CHAR month[2];  
847         CK_CHAR day[2];  
848     } CK_DATE;
```

850 The fields of the structure have the following meanings:

851 *year* the year (“1900” - “9999”)

852 *month* the month (“01” - “12”)

853 *day* the day (“01” - “31”)

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

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

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

860 ◆ **CK_PROFILE_ID; CK_PROFILE_ID_PTR**

861 **CK_PROFILE_ID** is an unsigend ulong value represting a specific token profile. It is defined as follows:

```
862     typedef CK ULONG CK_PROFILE_ID;
```

864 Profiles are defines in the PKCS #11 Cryptographic Token Interface Profiles document. s. ID's greater
865 than 0xffffffff may cause compatibility issues on platforms that have CK ULONG values of 32 bits, and
866 should be avoided.

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

```
868     CKP_VENDOR_DEFINED
```

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

872
873 *Valid Profile IDs in Cryptoki always have nonzero values.* For developers' convenience, Cryptoki defines
874 the following symbolic value:

```
875     CKP_INVALID_ID
```

876 CK_PROFILE_ID_PTR is a pointer to a CK_PROFILE_ID.

877 ◆ **CK_JAVA_MIDP_SECURITY_DOMAIN**

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

```
880     typedef CK ULONG CK_JAVA_MIDP_SECURITY_DOMAIN;
```

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

| Constant | Value | Meaning |
|---------------------------------|---------------|--------------------------------|
| CK_SECURITY_DOMAIN_UNSPECIFIED | 0x000000000UL | 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 |

883

884 **3.5 Data types for mechanisms**

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

886 ◆ **CK_MECHANISM_TYPE; CK_MECHANISM_TYPE_PTR**

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

```
888     typedef CK ULONG CK_MECHANISM_TYPE;
```

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

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

```
892     CKM_VENDOR_DEFINED
```

894 Mechanism types **CKM_VENDOR_DEFINED** and above are permanently reserved for token vendors.

895 For interoperability, vendors should register their mechanism types through the PKCS process.

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

897 ◆ **CK_MECHANISM; CK_MECHANISM_PTR**

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

```
900     typedef struct CK_MECHANISM {  
901         CK_MECHANISM_TYPE mechanism;  
902         CK_VOID_PTR pParameter;  
903         CK ULONG ulParameterLen;  
904     } CK_MECHANISM;
```

906 The fields of the structure have the following meanings:

907 **mechanism** the type of mechanism

908 **pParameter** pointer to the parameter if required by the mechanism

909 **ulParameterLen** length in bytes of the parameter

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

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

914 ◆ **CK_MECHANISM_INFO; CK_MECHANISM_INFO_PTR**

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

```
917     typedef struct CK_MECHANISM_INFO {  
918         CK ULONG ulMinKeySize;  
919         CK ULONG ulMaxKeySize;  
920         CK_FLAGS flags;  
921     } CK_MECHANISM_INFO;
```

923 The fields of the structure have the following meanings:

924 **ulMinKeySize** the minimum size of the key for the mechanism (whether this is
925 measured in bits or in bytes is mechanism-dependent)

926 **ulMaxKeySize** the maximum size of the key for the mechanism (whether this is
927 measured in bits or in bytes is mechanism-dependent)

928 **flags** bit flags specifying mechanism capabilities

- 929 For some mechanisms, the *ulMinKeySize* and *ulMaxKeySize* fields have meaningless values.
 930 The following table defines the *flags* field:
 931 *Table 8, Mechanism Information Flags*

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

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

932 CK_MECHANISM_INFO_PTR is a pointer to a CK_MECHANISM_INFO.

933 3.6 Function types

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

935 ♦ CK_RV

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

```
937     typedef CK ULONG CK_RV;
938 
```

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

```
940     CKR_VENDOR_DEFINED
941 
```

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

945 ♦ CK_NOTIFY

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

```
948     typedef CK_CALLBACK_FUNCTION(CK_RV, CK_NOTIFY) (
949         CK_SESSION_HANDLE hSession,
950         CK_NOTIFICATION event,
951         CK_VOID_PTR pApplication
952     );
953 
```

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

| | |
|-------------------------|---|
| 955 <i>hSession</i> | The handle of the session performing the callback |
| 956 <i>event</i> | The type of notification callback |
| 957 <i>pApplication</i> | An application-defined value. This is the same value as was passed to C_OpenSession to open the session performing the callback |

959 ♦ CK_C_XXX

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

965 ◆ **CK_FUNCTION_LIST; CK_FUNCTION_LIST_PTR;**
966 **CK_FUNCTION_LIST_PTR_PTR**

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

```
969     typedef struct CK_FUNCTION_LIST {
970         CK_VERSION version;
971         CK_C_Initialize C_Initialize;
972         CK_C_Finalize C_Finalize;
973         CK_C_GetInfo C_GetInfo;
974         CK_C_GetFunctionList C_GetFunctionList;
975         CK_C_GetSlotList C_GetSlotList;
976         CK_C_GetSlotInfo C_GetSlotInfo;
977         CK_C_GetTokenInfo C_GetTokenInfo;
978         CK_C_GetMechanismList C_GetMechanismList;
979         CK_C_GetMechanismInfo C_GetMechanismInfo;
980         CK_C_InitToken C_InitToken;
981         CK_C_InitPIN C_InitPIN;
982         CK_C_SetPIN C_SetPIN;
983         CK_C_OpenSession C_OpenSession;
984         CK_C_CloseSession C_CloseSession;
985         CK_C_CloseAllSessions C_CloseAllSessions;
986         CK_C_GetSessionInfo C_GetSessionInfo;
987
988         CK_C_GetOperationState C_GetOperationState;
989         CK_C_SetOperationState C_SetOperationState;
990         CK_C_Login C_Login;
991         CK_C_Logout C_Logout;
992         CK_C_CreateObject C_CreateObject;
993         CK_C_CopyObject C_CopyObject;
994         CK_C_DestroyObject C_DestroyObject;
995         CK_C_GetObjectSize C_GetObjectSize;
996         CK_C_GetAttributeValue C_GetAttributeValue;
997         CK_C_SetAttributeValue C_SetAttributeValue;
998         CK_C_FindObjectsInit C_FindObjectsInit;
999         CK_C_FindObjects C_FindObjects;
1000        CK_C_FindObjectsFinal C_FindObjectsFinal;
1001        CK_C_EncryptInit C_EncryptInit;
1002        CK_C_Encrypt C_Encrypt;
1003        CK_C_EncryptUpdate C_EncryptUpdate;
1004        CK_C_EncryptFinal C_EncryptFinal;
1005        CK_C_DecryptInit C_DecryptInit;
1006        CK_C_Decrypt C_Decrypt;
1007        CK_C_DecryptUpdate C_DecryptUpdate;
1008        CK_C_DecryptFinal C_DecryptFinal;
1009        CK_C_DigestInit C_DigestInit;
1010        CK_C_Digest C_Digest;
1011        CK_C_DigestUpdate C_DigestUpdate;
1012        CK_C_DigestKey C_DigestKey;
1013        CK_C_DigestFinal C_DigestFinal;
1014        CK_C_SignInit C_SignInit;
1015        CK_C_Sign C_Sign;
1016        CK_C_SignUpdate C_SignUpdate;
1017        CK_C_SignFinal C_SignFinal;
1018        CK_C_SignRecoverInit C_SignRecoverInit;
1019        CK_C_SignRecover C_SignRecover;
1020        CK_C_VerifyInit C_VerifyInit;
1021        CK_C_Verify C_Verify;
1022        CK_C_VerifyUpdate C_VerifyUpdate;
1023        CK_C_VerifyFinal C_VerifyFinal;
1024        CK_C_VerifyRecoverInit C_VerifyRecoverInit;
1025        CK_C_VerifyRecover C_VerifyRecover;
```

```

1026     CK_C_DigestEncryptUpdate C_DigestEncryptUpdate;
1027     CK_C_DecryptDigestUpdate C_DecryptDigestUpdate;
1028     CK_C_SignEncryptUpdate C_SignEncryptUpdate;
1029     CK_C_DecryptVerifyUpdate C_DecryptVerifyUpdate;
1030     CK_C_GenerateKey C_GenerateKey;
1031     CK_C_GenerateKeyPair C_GenerateKeyPair;
1032     CK_C_WrapKey C_WrapKey;
1033     CK_C_UnwrapKey C_UnwrapKey;
1034     CK_C_DeriveKey C_DeriveKey;
1035     CK_C_SeedRandom C_SeedRandom;
1036     CK_C_GenerateRandom C_GenerateRandom;
1037     CK_C_GetFunctionStatus C_GetFunctionStatus;
1038     CK_C_CancelFunction C_CancelFunction;
1039     CK_C_WaitForSlotEvent C_WaitForSlotEvent;
1040 } CK_FUNCTION_LIST;
1041

```

1042 Each Cryptoki library has a static **CK_FUNCTION_LIST** structure, and a pointer to it (or to a copy of it
1043 which is also owned by the library) may be obtained by the **C_GetFunctionList** function (see Section
1044 5.2). The value that this pointer points to can be used by an application to quickly find out where the
1045 executable code for each function in the Cryptoki API is located. Every function in the Cryptoki API
1046 MUST have an entry point defined in the Cryptoki library's **CK_FUNCTION_LIST** structure. If a particular
1047 function in the Cryptoki API is not supported by a library, then the function pointer for that function in the
1048 library's **CK_FUNCTION_LIST** structure should point to a function stub which simply returns
1049 **CKR_FUNCTION_NOT_SUPPORTED**.

1050 In this structure 'version' is the cryptoki specification version number. The major and minor versions must
1051 be set to 0x02 and 0x28 indicating a version 2.40 compatible structure. The updated function list table for
1052 this version of the specification may be returned via **C_GetInterfaceList** or **C_GetInterface**.
1053

1054 An application may or may not be able to modify a Cryptoki library's static **CK_FUNCTION_LIST**
1055 structure. Whether or not it can, it should never attempt to do so.

1056 PKCS #11 modules must not add new functions at the end of the **CK_FUNCTION_LIST** that are not
1057 contained within the defined structure. If a PKCS#11 module needs to define additional functions, they
1058 should be placed within a vendor defined interface returned via **C_GetInterfaceList** or **C_GetInterface**.

1059 **CK_FUNCTION_LIST_PTR** is a pointer to a **CK_FUNCTION_LIST**.

1060 **CK_FUNCTION_LIST_PTR_PTR** is a pointer to a **CK_FUNCTION_LIST_PTR**.

1061

1062 ◆ **CK_FUNCTION_LIST_3_0; CK_FUNCTION_LIST_3_0_PTR;**
1063 **CK_FUNCTION_LIST_3_0_PTR_PTR**

1064 **CK_FUNCTION_LIST_3_0** is a structure which contains the same function pointers as in
1065 **CK_FUNCTION_LIST** and additional functions added to the end of the structure that were defined in
1066 Cryptoki version 3.0. It is defined as follows:

```

1067     typedef struct CK_FUNCTION_LIST_3_0 {
1068         CK_VERSION version;
1069         CK_C_Initialize C_Initialize;
1070         CK_C_Finalize C_Finalize;
1071         CK_C_GetInfo C_GetInfo;
1072         CK_C_GetFunctionList C_GetFunctionList;
1073         CK_C_GetSlotList C_GetSlotList;
1074         CK_C_GetSlotInfo C_GetSlotInfo;
1075         CK_C_GetTokenInfo C_GetTokenInfo;
1076         CK_C_GetMechanismList C_GetMechanismList;
1077         CK_C_GetMechanismInfo C_GetMechanismInfo;
1078         CK_C_InitToken C_InitToken;
1079         CK_C_InitPIN C_InitPIN;

```

```
1080     CK_C_SetPIN C_SetPIN;
1081     CK_C_OpenSession C_OpenSession;
1082     CK_C_CloseSession C_CloseSession;
1083     CK_C_CloseAllSessions C_CloseAllSessions;
1084     CK_C_GetSessionInfo C_GetSessionInfo;
1085     CK_C_GetOperationState C_GetOperationState;
1086     CK_C_SetOperationState C_SetOperationState;
1087     CK_C_Login C_Login;
1088     CK_C_Logout C_Logout;
1089     CK_C_CreateObject C_CreateObject;
1090     CK_C_CopyObject C_CopyObject;
1091     CK_C_DestroyObject C_DestroyObject;
1092     CK_C_GetObjectSize C_GetObjectSize;
1093     CK_C_GetAttributeValue C_GetAttributeValue;
1094     CK_C_SetAttributeValue C_SetAttributeValue;
1095     CK_C_FindObjectsInit C_FindObjectsInit;
1096     CK_C_FindObjects C_FindObjects;
1097     CK_C_FindObjectsFinal C_FindObjectsFinal;
1098     CK_C_EncryptInit C_EncryptInit;
1099     CK_C_Encrypt C_Encrypt;
1100     CK_C_EncryptUpdate C_EncryptUpdate;
1101     CK_C_EncryptFinal C_EncryptFinal;
1102     CK_C_DecryptInit C_DecryptInit;
1103     CK_C_Decrypt C_Decrypt;
1104     CK_C_DecryptUpdate C_DecryptUpdate;
1105     CK_C_DecryptFinal C_DecryptFinal;
1106     CK_C_DigestInit C_DigestInit;
1107     CK_C_Digest C_Digest;
1108     CK_C_DigestUpdate C_DigestUpdate;
1109     CK_C_DigestKey C_DigestKey;
1110     CK_C_DigestFinal C_DigestFinal;
1111     CK_C_SignInit C_SignInit;
1112     CK_C_Sign C_Sign;
1113     CK_C_SignUpdate C_SignUpdate;
1114     CK_C_SignFinal C_SignFinal;
1115     CK_C_SignRecoverInit C_SignRecoverInit;
1116     CK_C_SignRecover C_SignRecover;
1117     CK_C_VerifyInit C_VerifyInit;
1118     CK_C_Verify C_Verify;
1119     CK_C_VerifyUpdate C_VerifyUpdate;
1120     CK_C_VerifyFinal C_VerifyFinal;
1121     CK_C_VerifyRecoverInit C_VerifyRecoverInit;
1122     CK_C_VerifyRecover C_VerifyRecover;
1123     CK_C_DigestEncryptUpdate C_DigestEncryptUpdate;
1124     CK_C_DecryptDigestUpdate C_DecryptDigestUpdate;
1125     CK_C_SignEncryptUpdate C_SignEncryptUpdate;
1126     CK_C_DecryptVerifyUpdate C_DecryptVerifyUpdate;
1127     CK_C_GenerateKey C_GenerateKey;
1128     CK_C_GenerateKeyPair C_GenerateKeyPair;
1129     CK_C_WrapKey C_WrapKey;
1130     CK_C_UnwrapKey C_UnwrapKey;
1131     CK_C_DeriveKey C_DeriveKey;
1132     CK_C_SeedRandom C_SeedRandom;
1133     CK_C_GenerateRandom C_GenerateRandom;
1134     CK_C_GetFunctionStatus C_GetFunctionStatus;
1135     CK_C_CancelFunction C_CancelFunction;
1136     CK_C_WaitForSlotEvent C_WaitForSlotEvent;
1137     CK_C_GetInterfaceList C_GetInterfaceList;
1138     CK_C_GetInterface C_GetInterface;
1139     CK_C_LoginUser C_LoginUser;
1140     CK_C_SessionCancel C_SessionCancel;
1141     CK_C_MessageEncryptInit C_MessageEncryptInit;
1142     CK_C_EncryptMessage C_EncryptMessage;
1143     CK_C_EncryptMessageBegin C_EncryptMessageBegin;
```

```

1144     CK_C_EncryptMessageNext C_EncryptMessageNext;
1145     CK_C_MessageEncryptFinal C_MessageEncryptFinal;
1146     CK_C_MessageDecryptInit C_MessageDecryptInit;
1147     CK_C_DecryptMessage C_DecryptMessage;
1148     CK_C_DecryptMessageBegin C_DecryptMessageBegin;
1149     CK_C_DecryptMessageNext C_DecryptMessageNext;
1150     CK_C_MessageDecryptFinal C_MessageDecryptFinal;
1151     CK_C_MessageSignInit C_MessageSignInit;
1152     CK_C_SignMessage C_SignMessage;
1153     CK_C_SignMessageBegin C_SignMessageBegin;
1154     CK_C_SignMessageNext C_SignMessageNext;
1155     CK_C_MessageSignFinal C_MessageSignFinal;
1156     CK_C_MessageVerifyInit C_MessageVerifyInit;
1157     CK_C_VerifyMessage C_VerifyMessage;
1158     CK_C_VerifyMessageBegin C_VerifyMessageBegin;
1159     CK_C_VerifyMessageNext C_VerifyMessageNext;
1160     CK_C_MessageVerifyFinal C_MessageVerifyFinal;
1161 } CK_FUNCTION_LIST_3_0;
1162

```

1163 For a general description of **CK_FUNCTION_LIST_3_0** see **CK_FUNCTION_LIST**.

1164 In this structure, *version* is the cryptoki specification version number. It should match the value of
1165 *cryptokiVersion* returned in the **CK_INFO** structure, but must be 3.0 at minimum.

1166 This function list may be returned via **C_GetInterfaceList** or **C_GetInterface**

1167 **CK_FUNCTION_LIST_3_0_PTR** is a pointer to a **CK_FUNCTION_LIST_3_0**.

1168 **CK_FUNCTION_LIST_3_0_PTR_PTR** is a pointer to a **CK_FUNCTION_LIST_3_0_PTR**.

1169 ♦ **CK_INTERFACE; CK_INTERFACE_PTR; CK_INTERFACE_PTR_PTR**

1170 **CK_INTERFACE** is a structure which contains an interface name with a function list and flag.

1171 It is defined as follows:

```

1172     typedef struct CK_INTERFACE {
1173         CK_UTF8CHAR_PTR pInterfaceName;
1174         CK_VOID_PTR     pFunctionList;
1175         CK_FLAGS        flags;
1176     } CK_INTERFACE;

```

1177

1178 The fields of the structure have the following meanings:

1179 *pInterfaceName* the name of the interface

1180 *pFunctionList* the interface function list which must always begin with a
1181 **CK_VERSION** structure as the first field

1182 *flags* bit flags specifying interface capabilities

1183 The interface name "PKCS 11" is reserved for use by interfaces defined within the cryptoki specification.

1184 Interfaces starting with the string: "Vendor " are reserved for vendor use and will not otherwise be
1185 defined as interfaces in the PKCS #11 specification. Vendors should supply new functions with interface
1186 names of "Vendor {vendor name}". For example "Vendor ACME Inc".

1187

1188 The following table defines the flags field:

1189 *Table 9, CK_INTERFACE Flags*

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

1190

1191 **CK_INTERFACE_PTR** is a pointer to a **CK_INTERFACE**.1192 **CK_INTERFACE_PTR_PTR** is a pointer to a **CK_INTERFACE_PTR**.1193

3.7 Locking-related types

1194 The types in this section are provided solely for applications which need to access Cryptoki from multiple
1195 threads simultaneously. *Applications which will not do this need not use any of these types.*1196 ◆ **CK_CREATEMUTEX**1197 **CK_CREATEMUTEX** is the type of a pointer to an application-supplied function which creates a new
1198 mutex object and returns a pointer to it. It is defined as follows:

```
1199     typedef CK_CALLBACK_FUNCTION(CK_RV, CK_CREATEMUTEX) (
1200         CK_VOID_PTR_PTR ppMutex
1201     );

```

1203 Calling a **CK_CREATEMUTEX** function returns the pointer to the new mutex object in the location pointed
1204 to by ppMutex. Such a function should return one of the following values:

```
1205     CKR_OK, CKR_GENERAL_ERROR
1206     CKR_HOST_MEMORY
```

1207 ◆ **CK_DESTROYMUTEX**1208 **CK_DESTROYMUTEX** is the type of a pointer to an application-supplied function which destroys an
1209 existing mutex object. It is defined as follows:

```
1210     typedef CK_CALLBACK_FUNCTION(CK_RV, CK_DESTROYMUTEX) (
1211         CK_VOID_PTR pMutex
1212     );

```

1214 The argument to a **CK_DESTROYMUTEX** function is a pointer to the mutex object to be destroyed. Such
1215 a function should return one of the following values:

```
1216     CKR_OK, CKR_GENERAL_ERROR
1217     CKR_HOST_MEMORY
1218     CKR_MUTEX_BAD
```

1219 ◆ **CK_LOCKMUTEX** and **CK_UNLOCKMUTEX**

1220 **CK_LOCKMUTEX** is the type of a pointer to an application-supplied function which locks an existing
1221 mutex object. **CK_UNLOCKMUTEX** is the type of a pointer to an application-supplied function which
1222 unlocks an existing mutex object. The proper behavior for these types of functions is as follows:

- 1223 • If a CK_LOCKMUTEX function is called on a mutex which is not locked, the calling thread obtains a
1224 lock on that mutex and returns.
- 1225 • If a CK_LOCKMUTEX function is called on a mutex which is locked by some thread other than the
1226 calling thread, the calling thread blocks and waits for that mutex to be unlocked.
- 1227 • If a CK_LOCKMUTEX function is called on a mutex which is locked by the calling thread, the
1228 behavior of the function call is undefined.
- 1229 • If a CK_UNLOCKMUTEX function is called on a mutex which is locked by the calling thread, that
1230 mutex is unlocked and the function call returns. Furthermore:
- 1231 ◦ If exactly one thread was blocking on that particular mutex, then that thread stops blocking,
1232 obtains a lock on that mutex, and its CK_LOCKMUTEX call returns.
- 1233 ◦ If more than one thread was blocking on that particular mutex, then exactly one of the
1234 blocking threads is selected somehow. That lucky thread stops blocking, obtains a lock on
1235 the mutex, and its CK_LOCKMUTEX call returns. All other threads blocking on that particular
1236 mutex continue to block.
- 1237 • If a CK_UNLOCKMUTEX function is called on a mutex which is not locked, then the function call
1238 returns the error code CKR_MUTEX_NOT_LOCKED.
- 1239 • If a CK_UNLOCKMUTEX function is called on a mutex which is locked by some thread other than the
1240 calling thread, the behavior of the function call is undefined.

1241 **CK_LOCKMUTEX** is defined as follows:

```
1242     typedef CK_CALLBACK_FUNCTION(CK_RV, CK_LOCKMUTEX) (
1243         CK_VOID_PTR pMutex
1244     );
```

1246 The argument to a CK_LOCKMUTEX function is a pointer to the mutex object to be locked. Such a
1247 function should return one of the following values:

```
1248     CKR_OK, CKR_GENERAL_ERROR
1249     CKR_HOST_MEMORY,
1250     CKR_MUTEX_BAD
1251
```

1252 **CK_UNLOCKMUTEX** is defined as follows:

```
1253     typedef CK_CALLBACK_FUNCTION(CK_RV, CK_UNLOCKMUTEX) (
1254         CK_VOID_PTR pMutex
1255     );
```

1257 The argument to a CK_UNLOCKMUTEX function is a pointer to the mutex object to be unlocked. Such a
1258 function should return one of the following values:

```
1259     CKR_OK, CKR_GENERAL_ERROR
1260     CKR_HOST_MEMORY
1261     CKR_MUTEX_BAD
1262     CKR_MUTEX_NOT_LOCKED
```

1263 ◆ **CK_C_INITIALIZE_ARGS; CK_C_INITIALIZE_ARGS_PTR**

1264 **CK_C_INITIALIZE_ARGS** is a structure containing the optional arguments for the **C_Initialize** function.
1265 For this version of Cryptoki, these optional arguments are all concerned with the way the library deals
1266 with threads. **CK_C_INITIALIZE_ARGS** is defined as follows:

```
1267     typedef struct CK_C_INITIALIZE_ARGS {  
1268         CK_CREATEMUTEX CreateMutex;  
1269         CK_DESTROYMUTEX DestroyMutex;  
1270         CK_LOCKMUTEX LockMutex;  
1271         CK_UNLOCKMUTEX UnlockMutex;  
1272         CK_FLAGS flags;  
1273         CK_VOID_PTR pReserved;  
1274     } CK_C_INITIALIZE_ARGS;  
1275
```

1276 The fields of the structure have the following meanings:

| | | |
|------|---------------------|--|
| 1277 | <i>CreateMutex</i> | pointer to a function to use for creating mutex objects |
| 1278 | <i>DestroyMutex</i> | pointer to a function to use for destroying mutex objects |
| 1279 | <i>LockMutex</i> | pointer to a function to use for locking mutex objects |
| 1280 | <i>UnlockMutex</i> | pointer to a function to use for unlocking mutex objects |
| 1281 | <i>flags</i> | bit flags specifying options for C_Initialize ; the flags are defined below |
| 1282 | <i>pReserved</i> | reserved for future use. Should be NULL_PTR for this version of Cryptoki |

1285 The following table defines the flags field:

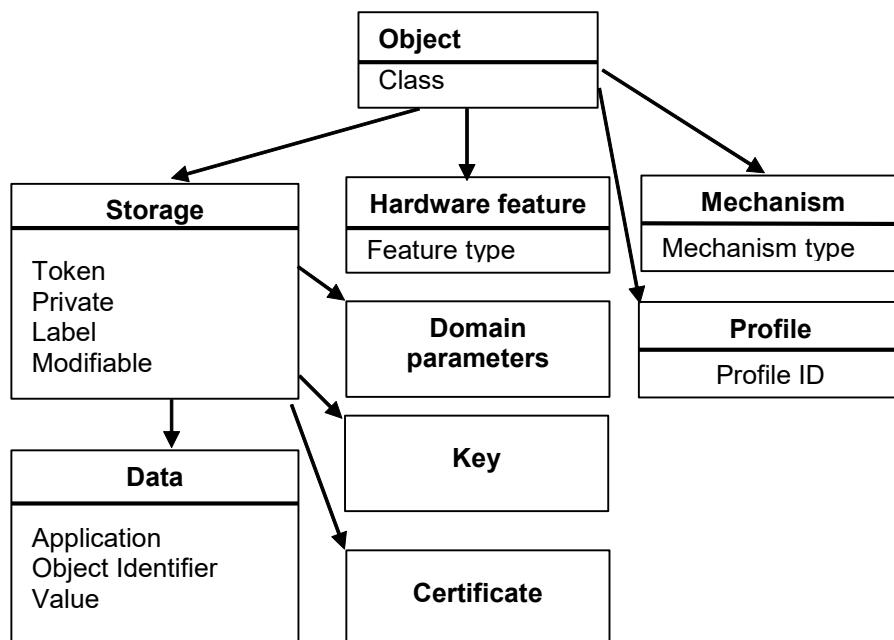
1286 *Table 10, C_Initialize Parameter Flags*

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

1287 **CK_C_INITIALIZE_ARGS_PTR** is a pointer to a **CK_C_INITIALIZE_ARGS**.

1288 4 Objects

1289 Cryptoki recognizes a number of classes of objects, as defined in the **CK_OBJECT_CLASS** data type.
1290 An object consists of a set of attributes, each of which has a given value. Each attribute that an object
1291 possesses has precisely one value. The following figure illustrates the high-level hierarchy of the
1292 Cryptoki objects and some of the attributes they support:



1293
1294 *Figure 1, Object Attribute Hierarchy*

1295 Cryptoki provides functions for creating, destroying, and copying objects in general, and for obtaining and
1296 modifying the values of their attributes. Some of the cryptographic functions (e.g., **C_GenerateKey**) also
1297 create key objects to hold their results.

1298 Objects are always “well-formed” in Cryptoki—that is, an object always contains all required attributes,
1299 and the attributes are always consistent with one another from the time the object is created. This
1300 contrasts with some object-based paradigms where an object has no attributes other than perhaps a
1301 class when it is created, and is uninitialized for some time. In Cryptoki, objects are always initialized.

1302 Tables throughout most of Section 4 define each Cryptoki attribute in terms of the data type of the
1303 attribute value and the meaning of the attribute, which may include a default initial value. Some of the
1304 data types are defined explicitly by Cryptoki (e.g., **CK_OBJECT_CLASS**). Attribute values may also take
1305 the following types:

| | | |
|------|----------------|---|
| 1306 | Byte array | an arbitrary string (array) of CK_BYTEs |
| 1307 | 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) |
| 1308 | Local string | an unpadded string of CK_CHARS (see Table 3) with no null-termination |
| 1309 | RFC2279 string | an unpadded string of CK_UTF8CHARs with no null-termination |

1313 A token can hold several identical objects, *i.e.*, it is permissible for two or more objects to have exactly the
1314 same values for all their attributes.

1315 In most cases each type of object in the Cryptoki specification possesses a completely well-defined set of
1316 Cryptoki attributes. Some of these attributes possess default values, and need not be specified when
1317 creating an object; some of these default values may even be the empty string (""). Nonetheless, the
1318 object possesses these attributes. A given object has a single value for each attribute it possesses, even
1319 if the attribute is a vendor-specific attribute whose meaning is outside the scope of Cryptoki.

1320 In addition to possessing Cryptoki attributes, objects may possess additional vendor-specific attributes
1321 whose meanings and values are not specified by Cryptoki.

1322 **4.1 Creating, modifying, and copying objects**

1323 All Cryptoki functions that create, modify, or copy objects take a template as one of their arguments,
1324 where the template specifies attribute values. Cryptographic functions that create objects (see Section
1325 5.18) may also contribute some additional attribute values themselves; which attributes have values
1326 contributed by a cryptographic function call depends on which cryptographic mechanism is being
1327 performed (see section 6 Mechanisms and [PKCS11-Hist] for specification of mechanisms for PKCS
1328 #11). In any case, all the required attributes supported by an object class that do not have default values
1329 MUST be specified when an object is created, either in the template or by the function itself.

1330 **4.1.1 Creating objects**

1331 Objects may be created with the Cryptoki functions **C_CreateObject** (see Section 5.7), **C_GenerateKey**,
1332 **C_GenerateKeyPair**, **C_UnwrapKey**, and **C_DeriveKey** (see Section 5.18). In addition, copying an
1333 existing object (with the function **C_CopyObject**) also creates a new object, but we consider this type of
1334 object creation separately in Section 4.1.3.

1335 Attempting to create an object with any of these functions requires an appropriate template to be
1336 supplied.

1337 1. If the supplied template specifies a value for an invalid attribute, then the attempt should fail with the
1338 error code **CKR_ATTRIBUTE_TYPE_INVALID**. An attribute is valid if it is either one of the attributes
1339 described in the Cryptoki specification or an additional vendor-specific attribute supported by the library
1340 and token.

1341 2. If the supplied template specifies an invalid value for a valid attribute, then the attempt should fail with
1342 the error code **CKR_ATTRIBUTE_VALUE_INVALID**. The valid values for Cryptoki attributes are
1343 described in the Cryptoki specification.

1344 3. If the supplied template specifies a value for a read-only attribute, then the attempt should fail with the
1345 error code **CKR_ATTRIBUTE_READ_ONLY**. Whether or not a given Cryptoki attribute is read-only is
1346 explicitly stated in the Cryptoki specification; however, a particular library and token may be even more
1347 restrictive than Cryptoki specifies. In other words, an attribute which Cryptoki says is not read-only may
1348 nonetheless be read-only under certain circumstances (*i.e.*, in conjunction with some combinations of
1349 other attributes) for a particular library and token. Whether or not a given non-Cryptoki attribute is read-
1350 only is obviously outside the scope of Cryptoki.

1351 4. If the attribute values in the supplied template, together with any default attribute values and any
1352 attribute values contributed to the object by the object-creation function itself, are insufficient to fully
1353 specify the object to create, then the attempt should fail with the error code
1354 **CKR_TEMPLATE_INCOMPLETE**.

1355 5. If the attribute values in the supplied template, together with any default attribute values and any
1356 attribute values contributed to the object by the object-creation function itself, are inconsistent, then the
1357 attempt should fail with the error code **CKR_TEMPLATE_INCONSISTENT**. A set of attribute values is
1358 inconsistent if not all of its members can be satisfied simultaneously by the token, although each value
1359 individually is valid in Cryptoki. One example of an inconsistent template would be using a template
1360 which specifies two different values for the same attribute. Another example would be trying to create
1361 a secret key object with an attribute which is appropriate for various types of public keys or private keys,
1362 but not for secret keys. A final example would be a template with an attribute that violates some token

1363 specific requirement. Note that this final example of an inconsistent template is token-dependent—on
1364 a different token, such a template might *not* be inconsistent.

- 1365 6. If the supplied template specifies the same value for a particular attribute more than once (or the
1366 template specifies the same value for a particular attribute that the object-creation function itself
1367 contributes to the object), then the behavior of Cryptoki is not completely specified. The attempt to
1368 create an object can either succeed—thereby creating the same object that would have been created
1369 if the multiply-specified attribute had only appeared once—or it can fail with error code
1370 CKR_TEMPLATE_INCONSISTENT. Library developers are encouraged to make their libraries behave
1371 as though the attribute had only appeared once in the template; application developers are strongly
1372 encouraged never to put a particular attribute into a particular template more than once.

1373 If more than one of the situations listed above applies to an attempt to create an object, then the error
1374 code returned from the attempt can be any of the error codes from above that applies.

1375 4.1.2 Modifying objects

1376 Objects may be modified with the Cryptoki function **C_SetAttributeValue** (see Section 5.7). The
1377 template supplied to **C_SetAttributeValue** can contain new values for attributes which the object already
1378 possesses; values for attributes which the object does not yet possess; or both.

1379 Some attributes of an object may be modified after the object has been created, and some may not. In
1380 addition, attributes which Cryptoki specifies are modifiable may actually *not* be modifiable on some
1381 tokens. That is, if a Cryptoki attribute is described as being modifiable, that really means only that it is
1382 modifiable *insofar as the Cryptoki specification is concerned*. A particular token might not actually
1383 support modification of some such attributes. Furthermore, whether or not a particular attribute of an
1384 object on a particular token is modifiable might depend on the values of certain attributes of the object.
1385 For example, a secret key object's **CKA_SENSITIVE** attribute can be changed from CK_FALSE to
1386 CK_TRUE, but not the other way around.

1387 All the scenarios in Section 4.1.1—and the error codes they return—apply to modifying objects with
1388 **C_SetAttributeValue**, except for the possibility of a template being incomplete.

1389 4.1.3 Copying objects

1390 Unless an object's **CKA_COPYABLE** (see Table 17) attribute is set to CK_FALSE, it may be copied with
1391 the Cryptoki function **C_CopyObject** (see Section 5.7). In the process of copying an object,
1392 **C_CopyObject** also modifies the attributes of the newly-created copy according to an application-
1393 supplied template.

1394 The Cryptoki attributes which can be modified during the course of a **C_CopyObject** operation are the
1395 same as the Cryptoki attributes which are described as being modifiable, plus the four special attributes
1396 **CKA_TOKEN**, **CKA_PRIVATE**, **CKA_MODIFIABLE** and **CKA_DESTROYABLE**. To be more precise,
1397 these attributes are modifiable during the course of a **C_CopyObject** operation *insofar as the Cryptoki*
1398 *specification is concerned*. A particular token might not actually support modification of some such
1399 attributes during the course of a **C_CopyObject** operation. Furthermore, whether or not a particular
1400 attribute of an object on a particular token is modifiable during the course of a **C_CopyObject** operation
1401 might depend on the values of certain attributes of the object. For example, a secret key object's
1402 **CKA_SENSITIVE** attribute can be changed from CK_FALSE to CK_TRUE during the course of a
1403 **C_CopyObject** operation, but not the other way around.

1404 If the **CKA_COPYABLE** attribute of the object to be copied is set to CK_FALSE, **C_CopyObject** returns
1405 CKR_ACTION_PROHIBITED. Otherwise, the scenarios described in 10.1.1 - and the error codes they
1406 return - apply to copying objects with **C_CopyObject**, except for the possibility of a template being
1407 incomplete.

1408 4.2 Common attributes

1409 *Table 11, Common footnotes for object attribute tables*

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

1410

1411 *Table 12, Common Object Attributes*

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

1412 Refer to Table 11 for footnotes

1413 The above table defines the attributes common to all objects.

1414 **4.3 Hardware Feature Objects**

1415 **4.3.1 Definitions**

1416 This section defines the object class CKO_HW_FEATURE for type CK_OBJECT_CLASS as used in the
1417 CKA_CLASS attribute of objects.

1418 **4.3.2 Overview**

1419 Hardware feature objects (**CKO_HW_FEATURE**) represent features of the device. They provide an easily
1420 expandable method for introducing new value-based features to the Cryptoki interface.

1421 When searching for objects using **C_FindObjectsInit** and **C_FindObjects**, hardware feature objects are
1422 not returned unless the **CKA_CLASS** attribute in the template has the value **CKO_HW_FEATURE**. This
1423 protects applications written to previous versions of Cryptoki from finding objects that they do not
1424 understand.

1425 *Table 13, Hardware Feature Common Attributes*

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

1426 - Refer to Table 11 for footnotes

1427 **4.3.3 Clock**

1428 **4.3.3.1 Definition**

1429 The CKA_HW_FEATURE_TYPE attribute takes the value CKH_CLOCK of type
1430 CK_HW_FEATURE_TYPE.

1431 **4.3.3.2 Description**

1432 Clock objects represent real-time clocks that exist on the device. This represents the same clock source
1433 as the **utcTime** field in the **CK_TOKEN_INFO** structure.

1434 *Table 14, Clock Object Attributes*

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

1435 The **CKA_VALUE** attribute may be set using the **C_SetAttributeValue** function if permitted by the
1436 device. The session used to set the time MUST be logged in. The device may require the SO to be the
1437 user logged in to modify the time value. **C_SetAttributeValue** will return the error
1438 CKR_USER_NOT_LOGGED_IN to indicate that a different user type is required to set the value.

1439 **4.3.4 Monotonic Counter Objects**

1440 **4.3.4.1 Definition**

1441 The CKA_HW_FEATURE_TYPE attribute takes the value CKH_MONOTONIC_COUNTER of type
1442 CK_HW_FEATURE_TYPE.

1443 **4.3.4.2 Description**

1444 Monotonic counter objects represent hardware counters that exist on the device. The counter is
1445 guaranteed to increase each time its value is read, but not necessarily by one. This might be used by an
1446 application for generating serial numbers to get some assurance of uniqueness per token.

1447 *Table 15, Monotonic Counter Attributes*

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

1448 ¹Read Only

1449 The **CKA_VALUE** attribute may not be set by the client.

1450 **4.3.5 User Interface Objects**

1451 **4.3.5.1 Definition**

1452 The CKA_HW_FEATURE_TYPE attribute takes the value CKH_USER_INTERFACE of type
1453 CK_HW_FEATURE_TYPE.

1454 **4.3.5.2 Description**

1455 User interface objects represent the presentation capabilities of the device.

1456 *Table 16, User Interface Object Attributes*

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

1457 The selection of attributes, and associated data types, has been done in an attempt to stay as aligned
1458 with RFC 2534 and CC/PP Struct as possible. The special value CK_UNAVAILABLE_INFORMATION
1459 may be used for CK ULONG-based attributes when information is not available or applicable.

1460 None of the attribute values may be set by an application.

1461 The value of the **CKA_ENCODING_METHODS** attribute may be used when the application needs to
1462 send MIME objects with encoded content to the token.

1463 **4.4 Storage Objects**

1464 This is not an object class; hence no CKO_ definition is required. It is a category of object classes with
1465 common attributes for the object classes that follow.

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

1467 Only the **CKA_LABEL** attribute can be modified after the object is created. (The **CKA_TOKEN**,
 1468 **CKA_PRIVATE**, and **CKA_MODIFIABLE** attributes can be changed in the process of copying an object,
 1469 however.)

1470 The **CKA_TOKEN** attribute identifies whether the object is a token object or a session object.

1471 When the **CKA_PRIVATE** attribute is CK_TRUE, a user may not access the object until the user has
 1472 been authenticated to the token.

1473 The value of the **CKA_MODIFIABLE** attribute determines whether or not an object is read-only.

1474 The **CKA_LABEL** attribute is intended to assist users in browsing.

1475 The value of the **CKA_COPYABLE** attribute determines whether or not an object can be copied. This
 1476 attribute can be used in conjunction with **CKA_MODIFIABLE** to prevent changes to the permitted usages
 1477 of keys and other objects.

1478 The value of the **CKA_DESTROYABLE** attribute determines whether the object can be destroyed using
 1479 C_DestroyObject.

1480 **4.4.1 The CKA_UNIQUE_ID attribute**

1481 Any time a new object is created, a value for **CKA_UNIQUE_ID** MUST be generated by the token and
 1482 stored with the object. The specific algorithm used to generate unique ID values for objects is token-
 1483 specific, but values generated MUST be unique across all objects visible to any particular session, and
 1484 SHOULD be unique across all objects created by the token. Reinitializing the token, such as by calling
 1485 C_InitToken, MAY cause reuse of **CKA_UNIQUE_ID** values.

1486 Any attempt to modify the **CKA_UNIQUE_ID** attribute of an existing object or to specify the value of the
 1487 **CKA_UNIQUE_ID** attribute in the template for an operation that creates one or more objects MUST fail.
 1488 Operations failing for this reason return the error code **CKR_ATTRIBUTE_READ_ONLY**.

1489

1490 **4.5 Data objects**

1491 **4.5.1 Definitions**

1492 This section defines the object class CKO_DATA for type CK_OBJECT_CLASS as used in the
1493 CKA_CLASS attribute of objects.

1494 **4.5.2 Overview**

1495 Data objects (object class **CKO_DATA**) hold information defined by an application. Other than providing
1496 access to it, Cryptoki does not attach any special meaning to a data object. The following table lists the
1497 attributes supported by data objects, in addition to the common attributes defined for this object class:

1498 *Table 18, Data Object Attributes*

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

1499 The **CKA_APPLICATION** attribute provides a means for applications to indicate ownership of the data
1500 objects they manage. Cryptoki does not provide a means of ensuring that only a particular application has
1501 access to a data object, however.

1502 The **CKA_OBJECT_ID** attribute provides an application independent and expandable way to indicate the
1503 type of the data object value. Cryptoki does not provide a means of insuring that the data object identifier
1504 matches the data value.

1505 The following is a sample template containing attributes for creating a data object:

```
1506 CK_OBJECT_CLASS class = CKO_DATA;
1507 CK_UTF8CHAR label[] = "A data object";
1508 CK_UTF8CHAR application[] = "An application";
1509 CK_BYTE data[] = "Sample data";
1510 CK_BBOOL true = CK_TRUE;
1511 CK_ATTRIBUTE template[] = {
1512     {CKA_CLASS, &class, sizeof(class)},
1513     {CKA_TOKEN, &true, sizeof(true)},
1514     {CKA_LABEL, label, sizeof(label)-1},
1515     {CKA_APPLICATION, application, sizeof(application)-1},
1516     {CKA_VALUE, data, sizeof(data)}}
1517 };
```

1518

1519 **4.6 Certificate objects**

1520 **4.6.1 Definitions**

1521 This section defines the object class CKO_CERTIFICATE for type CK_OBJECT_CLASS as used in the
1522 CKA_CLASS attribute of objects.

1523 **4.6.2 Overview**

1524 Certificate objects (object class **CKO_CERTIFICATE**) hold public-key or attribute certificates. Other than
1525 providing access to certificate objects, Cryptoki does not attach any special meaning to certificates. The
1526 following table defines the common certificate object attributes, in addition to the common attributes
1527 defined for this object class:

| 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) |

1529 Refer to Table 11 for footnotes

1530 Cryptoki does not enforce the relationship of the CKA_PUBLIC_KEY_INFO to the public key in the
1531 certificate, but does recommend that the key be extracted from the certificate to create this value.1532 The **CKA_CERTIFICATE_TYPE** attribute may not be modified after an object is created. This version of
1533 Cryptoki supports the following certificate types:

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

1537 The **CKA_TRUSTED** attribute cannot be set to CK_TRUE by an application. It MUST be set by a token
1538 initialization application or by the token's SO. Trusted certificates cannot be modified.1539 The **CKA_CERTIFICATE_CATEGORY** attribute is used to indicate if a stored certificate is a user
1540 certificate for which the corresponding private key is available on the token ("token user"), a CA certificate
1541 ("authority"), or another end-entity certificate ("other entity"). This attribute may not be modified after an
1542 object is created.1543 The **CKA_CERTIFICATE_CATEGORY** and **CKA_TRUSTED** attributes will together be used to map to
1544 the categorization of the certificates.1545 **CKA_CHECK_VALUE**: The value of this attribute is derived from the certificate by taking the first three
1546 bytes of the SHA-1 hash of the certificate object's CKA_VALUE attribute.1547 The **CKA_START_DATE** and **CKA_END_DATE** attributes are for reference only; Cryptoki does not
1548 attach any special meaning to them. When present, the application is responsible to set them to values
1549 that match the certificate's encoded "not before" and "not after" fields (if any).1550

4.6.3 X.509 public key certificate objects

1551 X.509 certificate objects (certificate type **CKC_X_509**) hold X.509 public key certificates. The following
1552 table defines the X.509 certificate object attributes, in addition to the common attributes defined for this
1553 object class:1554 *Table 20, X.509 Certificate Object Attributes*

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

¹MUST be specified when the object is created.

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

³MUST be non-empty if CKA_VALUE is empty.

⁴Can only be empty if CKA_URL is empty.

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

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

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

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

1573 The **CKA_URL** attribute enables the support for storage of the URL where the certificate can be found
 1574 instead of the certificate itself. Storage of a URL instead of the complete certificate is often used in mobile
 1575 environments.
 1576 The **CKA_HASH_OF SUBJECT PUBLIC KEY** and **CKA_HASH_OF_ISSUER_PUBLIC_KEY**
 1577 attributes are used to store the hashes of the public keys of the subject and the issuer. They are
 1578 particularly important when only the URL is available to be able to correlate a certificate with a private key
 1579 and when searching for the certificate of the issuer. The hash algorithm is defined by
 1580 **CKA_NAME_HASH_ALGORITHM**.
 1581 The **CKA_JAVA_MIDP_SECURITY_DOMAIN** attribute associates a certificate with a Java MIDP security
 1582 domain.

1583 The following is a sample template for creating an X.509 certificate object:

```

1584 CK_OBJECT_CLASS class = CKO_CERTIFICATE;
1585 CK_CERTIFICATE_TYPE certType = CKC_X_509;
1586 CK_UTF8CHAR label[] = "A certificate object";
1587 CK_BYTE subject[] = {...};
1588 CK_BYTE id[] = {123};
1589 CK_BYTE certificate[] = {...};
1590 CK_BBOOL true = CK_TRUE;
1591 CK_ATTRIBUTE template[] = {
1592   {CKA_CLASS, &class, sizeof(class)},
1593   {CKA_CERTIFICATE_TYPE, &certType, sizeof(certType)},
1594   {CKA_TOKEN, &true, sizeof(true)},
1595   {CKA_LABEL, label, sizeof(label)-1},
1596   {CKA_SUBJECT, subject, sizeof(subject)},
1597   {CKA_ID, id, sizeof(id)},
1598   {CKA_VALUE, certificate, sizeof(certificate)}}
1599 };

```

1600 4.6.4 WTLS public key certificate objects

1601 WTLS certificate objects (certificate type **CKC_WTLS**) hold WTLS public key certificates. The following
 1602 table defines the WTLS certificate object attributes, in addition to the common attributes defined for this
 1603 object class.

1604 *Table 21: WTLS Certificate Object Attributes*

| Attribute | Data type | Meaning |
|--|-------------------|---|
| CKA SUBJECT ¹ | Byte array | WTLS-encoding (Identifier type) of the certificate subject |
| CKA_ISSUER | Byte array | WTLS-encoding (Identifier type) of the certificate issuer (default empty) |
| CKA_VALUE ² | Byte array | WTLS-encoding of the certificate |
| CKA_URL ³ | RFC2279 string | If not empty this attribute gives the URL where the complete certificate can be obtained |
| CKA_HASH_OF SUBJECT_PU BLIC_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_PUB LIC_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_ALGORITH M | CK_MECHANISM_TYPE | Defines the mechanism used to calculate CKA_HASH_OF SUBJECT_PUBLIC |

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

1605 ¹MUST be specified when the object is created. Can only be empty if CKA_VALUE is empty.

1606 ²MUST be specified when the object is created. MUST be non-empty if CKA_URL is empty.

1607 ³MUST be non-empty if CKA_VALUE is empty.

1608 ⁴Can only be empty if CKA_URL is empty.

1609

1610 Only the **CKA_ISSUER** attribute may be modified after the object has been created.

1611 The encoding for the **CKA SUBJECT**, **CKA_ISSUER**, and **CKA_VALUE** attributes can be found in
1612 [WTLS].

1613 The **CKA_URL** attribute enables the support for storage of the URL where the certificate can be found
1614 instead of the certificate itself. Storage of a URL instead of the complete certificate is often used in mobile
1615 environments.

1616 The **CKA_HASH_OF_SUBJECT_PUBLIC_KEY** and **CKA_HASH_OF_ISSUER_PUBLIC_KEY**
1617 attributes are used to store the hashes of the public keys of the subject and the issuer. They are
1618 particularly important when only the URL is available to be able to correlate a certificate with a private key
1619 and when searching for the certificate of the issuer. The hash algorithm is defined by
1620 CKA_NAME_HASH_ALGORITHM.

1621 The following is a sample template for creating a WTLS certificate object:

```
1622 CK_OBJECT_CLASS class = CKO_CERTIFICATE;
1623 CK_CERTIFICATE_TYPE certType = CKC_WTLS;
1624 CK_UTF8CHAR label[] = "A certificate object";
1625 CK_BYTE subject[] = {...};
1626 CK_BYTE certificate[] = {...};
1627 CK_BBOOL true = CK_TRUE;
1628 CK_ATTRIBUTE template[] =
1629 {
1630     {CKA_CLASS, &class, sizeof(class)},
1631     {CKA_CERTIFICATE_TYPE, &certType, sizeof(certType)},
1632     {CKA_TOKEN, &true, sizeof(true)},
1633     {CKA_LABEL, label, sizeof(label)-1},
1634     {CKA_SUBJECT, subject, sizeof(subject)},
1635     {CKA_VALUE, certificate, sizeof(certificate)}
1636 };
```

1637 4.6.5 X.509 attribute certificate objects

1638 X.509 attribute certificate objects (certificate type **CKC_X_509_ATTR_CERT**) hold X.509 attribute
1639 certificates. The following table defines the X.509 attribute certificate object attributes, in addition to the
1640 common attributes defined for this object class:

1641 Table 22, X.509 Attribute Certificate Object Attributes

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

1642 ¹MUST be specified when the object is created1643 Only the **CKA_AC_ISSUER**, **CKA_SERIAL_NUMBER** and **CKA_ATTR_TYPES** attributes may be
1644 modified after the object is created.

1645 The following is a sample template for creating an X.509 attribute certificate object:

```

1646 CK_OBJECT_CLASS class = CKO_CERTIFICATE;
1647 CK_CERTIFICATE_TYPE certType = CKC_X_509_ATTR_CERT;
1648 CK_UTF8CHAR label[] = "An attribute certificate object";
1649 CK_BYTE owner[] = {...};
1650 CK_BYTE certificate[] = {...};
1651 CK_BBOOL true = CK_TRUE;
1652 CK_ATTRIBUTE template[] = {
1653     {CKA_CLASS, &class, sizeof(class)},
1654     {CKA_CERTIFICATE_TYPE, &certType, sizeof(certType)},
1655     {CKA_TOKEN, &true, sizeof(true)},
1656     {CKA_LABEL, label, sizeof(label)-1},
1657     {CKA_OWNER, owner, sizeof(owner)},
1658     {CKA_VALUE, certificate, sizeof(certificate)}}
1659 };

```

1660

4.7 Key objects

1661

4.7.1 Definitions

1662 There is no CKO_ definition for the base key object class, only for the key types derived from it.

1663 This section defines the object class CKO_PUBLIC_KEY, CKO_PRIVATE_KEY and
1664 CKO_SECRET_KEY for type CK_OBJECT_CLASS as used in the CKA_CLASS attribute of objects.1665

4.7.2 Overview

1666 Key objects hold encryption or authentication keys, which can be public keys, private keys, or secret
1667 keys. The following common footnotes apply to all the tables describing attributes of keys:1668 The following table defines the attributes common to public key, private key and secret key classes, in
1669 addition to the common attributes defined for this object class:

1670 Table 23, 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.e., 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.e., on the token) with a C_GenerateKey or C_GenerateKeyPair call • created with a C_CopyObject call as a copy of a key which had its CKA_LOCAL attribute set to CK_TRUE |
| CKA_KEY_GEN_MECHANISM ^{2,4,6} | CK_MECHANISM_TYPE | Identifier of the mechanism used to generate the key material. |
| CKA_ALLOWED_MECHANISMS | CK_MECHANISM_TYPE_PTR, pointer to a CK_MECHANISM_TYPE array | A list of mechanisms allowed to be used with this key. The number of mechanisms in the array is the <i>ulValueLen</i> component of the attribute divided by the size of CK_MECHANISM_TYPE. |

1671 Refer to Table 11 for footnotes

1672 The **CKA_ID** field is intended to distinguish among multiple keys. In the case of public and private keys,
 1673 this field assists in handling multiple keys held by the same subject; the key identifier for a public key and
 1674 its corresponding private key should be the same. The key identifier should also be the same as for the
 1675 corresponding certificate, if one exists. Cryptoki does not enforce these associations, however. (See
 1676 Section 4.6 for further commentary.)

1677 In the case of secret keys, the meaning of the **CKA_ID** attribute is up to the application.

1678 Note that the **CKA_START_DATE** and **CKA_END_DATE** attributes are for reference only; Cryptoki does
 1679 not attach any special meaning to them. In particular, it does not restrict usage of a key according to the
 1680 dates; doing this is up to the application.

1681 The **CKA_DERIVE** attribute has the value CK_TRUE if and only if it is possible to derive other keys from
 1682 the key.

1683 The **CKA_LOCAL** attribute has the value CK_TRUE if and only if the value of the key was originally
 1684 generated on the token by a **C_GenerateKey** or **C_GenerateKeyPair** call.

1685 The **CKA_KEY_GEN_MECHANISM** attribute identifies the key generation mechanism used to generate
 1686 the key material. It contains a valid value only if the **CKA_LOCAL** attribute has the value CK_TRUE. If
 1687 **CKA_LOCAL** has the value CK_FALSE, the value of the attribute is
 1688 CK_UNAVAILABLE_INFORMATION.

1689 4.8 Public key objects

1690 Public key objects (object class **CKO_PUBLIC_KEY**) hold public keys. The following table defines the
 1691 attributes common to all public keys, in addition to the common attributes defined for this object class:

1692 Table 24, 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_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>uValueLen</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) |

1693 - Refer to Table 11 for footnotes

1694 It is intended in the interests of interoperability that the subject name and key identifier for a public key will
1695 be the same as those for the corresponding certificate and private key. However, Cryptoki does not
1696 enforce this, and it is not required that the certificate and private key also be stored on the token.1697 To map between ISO/IEC 9594-8 (X.509) **keyUsage** flags for public keys and the PKCS #11 attributes for
1698 public keys, use the following table.

1699 Table 25, Mapping of X.509 key usage flags to Cryptoki attributes for public keys

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

1700 The value of the CKA_PUBLIC_KEY_INFO attribute is the DER encoded value of SubjectPublicKeyInfo:
 1701 SubjectPublicKeyInfo ::= SEQUENCE {
 1702 algorithm AlgorithmIdentifier,
 1703 subjectPublicKey BIT_STRING }

1704 The encodings for the subjectPublicKey field are specified in the description of the public key types in the
 1705 appropriate sections for the key types defined within this specification.

1706 4.9 Private key objects

1707 Private key objects (object class **CKO_PRIVATE_KEY**) hold private keys. The following table defines the
 1708 attributes common to all private keys, in addition to the common attributes defined for this object class:

1709 Table 26, Common Private Key Attributes

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

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

1710 - Refer to Table 11 for footnotes

1711 It is intended in the interests of interoperability that the subject name and key identifier for a private key
 1712 will be the same as those for the corresponding certificate and public key. However, this is not enforced
 1713 by Cryptoki, and it is not required that the certificate and public key also be stored on the token.

1714 If the **CKA_SENSITIVE** attribute is CK_TRUE, or if the **CKA_EXTRACTABLE** attribute is CK_FALSE,
 1715 then certain attributes of the private key cannot be revealed in plaintext outside the token. Which
 1716 attributes these are is specified for each type of private key in the attribute table in the section describing
 1717 that type of key.

1718 The **CKA_ALWAYS_AUTHENTICATE** attribute can be used to force re-authentication (i.e. force the user
 1719 to provide a PIN) for each use of a private key. "Use" in this case means a cryptographic operation such
 1720 as sign or decrypt. This attribute may only be set to CK_TRUE when **CKA_PRIVATE** is also CK_TRUE.

1721 Re-authentication occurs by calling **C_Login** with *userType* set to **CKU_CONTEXT_SPECIFIC**
 1722 immediately after a cryptographic operation using the key has been initiated (e.g. after **C_SignInit**). In
 1723 this call, the actual user type is implicitly given by the usage requirements of the active key. If **C_Login**
 1724 returns CKR_OK the user was successfully authenticated and this sets the active key in an authenticated
 1725 state that lasts until the cryptographic operation has successfully or unsuccessfully been completed (e.g.

1726 by **C_Sign**, **C_SignFinal**,..). A return value CKR_PIN_INCORRECT from **C_Login** means that the user
 1727 was denied permission to use the key and continuing the cryptographic operation will result in a behavior
 1728 as if **C_Login** had not been called. In both of these cases the session state will remain the same,
 1729 however repeated failed re-authentication attempts may cause the PIN to be locked. **C_Login** returns in
 1730 this case CKR_PIN_LOCKED and this also logs the user out from the token. Failing or omitting to re-
 1731 authenticate when CKA_ALWAYS_AUTHENTICATE is set to CK_TRUE will result in
 1732 CKR_USER_NOT_LOGGED_IN to be returned from calls using the key. **C_Login** will return
 1733 CKR_OPERATION_NOT_INITIALIZED, but the active cryptographic operation will not be affected, if an
 1734 attempt is made to re-authenticate when CKA_ALWAYS_AUTHENTICATE is set to CK_FALSE.
 1735 The **CKA_PUBLIC_KEY_INFO** attribute represents the public key associated with this private key. The
 1736 data it represents may either be stored as part of the private key data, or regenerated as needed from the
 1737 private key.
 1738 If this attribute is supplied as part of a template for **C_CreateObject**, **C_CopyObject** or
 1739 **C_SetAttributeValue** for a private key, the token MUST verify correspondence between the private key
 1740 data and the public key data as supplied in **CKA_PUBLIC_KEY_INFO**. This can be done either by
 1741 deriving a public key from the private key and comparing the values, or by doing a sign and verify
 1742 operation. If there is a mismatch, the command SHALL return **CKR_ATTRIBUTE_VALUE_INVALID**. A
 1743 token MAY choose not to support the **CKA_PUBLIC_KEY_INFO** attribute for commands which create
 1744 new private keys. If it does not support the attribute, the command SHALL return
 1745 **CKR_ATTRIBUTE_TYPE_INVALID**.
 1746 As a general guideline, private keys of any type SHOULD store sufficient information to retrieve the public
 1747 key information. In particular, the RSA private key description has been modified in PKCS #11 V2.40 to
 1748 add the **CKA_PUBLIC_EXPONENT** to the list of attributes required for an RSA private key. All other
 1749 private key types described in this specification contain sufficient information to recover the associated
 1750 public key.

1751 4.10 Secret key objects

1752 Secret key objects (object class **CKO_SECRET_KEY**) hold secret keys. The following table defines the
 1753 attributes common to all secret keys, in addition to the common attributes defined for this object class:
 1754

Table 27, Common Secret Key Attributes

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

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

1755 Refer to Table 11 for footnotes

1756 If the **CKA_SENSITIVE** attribute is CK_TRUE, or if the **CKA_EXTRACTABLE** attribute is CK_FALSE,
1757 then certain attributes of the secret key cannot be revealed in plaintext outside the token. Which
1758 attributes these are is specified for each type of secret key in the attribute table in the section describing
1759 that type of key.

1760 The key check value (KCV) attribute for symmetric key objects to be called **CKA_CHECK_VALUE**, of
1761 type byte array, length 3 bytes, operates like a fingerprint, or checksum of the key. They are intended to
1762 be used to cross-check symmetric keys against other systems where the same key is shared, and as a
1763 validity check after manual key entry or restore from backup. Refer to object definitions of specific key
1764 types for KCV algorithms.

1765 Properties:

- 1766 1. For two keys that are cryptographically identical the value of this attribute should be identical.
- 1767 2. CKA_CHECK_VALUE should not be usable to obtain any part of the key value.
- 1768 3. Non-uniqueness. Two different keys can have the same CKA_CHECK_VALUE. This is unlikely
(the probability can easily be calculated) but possible.

1770 The attribute is optional, but if supported, regardless of how the key object is created or derived, the value
1771 of the attribute is always supplied. It SHALL be supplied even if the encryption operation for the key is
1772 forbidden (i.e. when CKA_ENCRYPT is set to CK_FALSE).

1773 If a value is supplied in the application template (allowed but never necessary) then, if supported, it MUST
1774 match what the library calculates it to be or the library returns a CKR_ATTRIBUTE_VALUE_INVALID. If
1775 the library does not support the attribute then it should ignore it. Allowing the attribute in the template this
1776 way does no harm and allows the attribute to be treated like any other attribute for the purposes of key
1777 wrap and unwrap where the attributes are preserved also.

1778 The generation of the KCV may be prevented by the application supplying the attribute in the template as
1779 a no-value (0 length) entry. The application can query the value at any time like any other attribute using
1780 C_GetAttributeValue. C_SetAttributeValue may be used to destroy the attribute, by supplying no-value.

1781 Unless otherwise specified for the object definition, the value of this attribute is derived from the key
1782 object by taking the first three bytes of an encryption of a single block of null (0x00) bytes, using the
1783 default cipher and mode (e.g. ECB) associated with the key type of the secret key object.

1784 **4.11 Domain parameter objects**

1785 **4.11.1 Definitions**

1786 This section defines the object class CKO_DOMAIN_PARAMETERS for type CK_OBJECT_CLASS as
1787 used in the CKA_CLASS attribute of objects.

1788 **4.11.2 Overview**

1789 This object class was created to support the storage of certain algorithm's extended parameters. DSA
1790 and DH both use domain parameters in the key-pair generation step. In particular, some libraries support
1791 the generation of domain parameters (originally out of scope for PKCS11) so the object class was added.

1792 To use a domain parameter object you MUST extract the attributes into a template and supply them (still
1793 in the template) to the corresponding key-pair generation function.

1794 Domain parameter objects (object class **CKO_DOMAIN_PARAMETERS**) hold public domain parameters.

1795 The following table defines the attributes common to domain parameter objects in addition to the common
1796 attributes defined for this object class:

1797 *Table 28, Common Domain Parameter Attributes*

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

1798 ^ Refer to Table 11 for footnotes

1799 The **CKA_LOCAL** attribute has the value CK_TRUE if and only if the values of the domain parameters
1800 were originally generated on the token by a **C_GenerateKey** call.

1801 4.12 Mechanism objects

1802 4.12.1 Definitions

1803 This section defines the object class CKO_MECHANISM for type CK_OBJECT_CLASS as used in the
1804 CKA_CLASS attribute of objects.

1805 4.12.2 Overview

1806 Mechanism objects provide information about mechanisms supported by a device beyond that given by
1807 the **CK_MECHANISM_INFO** structure.

1808 When searching for objects using **C_FindObjectsInit** and **C_FindObjects**, mechanism objects are not
1809 returned unless the **CKA_CLASS** attribute in the template has the value **CKO_MECHANISM**. This
1810 protects applications written to previous versions of Cryptoki from finding objects that they do not
1811 understand.

1812 *Table 29, Common Mechanism Attributes*

| Attribute | Data Type | Meaning |
|--------------------|-------------------|------------------------------|
| CKA_MECHANISM_TYPE | CK_MECHANISM_TYPE | The type of mechanism object |

1813 The **CKA_MECHANISM_TYPE** attribute may not be set.

1814

1815 4.13 Profile objects

1816 4.13.1 Definitions

1817 This section defines the object class CKO_PROFILE for type CK_OBJECT_CLASS as used in the
1818 CKA_CLASS attribute of objects.

1819 4.13.2 Overview

1820 Profile objects (object class CKO_PROFILE) describe which PKCS #11 profiles the token implements.
1821 Profiles are defined in the OASIS PKCS #11 Cryptographic Token Interface Profiles document. A given
1822 token can contain more than one profile ID. The following table lists the attributes supported by profile
1823 objects, in addition to the common attributes defined for this object class:

1824 *Table 30, Profile Object Attributes*

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

1825 The **CKA_PROFILE_ID** attribute identifies a profile that the token supports.

1826 5 Functions

1827 Cryptoki's functions are organized into the following categories:

- 1828 • general-purpose functions (4 functions)
- 1829 • slot and token management functions (9 functions)
- 1830 • session management functions (8 functions)
- 1831 • object management functions (9 functions)
- 1832 • encryption functions (4 functions)
- 1833 • message-based encryption functions (5 functions)
- 1834 • decryption functions (4 functions)
- 1835 • message digesting functions (5 functions)
- 1836 • signing and MACing functions (6 functions)
- 1837 • functions for verifying signatures and MACs (6 functions)
- 1838 • dual-purpose cryptographic functions (4 functions)
- 1839 • key management functions (5 functions)
- 1840 • random number generation functions (2 functions)
- 1841 • parallel function management functions (2 functions)

1842
1843 In addition to these functions, Cryptoki can use application-supplied callback functions to notify an
1844 application of certain events, and can also use application-supplied functions to handle mutex objects for
1845 safe multi-threaded library access.

1846 The Cryptoki API functions are presented in the following table:

1847 *Table 31, Summary of Cryptoki Functions*

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

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

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

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

1848

1849 Execution of a Cryptoki function call is in general an all-or-nothing affair, *i.e.*, a function call accomplishes
1850 either its entire goal, or nothing at all.

- 1851 • If a Cryptoki function executes successfully, it returns the value CKR_OK.
- 1852 • If a Cryptoki function does not execute successfully, it returns some value other than CKR_OK, and
1853 the token is in the same state as it was in prior to the function call. If the function call was supposed
1854 to modify the contents of certain memory addresses on the host computer, these memory addresses
1855 may have been modified, despite the failure of the function.
- 1856 • In unusual (and extremely unpleasant!) circumstances, a function can fail with the return value
1857 CKR_GENERAL_ERROR. When this happens, the token and/or host computer may be in an
1858 inconsistent state, and the goals of the function may have been partially achieved.

1859 There are a small number of Cryptoki functions whose return values do not behave precisely as
1860 described above; these exceptions are documented individually with the description of the functions
1861 themselves.

1862 A Cryptoki library need not support every function in the Cryptoki API. However, even an unsupported
1863 function MUST have a “stub” in the library which simply returns the value
1864 CKR_FUNCTION_NOT_SUPPORTED. The function’s entry in the library’s **CK_FUNCTION_LIST**
1865 structure (as obtained by **C_GetFunctionList**) should point to this stub function (see Section 3.6).

1866 5.1 Function return values

1867 The Cryptoki interface possesses a large number of functions and return values. In Section 5.1, we
1868 enumerate the various possible return values for Cryptoki functions; most of the remainder of Section 5.1
1869 details the behavior of Cryptoki functions, including what values each of them may return.

1870 Because of the complexity of the Cryptoki specification, it is recommended that Cryptoki applications
1871 attempt to give some leeway when interpreting Cryptoki functions’ return values. We have attempted to
1872 specify the behavior of Cryptoki functions as completely as was feasible; nevertheless, there are
1873 presumably some gaps. For example, it is possible that a particular error code which might apply to a
1874 particular Cryptoki function is unfortunately not actually listed in the description of that function as a
1875 possible error code. It is conceivable that the developer of a Cryptoki library might nevertheless permit
1876 his/her implementation of that function to return that error code. It would clearly be somewhat ungraceful
1877 if a Cryptoki application using that library were to terminate by abruptly dumping core upon receiving that
1878 error code for that function. It would be far preferable for the application to examine the function’s return
1879 value, see that it indicates some sort of error (even if the application doesn’t know precisely *what* kind of
1880 error), and behave accordingly.

1881 See Section 5.1.8 for some specific details on how a developer might attempt to make an application that
1882 accommodates a range of behaviors from Cryptoki libraries.

1883 5.1.1 Universal Cryptoki function return values

1884 Any Cryptoki function can return any of the following values:

- 1885 • CKR_GENERAL_ERROR: Some horrible, unrecoverable error has occurred. In the worst case, it is
1886 possible that the function only partially succeeded, and that the computer and/or token is in an
1887 inconsistent state.
- 1888 • CKR_HOST_MEMORY: The computer that the Cryptoki library is running on has insufficient memory
1889 to perform the requested function.
- 1890 • CKR_FUNCTION_FAILED: The requested function could not be performed, but detailed information
1891 about why not is not available in this error return. If the failed function uses a session, it is possible
1892 that the **CK_SESSION_INFO** structure that can be obtained by calling **C_GetSessionInfo** will hold
1893 useful information about what happened in its *ulDeviceError* field. In any event, although the function
1894 call failed, the situation is not necessarily totally hopeless, as it is likely to be when
1895 CKR_GENERAL_ERROR is returned. Depending on what the root cause of the error actually was, it
1896 is possible that an attempt to make the exact same function call again would succeed.
- 1897 • CKR_OK: The function executed successfully. Technically, CKR_OK is not *quite* a “universal” return
1898 value; in particular, the legacy functions **C_GetFunctionStatus** and **C_CancelFunction** (see Section
1899 5.20) cannot return CKR_OK.

1900 The relative priorities of these errors are in the order listed above, e.g., if either of
1901 CKR_GENERAL_ERROR or CKR_HOST_MEMORY would be an appropriate error return, then
1902 CKR_GENERAL_ERROR should be returned.

1903 5.1.2 Cryptoki function return values for functions that use a session 1904 handle

1905 Any Cryptoki function that takes a session handle as one of its arguments (i.e., any Cryptoki function
1906 except for **C_Initialize**, **C_Finalize**, **C_GetInfo**, **C_GetFunctionList**, **C_GetSlotList**, **C_GetSlotInfo**,
1907 **C_GetTokenInfo**, **C_WaitForSlotEvent**, **C_GetMechanismList**, **C_GetMechanismInfo**, **C_InitToken**,
1908 **C_OpenSession**, and **C_CloseAllSessions**) can return the following values:

- 1909 • CKR_SESSION_HANDLE_INVALID: The specified session handle was invalid *at the time that the*
1910 *function was invoked*. Note that this can happen if the session’s token is removed before the function
1911 invocation, since removing a token closes all sessions with it.
- 1912 • CKR_DEVICE_REMOVED: The token was removed from its slot *during the execution of the function*.
- 1913 • CKR_SESSION_CLOSED: The session was closed *during the execution of the function*. Note that,
1914 as stated in [PKCS11-UG], the behavior of Cryptoki is *undefined* if multiple threads of an application
1915 attempt to access a common Cryptoki session simultaneously. Therefore, there is actually no
1916 guarantee that a function invocation could ever return the value CKR_SESSION_CLOSED. An
1917 example of multiple threads accessing a common session simultaneously is where one thread is
1918 using a session when another thread closes that same session.

1919 The relative priorities of these errors are in the order listed above, e.g., if either of
1920 CKR_SESSION_HANDLE_INVALID or CKR_DEVICE_REMOVED would be an appropriate error return,
1921 then CKR_SESSION_HANDLE_INVALID should be returned.

1922 In practice, it is often not crucial (or possible) for a Cryptoki library to be able to make a distinction
1923 between a token being removed *before* a function invocation and a token being removed *during a*
1924 function execution.

1925 5.1.3 Cryptoki function return values for functions that use a token

1926 Any Cryptoki function that uses a particular token (i.e., any Cryptoki function except for **C_Initialize**,
1927 **C_Finalize**, **C_GetInfo**, **C_GetFunctionList**, **C_GetSlotList**, **C_GetSlotInfo**, or **C_WaitForSlotEvent**)
1928 can return any of the following values:

- 1929 • CKR_DEVICE_MEMORY: The token does not have sufficient memory to perform the requested
1930 function.

- 1931 • CKR_DEVICE_ERROR: Some problem has occurred with the token and/or slot. This error code can
1932 be returned by more than just the functions mentioned above; in particular, it is possible for
1933 **C_GetSlotInfo** to return CKR_DEVICE_ERROR.
1934 • CKR_TOKEN_NOT_PRESENT: The token was not present in its slot *at the time that the function was*
1935 *invoked*.
1936 • CKR_DEVICE_REMOVED: The token was removed from its slot *during the execution of the function*.
1937 The relative priorities of these errors are in the order listed above, e.g., if either of
1938 CKR_DEVICE_MEMORY or CKR_DEVICE_ERROR would be an appropriate error return, then
1939 CKR_DEVICE_MEMORY should be returned.
1940 In practice, it is often not critical (or possible) for a Cryptoki library to be able to make a distinction
1941 between a token being removed *before* a function invocation and a token being removed *during* a
1942 function execution.

1943 5.1.4 Special return value for application-supplied callbacks

1944 There is a special-purpose return value which is not returned by any function in the actual Cryptoki API,
1945 but which may be returned by an application-supplied callback function. It is:

- 1946 • CKR_CANCEL: When a function executing in serial with an application decides to give the application
1947 a chance to do some work, it calls an application-supplied function with a CKN_SURRENDER
1948 callback (see Section 5.21). If the callback returns the value CKR_CANCEL, then the function aborts
1949 and returns CKR_FUNCTION_CANCELED.

1950 5.1.5 Special return values for mutex-handling functions

1951 There are two other special-purpose return values which are not returned by any actual Cryptoki
1952 functions. These values may be returned by application-supplied mutex-handling functions, and they may
1953 safely be ignored by application developers who are not using their own threading model. They are:

- 1954 • CKR_MUTEX_BAD: This error code can be returned by mutex-handling functions that are passed a
1955 bad mutex object as an argument. Unfortunately, it is possible for such a function not to recognize a
1956 bad mutex object. There is therefore no guarantee that such a function will successfully detect bad
1957 mutex objects and return this value.
1958 • CKR_MUTEX_NOT_LOCKED: This error code can be returned by mutex-unlocking functions. It
1959 indicates that the mutex supplied to the mutex-unlocking function was not locked.

1960 5.1.6 All other Cryptoki function return values

1961 Descriptions of the other Cryptoki function return values follow. Except as mentioned in the descriptions
1962 of particular error codes, there are in general no particular priorities among the errors listed below, i.e., if
1963 more than one error code might apply to an execution of a function, then the function may return any
1964 applicable error code.

- 1965 • CKR_ACTION_PROHIBITED: This value can only be returned by C_CopyObject,
1966 C_SetAttributeValue and C_DestroyObject. It denotes that the action may not be taken, either
1967 because of underlying policy restrictions on the token, or because the object has the relevant
1968 CKA_COPYABLE, CKA_MODIFIABLE or CKA_DESTROYABLE policy attribute set to CK_FALSE.
1969 • CKR_ARGUMENTS_BAD: This is a rather generic error code which indicates that the arguments
1970 supplied to the Cryptoki function were in some way not appropriate.
1971 • CKR_ATTRIBUTE_READ_ONLY: An attempt was made to set a value for an attribute which may not
1972 be set by the application, or which may not be modified by the application. See Section 4.1 for more
1973 information.
1974 • CKR_ATTRIBUTE_SENSITIVE: An attempt was made to obtain the value of an attribute of an object
1975 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 function cannot be executed because the Cryptoki library has not yet been initialized by a call to **C_Initialize**.
- CKR_CURVE_NOT_SUPPORTED: This curve is not supported by this token. Used with Elliptic Curve mechanisms.
- CKR_DATA_INVALID: The plaintext input data to a cryptographic operation is invalid. This return value has lower priority than CKR_DATA_LEN_RANGE.
- CKR_DATA_LEN_RANGE: The plaintext input data to a cryptographic operation has a bad length. Depending on the operation's mechanism, this could mean that the plaintext data is too short, too long, or is not a multiple of some particular block size. This return value has higher priority than CKR_DATA_INVALID.
- CKR_DOMAIN_PARAMS_INVALID: Invalid or unsupported domain parameters were supplied to the function. Which representation methods of domain parameters are supported by a given mechanism can vary from token to token.
- CKR_ENCRYPTED_DATA_INVALID: The encrypted input to a decryption operation has been determined to be invalid ciphertext. This return value has lower priority than CKR_ENCRYPTED_DATA_LEN_RANGE.
- CKR_ENCRYPTED_DATA_LEN_RANGE: The ciphertext input to a decryption operation has been determined to be invalid ciphertext solely on the basis of its length. Depending on the operation's mechanism, this could mean that the ciphertext is too short, too long, or is not a multiple of some particular block size. This return value has higher priority than CKR_ENCRYPTED_DATA_INVALID.
- CKR_EXCEEDED_MAX_ITERATIONS: An iterative algorithm (for key pair generation, domain parameter generation etc.) failed because we have exceeded the maximum number of iterations. This error code has precedence over CKR_FUNCTION_FAILED. Examples of iterative algorithms include DSA signature generation (retry if either r = 0 or s = 0) and generation of DSA primes p and q specified in FIPS 186-4.
- CKR_FIPS_SELF_TEST_FAILED: A FIPS 140-2 power-up self-test or conditional self-test failed. The token entered an error state. Future calls to cryptographic functions on the token will return CKR_GENERAL_ERROR. CKR_FIPS_SELF_TEST_FAILED has a higher precedence over CKR_GENERAL_ERROR. This error may be returned by C_Initialize, if a power-up self-test failed, by C_GenerateRandom or C_SeedRandom, if the continuous random number generator test failed, or by C_GenerateKeyPair, if the pair-wise consistency test failed.
- CKR_FUNCTION_CANCELED: The function was canceled in mid-execution. This happens to a cryptographic function if the function makes a **CKN_SURRENDER** application callback which returns CKR_CANCEL (see CKR_CANCEL). It also happens to a function that performs PIN entry through a protected path. The method used to cancel a protected path PIN entry operation is device dependent.
- CKR_FUNCTION_NOT_PARALLEL: There is currently no function executing in parallel in the specified session. This is a legacy error code which is only returned by the legacy functions **C_GetFunctionStatus** and **C_CancelFunction**.

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

- 2076 2. The library cannot function properly without being able to spawn new threads in the above
2077 fashion.
- 2078 • CKR_NO_EVENT: This value can only be returned by **C_WaitForSlotEvent**. It is returned when
2079 **C_WaitForSlotEvent** is called in non-blocking mode and there are no new slot events to return.
- 2080 • CKR_OBJECT_HANDLE_INVALID: The specified object handle is not valid. We reiterate here that 0
2081 is never a valid object handle.
- 2082 • CKR_OPERATION_ACTIVE: There is already an active operation (or combination of active
2083 operations) which prevents Cryptoki from activating the specified operation. For example, an active
2084 object-searching operation would prevent Cryptoki from activating an encryption operation with
2085 **C_EncryptInit**. Or, an active digesting operation and an active encryption operation would prevent
2086 Cryptoki from activating a signature operation. Or, on a token which doesn't support simultaneous
2087 dual cryptographic operations in a session (see the description of the
2088 **CKF_DUAL_CRYPTO_OPERATIONS** flag in the **CK_TOKEN_INFO** structure), an active signature
2089 operation would prevent Cryptoki from activating an encryption operation.
- 2090 • CKR_OPERATION_NOT_INITIALIZED: There is no active operation of an appropriate type in the
2091 specified session. For example, an application cannot call **C_Encrypt** in a session without having
2092 called **C_EncryptInit** first to activate an encryption operation.
- 2093 • CKR_PIN_EXPIRED: The specified PIN has expired, and the requested operation cannot be carried
2094 out unless **C_SetPIN** is called to change the PIN value. Whether or not the normal user's PIN on a
2095 token ever expires varies from token to token.
- 2096 • CKR_PIN_INCORRECT: The specified PIN is incorrect, *i.e.*, does not match the PIN stored on the
2097 token. More generally-- when authentication to the token involves something other than a PIN-- the
2098 attempt to authenticate the user has failed.
- 2099 • CKR_PIN_INVALID: The specified PIN has invalid characters in it. This return code only applies to
2100 functions which attempt to set a PIN.
- 2101 • CKR_PIN_LEN_RANGE: The specified PIN is too long or too short. This return code only applies to
2102 functions which attempt to set a PIN.
- 2103 • CKR_PIN_LOCKED: The specified PIN is "locked", and cannot be used. That is, because some
2104 particular number of failed authentication attempts has been reached, the token is unwilling to permit
2105 further attempts at authentication. Depending on the token, the specified PIN may or may not remain
2106 locked indefinitely.
- 2107 • CKR_PIN_TOO_WEAK: The specified PIN is too weak so that it could be easy to guess. If the PIN is
2108 too short, **CKR_PIN_LEN_RANGE** should be returned instead. This return code only applies to
2109 functions which attempt to set a PIN.
- 2110 • CKR_PUBLIC_KEY_INVALID: The public key fails a public key validation. For example, an EC
2111 public key fails the public key validation specified in Section 5.2.2 of ANSI X9.62. This error code may
2112 be returned by **C_CreateObject**, when the public key is created, or by **C_VerifyInit** or
2113 **C_VerifyRecoverInit**, when the public key is used. It may also be returned by **C_DeriveKey**, in
2114 preference to **CKR_MECHANISM_PARAM_INVALID**, if the other party's public key specified in the
2115 mechanism's parameters is invalid.
- 2116 • CKR_RANDOM_NO_RNG: This value can be returned by **C_SeedRandom** and
2117 **C_GenerateRandom**. It indicates that the specified token doesn't have a random number generator.
2118 This return value has higher priority than **CKR_RANDOM_SEED_NOT_SUPPORTED**.
- 2119 • CKR_RANDOM_SEED_NOT_SUPPORTED: This value can only be returned by **C_SeedRandom**.
2120 It indicates that the token's random number generator does not accept seeding from an application.
2121 This return value has lower priority than **CKR_RANDOM_NO_RNG**.
- 2122 • CKR_SAVED_STATE_INVALID: This value can only be returned by **C_SetOperationState**. It
2123 indicates that the supplied saved cryptographic operations state is invalid, and so it cannot be
2124 restored to the specified session.

- CKR_SESSION_COUNT: This value can only be returned by **C_OpenSession**. It indicates that the attempt to open a session failed, either because the token has too many sessions already open, or because the token has too many read/write sessions already open.
- CKR_SESSION_EXISTS: This value can only be returned by **C_InitToken**. It indicates that a session with the token is already open, and so the token cannot be initialized.
- CKR_SESSION_PARALLEL_NOT_SUPPORTED: The specified token does not support parallel sessions. This is a legacy error code—in Cryptoki Version 2.01 and up, no token supports parallel sessions. CKR_SESSION_PARALLEL_NOT_SUPPORTED can only be returned by **C_OpenSession**, and it is only returned when **C_OpenSession** is called in a particular [deprecated] way.
- CKR_SESSION_READ_ONLY: The specified session was unable to accomplish the desired action because it is a read-only session. This return value has lower priority than CKR_TOKEN_WRITE_PROTECTED.
- CKR_SESSION_READ_ONLY_EXISTS: A read-only session already exists, and so the SO cannot be logged in.
- CKR_SESSION_READ_WRITE_SO_EXISTS: A read/write SO session already exists, and so a read-only session cannot be opened.
- CKR_SIGNATURE_LEN_RANGE: The provided signature/MAC can be seen to be invalid solely on the basis of its length. This return value has higher priority than CKR_SIGNATURE_INVALID.
- CKR_SIGNATURE_INVALID: The provided signature/MAC is invalid. This return value has lower priority than CKR_SIGNATURE_LEN_RANGE.
- CKR_SLOT_ID_INVALID: The specified slot ID is not valid.
- CKR_STATE_UNSAVEABLE: The cryptographic operations state of the specified session cannot be saved for some reason (possibly the token is simply unable to save the current state). This return value has lower priority than CKR_OPERATION_NOT_INITIALIZED.
- CKR_TEMPLATE_INCOMPLETE: The template specified for creating an object is incomplete, and lacks some necessary attributes. See Section 4.1 for more information.
- CKR_TEMPLATE_INCONSISTENT: The template specified for creating an object has conflicting attributes. See Section 4.1 for more information.
- CKR_TOKEN_NOT_RECOGNIZED: The Cryptoki library and/or slot does not recognize the token in the slot.
- CKR_TOKEN_WRITE_PROTECTED: The requested action could not be performed because the token is write-protected. This return value has higher priority than CKR_SESSION_READ_ONLY.
- CKR_UNWRAPPING_KEY_HANDLE_INVALID: This value can only be returned by **C_UnwrapKey**. It indicates that the key handle specified to be used to unwrap another key is not valid.
- CKR_UNWRAPPING_KEY_SIZE_RANGE: This value can only be returned by **C_UnwrapKey**. It indicates that although the requested unwrapping operation could in principle be carried out, this Cryptoki library (or the token) is unable to actually do it because the supplied key's size is outside the range of key sizes that it can handle.
- CKR_UNWRAPPING_KEY_TYPE_INCONSISTENT: This value can only be returned by **C_UnwrapKey**. It indicates that the type of the key specified to unwrap another key is not consistent with the mechanism specified for unwrapping.
- CKR_USER_ALREADY_LOGGED_IN: This value can only be returned by **C_Login**. It indicates that the specified user cannot be logged into the session, because it is already logged into the session. For example, if an application has an open SO session, and it attempts to log the SO into it, it will receive this error code.
- CKR_USER_ANOTHER_ALREADY_LOGGED_IN: This value can only be returned by **C_Login**. It indicates that the specified user cannot be logged into the session, because another user is already

2173 logged into the session. For example, if an application has an open SO session, and it attempts to
2174 log the normal user into it, it will receive this error code.

- 2175 • CKR_USER_NOT_LOGGED_IN: The desired action cannot be performed because the appropriate
2176 user (or *an* appropriate user) is not logged in. One example is that a session cannot be logged out
2177 unless it is logged in. Another example is that a private object cannot be created on a token unless
2178 the session attempting to create it is logged in as the normal user. A final example is that
2179 cryptographic operations on certain tokens cannot be performed unless the normal user is logged in.
- 2180 • CKR_USER_PIN_NOT_INITIALIZED: This value can only be returned by **C_Login**. It indicates that
2181 the normal user's PIN has not yet been initialized with **C_InitPIN**.
- 2182 • CKR_USER_TOO_MANY_TYPES: An attempt was made to have more distinct users simultaneously
2183 logged into the token than the token and/or library permits. For example, if some application has an
2184 open SO session, and another application attempts to log the normal user into a session, the attempt
2185 may return this error. It is not required to, however. Only if the simultaneous distinct users cannot be
2186 supported does **C_Login** have to return this value. Note that this error code generalizes to true multi-
2187 user tokens.
- 2188 • CKR_USER_TYPE_INVALID: An invalid value was specified as a **CK_USER_TYPE**. Valid types are
2189 **CKU_SO**, **CKU_USER**, and **CKU_CONTEXT_SPECIFIC**.
- 2190 • CKR_WWRAPPED_KEY_INVALID: This value can only be returned by **C_UnwrapKey**. It indicates
2191 that the provided wrapped key is not valid. If a call is made to **C_UnwrapKey** to unwrap a particular
2192 type of key (*i.e.*, some particular key type is specified in the template provided to **C_UnwrapKey**),
2193 and the wrapped key provided to **C_UnwrapKey** is recognizably not a wrapped key of the proper
2194 type, then **C_UnwrapKey** should return CKR_WWRAPPED_KEY_INVALID. This return value has
2195 lower priority than CKR_WWRAPPED_KEY_LEN_RANGE.
- 2196 • CKR_WWRAPPED_KEY_LEN_RANGE: This value can only be returned by **C_UnwrapKey**. It
2197 indicates that the provided wrapped key can be seen to be invalid solely on the basis of its length.
2198 This return value has higher priority than CKR_WWRAPPED_KEY_INVALID.
- 2199 • CKR_WWRAPPING_KEY_HANDLE_INVALID: This value can only be returned by **C_WrapKey**. It
2200 indicates that the key handle specified to be used to wrap another key is not valid.
- 2201 • CKR_WWRAPPING_KEY_SIZE_RANGE: This value can only be returned by **C_WrapKey**. It indicates
2202 that although the requested wrapping operation could in principle be carried out, this Cryptoki library
2203 (or the token) is unable to actually do it because the supplied wrapping key's size is outside the range
2204 of key sizes that it can handle.
- 2205 • CKR_WWRAPPING_KEY_TYPE_INCONSISTENT: This value can only be returned by **C_WrapKey**. It
2206 indicates that the type of the key specified to wrap another key is not consistent with the mechanism
2207 specified for wrapping.
- 2208 • CKR_OPERATION_CANCEL_FAILED: This value can only be returned by **C_SessionCancel**. It
2209 means that one or more of the requested operations could not be cancelled for implementation or
2210 vendor-specific reasons.

2211 5.1.7 More on relative priorities of Cryptoki errors

2212 In general, when a Cryptoki call is made, error codes from Section 5.1.1 (other than CKR_OK) take
2213 precedence over error codes from Section 5.1.2, which take precedence over error codes from Section
2214 5.1.3, which take precedence over error codes from Section 5.1.6. One minor implication of this is that
2215 functions that use a session handle (*i.e.*, *most* functions!) never return the error code
2216 CKR_TOKEN_NOT_PRESENT (they return CKR_SESSION_HANDLE_INVALID instead). Other than
2217 these precedences, if more than one error code applies to the result of a Cryptoki call, any of the
2218 applicable error codes may be returned. Exceptions to this rule will be explicitly mentioned in the
2219 descriptions of functions.

2220 5.1.8 Error code “gotchas”

2221 Here is a short list of a few particular things about return values that Cryptoki developers might want to be
2222 aware of:

- 2223 1. As mentioned in Sections 5.1.2 and 5.1.3, a Cryptoki library may not be able to make a distinction
2224 between a token being removed *before* a function invocation and a token being removed *during* a
2225 function invocation.
- 2226 2. As mentioned in Section 5.1.2, an application should never count on getting a
2227 CKR_SESSION_CLOSED error.
- 2228 3. The difference between CKR_DATA_INVALID and CKR_DATA_LEN_RANGE can be somewhat
2229 subtle. Unless an application *needs* to be able to distinguish between these return values, it is best to
2230 always treat them equivalently.
- 2231 4. Similarly, the difference between CKR_ENCRYPTED_DATA_INVALID and
2232 CKR_ENCRYPTED_DATA_LEN_RANGE, and between CKR_WWRAPPED_KEY_INVALID and
2233 CKR_WWRAPPED_KEY_LEN_RANGE, can be subtle, and it may be best to treat these return values
2234 equivalently.
- 2235 5. Even with the guidance of Section 4.1, it can be difficult for a Cryptoki library developer to know which
2236 of CKR_ATTRIBUTE_VALUE_INVALID, CKR_TEMPLATE_INCOMPLETE, or
2237 CKR_TEMPLATE_INCONSISTENT to return. When possible, it is recommended that application
2238 developers be generous in their interpretations of these error codes.

2239 5.2 Conventions for functions returning output in a variable-length 2240 buffer

2241 A number of the functions defined in Cryptoki return output produced by some cryptographic mechanism.
2242 The amount of output returned by these functions is returned in a variable-length application-supplied
2243 buffer. An example of a function of this sort is **C_Encrypt**, which takes some plaintext as an argument,
2244 and outputs a buffer full of ciphertext.

2245 These functions have some common calling conventions, which we describe here. Two of the arguments
2246 to the function are a pointer to the output buffer (say *pBuf*) and a pointer to a location which will hold the
2247 length of the output produced (say *pulBufLen*). There are two ways for an application to call such a
2248 function:

- 2249 1. If *pBuf* is NULL_PTR, then all that the function does is return (in **pulBufLen*) a number of bytes which
2250 would suffice to hold the cryptographic output produced from the input to the function. This number
2251 may somewhat exceed the precise number of bytes needed, but should not exceed it by a large
2252 amount. CKR_OK is returned by the function.
- 2253 2. If *pBuf* is not NULL_PTR, then **pulBufLen* MUST contain the size in bytes of the buffer pointed to by
2254 *pBuf*. If that buffer is large enough to hold the cryptographic output produced from the input to the
2255 function, then that cryptographic output is placed there, and CKR_OK is returned by the function and
2256 **pulBufLen* is set to the exact number of bytes returned. If the buffer is not large enough, then
2257 CKR_BUFFER_TOO_SMALL is returned and **pulBufLen* is set to at least the number of bytes
2258 needed to hold the cryptographic output produced from the input to the function.

2259 NOTE: This is a change from previous specs. The problem is that in some decrypt cases, the token
2260 doesn't know how big a buffer is needed until the decrypt completes. The act of doing decrypt can mess
2261 up the internal encryption state. Many tokens already implement this relaxed behavior, tokens which
2262 implement the more precise behavior are still compliant. The one corner case is applications using a
2263 token that knows exactly how big the decryption is (through some out of band means), could get
2264 CKR_BUFFER_TOO_SMALL returned when it supplied a buffer exactly big enough to hold the decrypted
2265 value when it may previously have succeeded.

2266 All functions which use the above convention will explicitly say so.

2267 Cryptographic functions which return output in a variable-length buffer should always return as much
2268 output as can be computed from what has been passed in to them thus far. As an example, consider a
2269 session which is performing a multiple-part decryption operation with DES in cipher-block chaining mode

2270 with PKCS padding. Suppose that, initially, 8 bytes of ciphertext are passed to the **C_DecryptUpdate**
2271 function. The block size of DES is 8 bytes, but the PKCS padding makes it unclear at this stage whether
2272 the ciphertext was produced from encrypting a 0-byte string, or from encrypting some string of length at
2273 least 8 bytes. Hence the call to **C_DecryptUpdate** should return 0 bytes of plaintext. If a single
2274 additional byte of ciphertext is supplied by a subsequent call to **C_DecryptUpdate**, then that call should
2275 return 8 bytes of plaintext (one full DES block).

2276 5.3 Disclaimer concerning sample code

2277 For the remainder of this section, we enumerate the various functions defined in Cryptoki. Most functions
2278 will be shown in use in at least one sample code snippet. For the sake of brevity, sample code will
2279 frequently be somewhat incomplete. In particular, sample code will generally ignore possible error
2280 returns from C library functions, and also will not deal with Cryptoki error returns in a realistic fashion.

2281 5.4 General-purpose functions

2282 Cryptoki provides the following general-purpose functions:

2283 5.4.1 C_Initialize

```
2284 CK_DECLARE_FUNCTION(CK_RV, C_Initialize) {  
2285     CK_VOID_PTR pInitArgs  
2286 };
```

2287 **C_Initialize** initializes the Cryptoki library. *pInitArgs* either has the value `NULL_PTR` or points to a
2288 **CK_C_INITIALIZE_ARGS** structure containing information on how the library should deal with multi-
2289 threaded access. If an application will not be accessing Cryptoki through multiple threads simultaneously,
2290 it can generally supply the value `NULL_PTR` to **C_Initialize** (the consequences of supplying this value will
2291 be explained below).

2292 If *pInitArgs* is non-`NULL_PTR`, **C_Initialize** should cast it to a **CK_C_INITIALIZE_ARGS_PTR** and then
2293 dereference the resulting pointer to obtain the **CK_C_INITIALIZE_ARGS** fields *CreateMutex*,
2294 *DestroyMutex*, *LockMutex*, *UnlockMutex*, *flags*, and *pReserved*. For this version of Cryptoki, the value of
2295 *pReserved* thereby obtained MUST be `NULL_PTR`; if it's not, then **C_Initialize** should return with the
2296 value `CKR_ARGUMENTS_BAD`.

2297 If the **CKF_LIBRARY_CANT_CREATE_OS_THREADS** flag in the *flags* field is set, that indicates that
2298 application threads which are executing calls to the Cryptoki library are not permitted to use the native
2299 operation system calls to spawn off new threads. In other words, the library's code may not create its
2300 own threads. If the library is unable to function properly under this restriction, **C_Initialize** should return
2301 with the value `CKR_NEED_TO_CREATE_THREADS`.

2302 A call to **C_Initialize** specifies one of four different ways to support multi-threaded access via the value of
2303 the **CKF_OS_LOCKING_OK** flag in the *flags* field and the values of the *CreateMutex*, *DestroyMutex*,
2304 *LockMutex*, and *UnlockMutex* function pointer fields:

- 2305 1. If the flag *isn't* set, and the function pointer fields *aren't* supplied (*i.e.*, they all have the value
2306 `NULL_PTR`), that means that the application *won't* be accessing the Cryptoki library from multiple
2307 threads simultaneously.
- 2308 2. If the flag *is* set, and the function pointer fields *aren't* supplied (*i.e.*, they all have the value
2309 `NULL_PTR`), that means that the application *will* be performing multi-threaded Cryptoki access, and
2310 the library needs to use the native operating system primitives to ensure safe multi-threaded access.
2311 If the library is unable to do this, **C_Initialize** should return with the value `CKR_CANT_LOCK`.
- 2312 3. If the flag *isn't* set, and the function pointer fields *are* supplied (*i.e.*, they all have non-`NULL_PTR`
2313 values), that means that the application *will* be performing multi-threaded Cryptoki access, and the
2314 library needs to use the supplied function pointers for mutex-handling to ensure safe multi-threaded
2315 access. If the library is unable to do this, **C_Initialize** should return with the value
2316 `CKR_CANT_LOCK`.

2317 4. If the flag *is* set, and the function pointer fields are supplied (*i.e.*, they all have non-NUL_PTR
2318 values), that means that the application *will* be performing multi-threaded Cryptoki access, and the
2319 library needs to use either the native operating system primitives or the supplied function pointers for
2320 mutex-handling to ensure safe multi-threaded access. If the library is unable to do this, **C_Initialize**
2321 should return with the value CKR_CANT_LOCK.
2322 If some, but not all, of the supplied function pointers to **C_Initialize** are non-NUL_PTR, then **C_Initialize**
2323 should return with the value CKR_ARGUMENTS_BAD.
2324 A call to **C_Initialize** with *pInitArgs* set to NULL_PTR is treated like a call to **C_Initialize** with *pInitArgs*
2325 pointing to a **CK_C_INITIALIZE_ARGS** which has the *CreateMutex*, *DestroyMutex*, *LockMutex*,
2326 *UnlockMutex*, and *pReserved* fields set to NULL_PTR, and has the *flags* field set to 0.
2327 **C_Initialize** should be the first Cryptoki call made by an application, except for calls to
2328 **C_GetFunctionList**, **C_GetInterfaceList**, or **C_GetInterface**. What this function actually does is
2329 implementation-dependent; typically, it might cause Cryptoki to initialize its internal memory buffers, or
2330 any other resources it requires.
2331 If several applications are using Cryptoki, each one should call **C_Initialize**. Every call to **C_Initialize**
2332 should (eventually) be succeeded by a single call to **C_Finalize**. See [PKCS11-UG] for further details.
2333 Return values: CKR_ARGUMENTS_BAD, CKR_CANT_LOCK,
2334 CKR_CRYPTOKI_ALREADY_INITIALIZED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
2335 CKR_HOST_MEMORY, CKR_NEED_TO_CREATE_THREADS, CKR_OK.
2336 Example: see **C_GetInfo**.

2337 5.4.2 C_Finalize

```
2338 CK_DECLARE_FUNCTION(CK_RV, C_Finalize) (
2339     CK_VOID_PTR pReserved
2340 );
```

2341 **C_Finalize** is called to indicate that an application is finished with the Cryptoki library. It should be the
2342 last Cryptoki call made by an application. The *pReserved* parameter is reserved for future versions; for
2343 this version, it should be set to NULL_PTR (if **C_Finalize** is called with a non-NUL_PTR value for
2344 *pReserved*, it should return the value CKR_ARGUMENTS_BAD).

2345 If several applications are using Cryptoki, each one should call **C_Finalize**. Each application's call to
2346 **C_Finalize** should be preceded by a single call to **C_Initialize**; in between the two calls, an application
2347 can make calls to other Cryptoki functions. See [PKCS11-UG] for further details.

2348 *Despite the fact that the parameters supplied to **C_Initialize** can in general allow for safe multi-threaded*
2349 *access to a Cryptoki library, the behavior of **C_Finalize** is nevertheless undefined if it is called by an*
2350 *application while other threads of the application are making Cryptoki calls. The exception to this*
2351 *exceptional behavior of **C_Finalize** occurs when a thread calls **C_Finalize** while another of the*
2352 *application's threads is blocking on Cryptoki's **C_WaitForSlotEvent** function. When this happens, the*
2353 *blocked thread becomes unblocked and returns the value CKR_CRYPTOKI_NOT_INITIALIZED. See*
2354 ***C_WaitForSlotEvent** for more information.*

2355 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
2356 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK.

2357 Example: see **C_GetInfo**.

2358 5.4.3 C_GetInfo

```
2359 CK_DECLARE_FUNCTION(CK_RV, C_GetInfo) (
2360     CK_INFO_PTR pInfo
2361 );
```

2362 **C_GetInfo** returns general information about Cryptoki. *pInfo* points to the location that receives the
2363 information.

2364 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
2365 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK.

2366 Example:

```
2367 CK_INFO info;
2368 CK_RV rv;
2369 CK_C_INITIALIZE_ARGS InitArgs;
2370
2371 InitArgs.CreateMutex = &MyCreateMutex;
2372 InitArgs.DestroyMutex = &MyDestroyMutex;
2373 InitArgs.LockMutex = &MyLockMutex;
2374 InitArgs.UnlockMutex = &MyUnlockMutex;
2375 InitArgs.flags = CKF_OS_LOCKING_OK;
2376 InitArgs.pReserved = NULL_PTR;
2377
2378 rv = C_Initialize((CK_VOID_PTR)&InitArgs);
2379 assert(rv == CKR_OK);
2380
2381 rv = C_GetInfo(&info);
2382 assert(rv == CKR_OK);
2383 if(info.cryptokiVersion.major == 2) {
2384     /* Do lots of interesting cryptographic things with the token */
2385     .
2386     .
2387 }
2388
2389 rv = C_Finalize(NULL_PTR);
2390 assert(rv == CKR_OK);
```

2391 **5.4.4 C_GetFunctionList**

```
2392 CK_DECLARE_FUNCTION(CK_RV, C_GetFunctionList) (
2393     CK_FUNCTION_LIST_PTR_PTP pFunctionList
2394 );
```

2395 **C_GetFunctionList** obtains a pointer to the Cryptoki library's list of function pointers. *ppFunctionList*
2396 points to a value which will receive a pointer to the library's **CK_FUNCTION_LIST** structure, which in turn
2397 contains function pointers for all the Cryptoki API routines in the library. *The pointer thus obtained may*
2398 *point into memory which is owned by the Cryptoki library, and which may or may not be writable.*
2399 Whether or not this is the case, no attempt should be made to write to this memory.

2400 **C_GetFunctionList**, **C_GetInterfaceList**, and **C_GetInterface** are the only Cryptoki functions which an
2401 application may call before calling **C_Initialize**. It is provided to make it easier and faster for applications
2402 to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously.

2403 Return values: CKR_ARGUMENTS_BAD, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
2404 CKR_HOST_MEMORY, CKR_OK.

2405 Example:

```
2406 CK_FUNCTION_LIST_PTR pFunctionList;
```

```

2407 CK_C_Initialize pC_Initialize;
2408 CK_RV rv;
2409
2410 /* It's OK to call C_GetFunctionList before calling C_Initialize */
2411 rv = C_GetFunctionList(&pFunctionList);
2412 assert(rv == CKR_OK);
2413 pC_Initialize = pFunctionList -> C_Initialize;
2414
2415 /* Call the C_Initialize function in the library */
2416 rv = (*pC_Initialize)(NULL_PTR);

```

2417 5.4.5 C_GetInterfaceList

```

2418 CK_DECLARE_FUNCTION(CK_RV, C_GetInterfaceList) (
2419     CK_INTERFACE_PTR      pInterfaceList,
2420     CK ULONG_PTR          pulCount
2421 );

```

2422 **C_GetInterfaceList** is used to obtain a list of interfaces supported by a Cryptoki library. *pulCount* points
 2423 to the location that receives the number of interfaces.

2424 There are two ways for an application to call **C_GetInterfaceList**:

- 2425 1. If *pInterfaceList* is NULL_PTR, then all that **C_GetInterfaceList** does is return (in **pulCount*) the
 2426 number of interfaces, without actually returning a list of interfaces. The contents of **pulCount* on
 2427 entry to **C_GetInterfaceList** has no meaning in this case, and the call returns the value CKR_OK.
- 2428 2. If *pInterfaceList* is not NULL_PTR, then **pulCount* MUST contain the size (in terms of
 2429 **CK_INTERFACE** elements) of the buffer pointed to by *pInterfaceList*. If that buffer is large enough to
 2430 hold the list of interfaces, then the list is returned in it, and CKR_OK is returned. If not, then the call
 2431 to **C_GetInterfaceList** returns the value CKR_BUFFER_TOO_SMALL. In either case, the value
 2432 **pulCount* is set to hold the number of interfaces.

2433 Because **C_GetInterfaceList** does not allocate any space of its own, an application will often call
 2434 **C_GetInterfaceList** twice. However, this behavior is by no means required.

2435 **C_GetInterfaceList** obtains (in **pFunctionList* of each interface) a pointer to the Cryptoki library's list of
 2436 function pointers. *The pointer thus obtained may point into memory which is owned by the Cryptoki*
 2437 *library, and which may or may not be writable*. Whether or not this is the case, no attempt should be
 2438 made to write to this memory. The same caveat applies to the interface names returned.

2439

2440 **C_GetFunctionList**, **C_GetInterfaceList**, and **C_GetInterface** are the only Cryptoki functions which an
 2441 application may call before calling **C_Initialize**. It is provided to make it easier and faster for applications
 2442 to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously.

2443 Return values: CKR_BUFFER_TOO_SMALL, CKR_ARGUMENTS_BAD, CKR_FUNCTION_FAILED,
 2444 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK.

2445 Example:

```

2446 CK ULONG ulCount=0;
2447 CK_INTERFACE_PTR interfaceList=NULL;
2448 CK_RV rv;
2449 int I;
2450
2451 /* get number of interfaces */

```

```

2452 rv = C_GetInterfaceList(NULL, &ulCount);
2453 if (rv == CKR_OK) {
2454     /* get copy of interfaces */
2455     interfaceList = (CK_INTERFACE_PTR)malloc(ulCount*sizeof(CK_INTERFACE));
2456     rv = C_GetInterfaceList(interfaceList,&ulCount);
2457     for(i=0;i<ulCount;i++) {
2458         printf("interface %s version %d.%d funcs %p flags 0x%lu\n",
2459             interfaceList[i].pInterfaceName,
2460             ((CK_VERSION *)interfaceList[i].pFunctionList)->major,
2461             ((CK_VERSION *)interfaceList[i].pFunctionList)->minor,
2462             interfaceList[i].pFunctionList,
2463             interfaceList[i].flags);
2464     }
2465 }
2466

```

2467 5.4.6 C_GetInterface

```

2468 CK_DECLARE_FUNCTION(CK_RV,C_GetInterface) (
2469     CK_UTF8CHAR_PTR          pInterfaceName,
2470     CK_VERSION_PTR           pVersion,
2471     CK_INTERFACE_PTR_PTR    ppInterface,
2472     CK_FLAGS                 flags
2473 );

```

2474 **C_GetInterface** is used to obtain an interface supported by a Cryptoki library. *pInterfaceName* specifies
 2475 the name of the interface, *pVersion* specifies the interface version, *ppInterface* points to the location that
 2476 receives the interface, *flags* specifies the required interface flags.

2477 There are multiple ways for an application to specify a particular interface when calling **C_GetInterface**:

- 2478 1. If *pInterfaceName* is not NULL_PTR, the name of the interface returned must match. If *pInterfaceName* is NULL_PTR, the cryptoki library can return a default interface of its choice
- 2479 2. If *pVersion* is not NULL_PTR, the version of the interface returned must match. If *pVersion* is NULL_PTR, the cryptoki library can return an interface of any version
- 2480 3. If *flags* is non-zero, the interface returned must match all of the supplied flag values (but may include additional flags not specified). If *flags* is 0, the cryptoki library can return an interface with any flags

2482 **C_GetInterface** obtains (in **pFunctionList* of each interface) a pointer to the Cryptoki library's list of
 2483 function pointers. *The pointer thus obtained may point into memory which is owned by the Cryptoki
 2484 library, and which may or may not be writable*. Whether or not this is the case, no attempt should be
 2485 made to write to this memory. The same caveat applies to the interface names returned.

2486 **C_GetFunctionList**, **C_GetInterfaceList**, and **C_GetInterface** are the only Cryptoki functions which an
 2487 application may call before calling **C_Initialize**. It is provided to make it easier and faster for applications
 2488 to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously.

2489 Return values: CKR_BUFFER_TOO_SMALL, CKR_ARGUMENTS_BAD, CKR_FUNCTION_FAILED,
 2490 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK.

2491 Example:

```

2494 CK_INTERFACE_PTR interface;
2495 CK_RV rv;

```

```

2496 CK_VERSION version;
2497 CK_FLAGS flags=CKF_INTERFACE_FORK_SAFE;
2498
2499 /* get default interface */
2500 rv = C_GetInterface(NULL,NULL,&interface,flags);
2501 if (rv == CKR_OK) {
2502     printf("interface %s version %d.%d funcs %p flags 0x%lu\n",
2503            interface->pInterfaceName,
2504            ((CK_VERSION *)interface->pFunctionList)->major,
2505            ((CK_VERSION *)interface->pFunctionList)->minor,
2506            interface->pFunctionList,
2507            interface->flags);
2508 }
2509
2510 /* get default standard interface */
2511 rv = C_GetInterface((CK_UTF8CHAR_PTR)"PKCS 11",NULL,&interface,flags);
2512 if (rv == CKR_OK) {
2513     printf("interface %s version %d.%d funcs %p flags 0x%lu\n",
2514            interface->pInterfaceName,
2515            ((CK_VERSION *)interface->pFunctionList)->major,
2516            ((CK_VERSION *)interface->pFunctionList)->minor,
2517            interface->pFunctionList,
2518            interface->flags);
2519 }
2520
2521 /* get specific standard version interface */
2522 version.major=3;
2523 version.minor=0;
2524 rv = C_GetInterface((CK_UTF8CHAR_PTR)"PKCS 11",&version,&interface,flags);
2525 if (rv == CKR_OK) {
2526     CK_FUNCTION_LIST_3_0_PTR pkcs11=interface->pFunctionList;
2527
2528     /* ... use the new functions */
2529     pkcs11->C_LoginUser(hSession,userType,pPin,ulPinLen,
2530                           pUsername,ulUsernameLen);
2531 }
2532
2533 /* get specific vendor version interface */
2534 version.major=1;
2535 version.minor=0;
2536 rv = C_GetInterface((CK_UTF8CHAR_PTR)
2537                      "Vendor VendorName",&version,&interface,flags);
2538 if (rv == CKR_OK) {

```

```

2539     CK_FUNCTION_LIST_VENDOR_1_0_PTR pkcs11=interface->pFunctionList;
2540
2541     /* ... use vendor specific functions */
2542     pkcs11->C_VendorFunction1(param1,param2,param3);
2543 }
2544

```

2545 5.5 Slot and token management functions

2546 Cryptoki provides the following functions for slot and token management:

2547 5.5.1 C_GetSlotList

```

2548 CK_DECLARE_FUNCTION(CK_RV, C_GetSlotList) (
2549     CK_BBOOL tokenPresent,
2550     CK_SLOT_ID_PTR pSlotList,
2551     CK ULONG_PTR pulCount
2552 );

```

2553 **C_GetSlotList** is used to obtain a list of slots in the system. *tokenPresent* indicates whether the list
2554 obtained includes only those slots with a token present (CK_TRUE), or all slots (CK_FALSE); *pulCount*
2555 points to the location that receives the number of slots.

2556 There are two ways for an application to call **C_GetSlotList**:

- 2557 1. If *pSlotList* is NULL_PTR, then all that **C_GetSlotList** does is return (in **pulCount*) the number of
2558 slots, without actually returning a list of slots. The contents of the buffer pointed to by *pulCount* on
2559 entry to **C_GetSlotList** has no meaning in this case, and the call returns the value CKR_OK.
- 2560 2. If *pSlotList* is not NULL_PTR, then **pulCount* MUST contain the size (in terms of CK_SLOT_ID
2561 elements) of the buffer pointed to by *pSlotList*. If that buffer is large enough to hold the list of slots,
2562 then the list is returned in it, and CKR_OK is returned. If not, then the call to **C_GetSlotList** returns
2563 the value CKR_BUFFER_TOO_SMALL. In either case, the value **pulCount* is set to hold the number
2564 of slots.

2565 Because **C_GetSlotList** does not allocate any space of its own, an application will often call
2566 **C_GetSlotList** twice (or sometimes even more times—if an application is trying to get a list of all slots
2567 with a token present, then the number of such slots can (unfortunately) change between when the
2568 application asks for how many such slots there are and when the application asks for the slots
2569 themselves). However, multiple calls to **C_GetSlotList** are by no means required.

2570 All slots which **C_GetSlotList** reports MUST be able to be queried as valid slots by **C_GetSlotInfo**.
2571 Furthermore, the set of slots accessible through a Cryptoki library is checked at the time that
2572 **C_GetSlotList**, for list length prediction (NULL *pSlotList* argument) is called. If an application calls
2573 **C_GetSlotList** with a non-NULL *pSlotList*, and then the user adds or removes a hardware device, the
2574 changed slot list will only be visible and effective if **C_GetSlotList** is called again with NULL. Even if **C**
2575 **GetSlotList** is successfully called this way, it may or may not be the case that the changed slot list will be
2576 successfully recognized depending on the library implementation. On some platforms, or earlier PKCS11
2577 compliant libraries, it may be necessary to successfully call **C_Initialize** or to restart the entire system.

2578

2579 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
2580 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
2581 CKR_HOST_MEMORY, CKR_OK.

2582 Example:

```

2583 CK ULONG ulSlotCount, ulSlotWithTokenCount;
2584 CK_SLOT_ID_PTR pSlotList, pSlotWithTokenList;

```

```

2585 CK_RV rv;
2586
2587 /* Get list of all slots */
2588 rv = C_GetSlotList(CK_FALSE, NULL_PTR, &ulSlotCount);
2589 if (rv == CKR_OK) {
2590     pSlotList =
2591         (CK_SLOT_ID_PTR) malloc(ulSlotCount*sizeof(CK_SLOT_ID));
2592     rv = C_GetSlotList(CK_FALSE, pSlotList, &ulSlotCount);
2593     if (rv == CKR_OK) {
2594         /* Now use that list of all slots */
2595         .
2596         .
2597     }
2598
2599     free(pSlotList);
2600 }
2601
2602 /* Get list of all slots with a token present */
2603 pSlotWithTokenList = (CK_SLOT_ID_PTR) malloc(0);
2604 ulSlotWithTokenCount = 0;
2605 while (1) {
2606     rv = C_GetSlotList(
2607         CK_TRUE, pSlotWithTokenList, &ulSlotWithTokenCount);
2608     if (rv != CKR_BUFFER_TOO_SMALL)
2609         break;
2610     pSlotWithTokenList = realloc(
2611         pSlotWithTokenList,
2612         ulSlotWithTokenList*sizeof(CK_SLOT_ID));
2613 }
2614
2615 if (rv == CKR_OK) {
2616     /* Now use that list of all slots with a token present */
2617     .
2618     .
2619 }
2620
2621 free(pSlotWithTokenList);

```

2622 5.5.2 C_GetSlotInfo

```

2623 CK_DECLARE_FUNCTION(CK_RV, C_GetSlotInfo) (
2624     CK_SLOT_ID slotID,
2625     CK_SLOT_INFO_PTR pInfo
2626 );

```

2627 **C_GetSlotInfo** obtains information about a particular slot in the system. *slotID* is the ID of the slot; *pInfo*
2628 points to the location that receives the slot information.
2629 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
2630 CKR_DEVICE_ERROR, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
2631 CKR_OK, CKR_SLOT_ID_INVALID.
2632 Example: see **C_GetTokenInfo**.

2633 5.5.3 C_GetTokenInfo

```
2634 CK_DECLARE_FUNCTION(CK_RV, C_GetTokenInfo) (
2635     CK_SLOT_ID slotID,
2636     CK_TOKEN_INFO_PTR pInfo
2637 );
```

2638 **C_GetTokenInfo** obtains information about a particular token in the system. *slotID* is the ID of the
2639 token's slot; *pInfo* points to the location that receives the token information.

2640 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
2641 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
2642 CKR_HOST_MEMORY, CKR_OK, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT,
2643 CKR_TOKEN_NOT_RECOGNIZED, CKR_ARGUMENTS_BAD.

2644 Example:

```
2645 CK ULONG ulCount;
2646 CK_SLOT_ID_PTR pSlotList;
2647 CK_SLOT_INFO slotInfo;
2648 CK_TOKEN_INFO tokenInfo;
2649 CK_RV rv;
2650
2651 rv = C_GetSlotList(CK_FALSE, NULL_PTR, &ulCount);
2652 if ((rv == CKR_OK) && (ulCount > 0)) {
2653     pSlotList = (CK_SLOT_ID_PTR) malloc(ulCount*sizeof(CK_SLOT_ID));
2654     rv = C_GetSlotList(CK_FALSE, pSlotList, &ulCount);
2655     assert(rv == CKR_OK);
2656
2657     /* Get slot information for first slot */
2658     rv = C_GetSlotInfo(pSlotList[0], &slotInfo);
2659     assert(rv == CKR_OK);
2660
2661     /* Get token information for first slot */
2662     rv = C_GetTokenInfo(pSlotList[0], &tokenInfo);
2663     if (rv == CKR_TOKEN_NOT_PRESENT) {
2664         .
2665         .
2666     }
2667     .
2668     .
2669     free(pSlotList);
```

2670 }

2671 5.5.4 C_WaitForSlotEvent

```
2672 CK_DECLARE_FUNCTION(CK_RV, C_WaitForSlotEvent) (
2673     CK_FLAGS flags,
2674     CK_SLOT_ID_PTR pSlot,
2675     CK_VOID_PTR pReserved
2676 );
```

2677 **C_WaitForSlotEvent** waits for a slot event, such as token insertion or token removal, to occur. *flags* determines whether or not the **C_WaitForSlotEvent** call blocks (*i.e.*, waits for a slot event to occur); *pSlot* points to a location which will receive the ID of the slot that the event occurred in. *pReserved* is reserved for future versions; for this version of Cryptoki, it should be `NULL_PTR`.

2681 At present, the only flag defined for use in the *flags* argument is `CKF_DONT_BLOCK`:

2682 Internally, each Cryptoki application has a flag for each slot which is used to track whether or not any unrecognized events involving that slot have occurred. When an application initially calls **C_Initialize**, every slot's event flag is cleared. Whenever a slot event occurs, the flag corresponding to the slot in which the event occurred is set.

2686 If **C_WaitForSlotEvent** is called with the `CKF_DONT_BLOCK` flag set in the *flags* argument, and some slot's event flag is set, then that event flag is cleared, and the call returns with the ID of that slot in the 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 chosen by the library to have its event flag cleared and to have its slot ID returned.

2690 If **C_WaitForSlotEvent** is called with the `CKF_DONT_BLOCK` flag set in the *flags* argument, and no slot's event flag is set, then the call returns with the value `CKR_NO_EVENT`. In this case, the contents of the location pointed to by *pSlot* when **C_WaitForSlotEvent** are undefined.

2693 If **C_WaitForSlotEvent** is called with the `CKF_DONT_BLOCK` flag clear in the *flags* argument, then the 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, **C_WaitForSlotEvent** will wait until some slot's event flag becomes set. If a thread of an application has a **C_WaitForSlotEvent** call blocking when another thread of that application calls **C_Finalize**, the **C_WaitForSlotEvent** call returns with the value `CKR_CRYPTOKI_NOT_INITIALIZED`.

2698 Although the parameters supplied to **C_Initialize** can in general allow for safe multi-threaded access to a Cryptoki library, **C_WaitForSlotEvent** is exceptional in that the behavior of Cryptoki is undefined if multiple threads of a single application make simultaneous calls to **C_WaitForSlotEvent**.

2701 Return values: `CKR_ARGUMENTS_BAD`, `CKR_CRYPTOKI_NOT_INITIALIZED`,
2702 `CKR_FUNCTION_FAILED`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_NO_EVENT`,
2703 `CKR_OK`.

2704 Example:

```
2705 CK_FLAGS flags = 0;
2706 CK_SLOT_ID slotID;
2707 CK_SLOT_INFO slotInfo;
2708 CK_RV rv;
2709 .
2710 .
2711 /* Block and wait for a slot event */
2712 rv = C_WaitForSlotEvent(flags, &slotID, NULL_PTR);
2713 assert(rv == CKR_OK);
2714
2715 /* See what's up with that slot */
```

```
2716     rv = C_GetSlotInfo(slotID, &slotInfo);  
2717     assert(rv == CKR_OK);  
2718 }
```

2719 5.5.5 C_GetMechanismList

```
2720 CK_DECLARE_FUNCTION(CK_RV, C_GetMechanismList) (  
2721     CK_SLOT_ID slotID,  
2722     CK_MECHANISM_TYPE_PTR pMechanismList,  
2723     CK ULONG_PTR pulCount  
2724 );
```

2725 **C_GetMechanismList** is used to obtain a list of mechanism types supported by a token. *SlotID* is the ID
2726 of the token's slot; *pulCount* points to the location that receives the number of mechanisms.

2727 There are two ways for an application to call **C_GetMechanismList**:

- 2728 1. If *pMechanismList* is NULL_PTR, then all that **C_GetMechanismList** does is return (in **pulCount*)
2729 the number of mechanisms, without actually returning a list of mechanisms. The contents of
2730 **pulCount* on entry to **C_GetMechanismList** has no meaning in this case, and the call returns the
2731 value CKR_OK.
- 2732 2. If *pMechanismList* is not NULL_PTR, then **pulCount* MUST contain the size (in terms of
2733 **CK_MECHANISM_TYPE** elements) of the buffer pointed to by *pMechanismList*. If that buffer is large
2734 enough to hold the list of mechanisms, then the list is returned in it, and CKR_OK is returned. If not,
2735 then the call to **C_GetMechanismList** returns the value CKR_BUFFER_TOO_SMALL. In either
2736 case, the value **pulCount* is set to hold the number of mechanisms.

2737 Because **C_GetMechanismList** does not allocate any space of its own, an application will often call
2738 **C_GetMechanismList** twice. However, this behavior is by no means required.

2739 Return values: CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED,
2740 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
2741 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
2742 CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT, CKR_TOKEN_NOT_RECOGNIZED,
2743 CKR_ARGUMENTS_BAD.

2744 Example:

```
2745 CK_SLOT_ID slotID;  
2746 CK ULONG ulCount;  
2747 CK_MECHANISM_TYPE_PTR pMechanismList;  
2748 CK_RV rv;  
2749  
2750 .  
2751 .  
2752 rv = C_GetMechanismList(slotID, NULL_PTR, &ulCount);  
2753 if ((rv == CKR_OK) && (ulCount > 0)) {  
2754     pMechanismList =  
2755         (CK_MECHANISM_TYPE_PTR)  
2756         malloc(ulCount*sizeof(CK_MECHANISM_TYPE));  
2757     rv = C_GetMechanismList(slotID, pMechanismList, &ulCount);  
2758     if (rv == CKR_OK) {  
2759         .  
2760     }
```

```
2761     }
2762     free(pMechanismList);
2763 }
```

2764 5.5.6 C_GetMechanismInfo

```
2765 CK_DECLARE_FUNCTION(CK_RV, C_GetMechanismInfo) (
2766     CK_SLOT_ID slotID,
2767     CK_MECHANISM_TYPE type,
2768     CK_MECHANISM_INFO_PTR pInfo
2769 );
```

2770 **C_GetMechanismInfo** obtains information about a particular mechanism possibly supported by a token.
2771 *slotID* is the ID of the token's slot; *type* is the type of mechanism; *pInfo* points to the location that receives
2772 the mechanism information.

2773 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
2774 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
2775 CKR_HOST_MEMORY, CKR_MECHANISM_INVALID, CKR_OK, CKR_SLOT_ID_INVALID,
2776 CKR_TOKEN_NOT_PRESENT, CKR_TOKEN_NOT_RECOGNIZED, CKR_ARGUMENTS_BAD.

2777 Example:

```
2778 CK_SLOT_ID slotID;
2779 CK_MECHANISM_INFO info;
2780 CK_RV rv;
2781 .
2782 .
2783 .
2784 /* Get information about the CKM_MD2 mechanism for this token */
2785 rv = C_GetMechanismInfo(slotID, CKM_MD2, &info);
2786 if (rv == CKR_OK) {
2787     if (info.flags & CKF_DIGEST) {
2788         .
2789         .
2790     }
2791 }
```

2792 5.5.7 C_InitToken

```
2793 CK_DECLARE_FUNCTION(CK_RV, C_InitToken) (
2794     CK_SLOT_ID slotID,
2795     CK_UTF8CHAR_PTR pPin,
2796     CK_ULONG ulPinLen,
2797     CK_UTF8CHAR_PTR pLabel
2798 );
```

2799 **C_InitToken** initializes a token. *slotID* is the ID of the token's slot; *pPin* points to the SO's initial PIN
2800 (which need *not* be null-terminated); *ulPinLen* is the length in bytes of the PIN; *pLabel* points to the 32-
2801 byte label of the token (which MUST be padded with blank characters, and which MUST *not* be null-

2802 terminated). This standard allows PIN values to contain any valid UTF8 character, but the token may
 2803 impose subset restrictions.
 2804 If the token has not been initialized (i.e. new from the factory), then the *pPin* parameter becomes the
 2805 initial value of the SO PIN. If the token is being reinitialized, the *pPin* parameter is checked against the
 2806 existing SO PIN to authorize the initialization operation. In both cases, the SO PIN is the value *pPin* after
 2807 the function completes successfully. If the SO PIN is lost, then the card MUST be reinitialized using a
 2808 mechanism outside the scope of this standard. The **CKF_TOKEN_INITIALIZED** flag in the
 2809 **CK_TOKEN_INFO** structure indicates the action that will result from calling **C_InitToken**. If set, the token
 2810 will be reinitialized, and the client MUST supply the existing SO password in *pPin*.
 2811 When a token is initialized, all objects that can be destroyed are destroyed (i.e., all except for
 2812 “indestructible” objects such as keys built into the token). Also, access by the normal user is disabled
 2813 until the SO sets the normal user’s PIN. Depending on the token, some “default” objects may be created,
 2814 and attributes of some objects may be set to default values.
 2815 If the token has a “protected authentication path”, as indicated by the
 2816 **CKF_PROTECTED_AUTHENTICATION_PATH** flag in its **CK_TOKEN_INFO** being set, then that means
 2817 that there is some way for a user to be authenticated to the token without having the application send a
 2818 PIN through the Cryptoki library. One such possibility is that the user enters a PIN on a PINpad on the
 2819 token itself, or on the slot device. To initialize a token with such a protected authentication path, the *pPin*
 2820 parameter to **C_InitToken** should be **NULL_PTR**. During the execution of **C_InitToken**, the SO’s PIN will
 2821 be entered through the protected authentication path.
 2822 If the token has a protected authentication path other than a PINpad, then it is token-dependent whether
 2823 or not **C_InitToken** can be used to initialize the token.
 2824 A token cannot be initialized if Cryptoki detects that *any* application has an open session with it; when a
 2825 call to **C_InitToken** is made under such circumstances, the call fails with error **CKR_SESSION_EXISTS**.
 2826 Unfortunately, it may happen when **C_InitToken** is called that some other application *does* have an open
 2827 session with the token, but Cryptoki cannot detect this, because it cannot detect anything about other
 2828 applications using the token. If this is the case, then the consequences of the **C_InitToken** call are
 2829 undefined.
 2830 The **C_InitToken** function may not be sufficient to properly initialize complex tokens. In these situations,
 2831 an initialization mechanism outside the scope of Cryptoki MUST be employed. The definition of “complex
 2832 token” is product specific.
 2833 Return values: **CKR_CRYPTOKI_NOT_INITIALIZED**, **CKR_DEVICE_ERROR**, **CKR_DEVICE_MEMORY**,
 2834 **CKR_DEVICE_REMOVED**, **CKR_FUNCTION_CANCELED**, **CKR_FUNCTION_FAILED**,
 2835 **CKR_GENERAL_ERROR**, **CKR_HOST_MEMORY**, **CKR_OK**, **CKR_PIN_INCORRECT**,
 2836 **CKR_PIN_LOCKED**, **CKR_SESSION_EXISTS**, **CKR_SLOT_ID_INVALID**,
 2837 **CKR_TOKEN_NOT_PRESENT**, **CKR_TOKEN_NOT_RECOGNIZED**,
 2838 **CKR_TOKEN_WRITE_PROTECTED**, **CKR_ARGUMENTS_BAD**.
 2839 Example:

```

2840 CK_SLOT_ID slotID;
2841 CK_UTF8CHAR pin[] = {"MyPIN"};
2842 CK_UTF8CHAR label[32];
2843 CK_RV rv;
2844
2845 .
2846 .
2847 memset(label, ' ', sizeof(label));
2848 memcpy(label, "My first token", strlen("My first token"));
2849 rv = C_InitToken(slotID, pin, strlen(pin), label);
2850 if (rv == CKR_OK) {
2851 .

```

```
2852 .
2853 }
```

2854 5.5.8 C_InitPIN

```
2855 CK_DECLARE_FUNCTION(CK_RV, C_InitPIN) (
2856     CK_SESSION_HANDLE hSession,
2857     CK_UTF8CHAR_PTR pPin,
2858     CK ULONG ulPinLen
2859 );
```

2860 **C_InitPIN** initializes the normal user's PIN. *hSession* is the session's handle; *pPin* points to the normal
2861 user's PIN; *ulPinLen* is the length in bytes of the PIN. This standard allows PIN values to contain any
2862 valid UTF8 character, but the token may impose subset restrictions.

2863 **C_InitPIN** can only be called in the "R/W SO Functions" state. An attempt to call it from a session in any
2864 other state fails with error CKR_USER_NOT_LOGGED_IN.

2865 If the token has a "protected authentication path", as indicated by the
2866 CKF_PROTECTED_AUTHENTICATION_PATH flag in its **CK_TOKEN_INFO** being set, then that means
2867 that there is some way for a user to be authenticated to the token without having to send a PIN through
2868 the Cryptoki library. One such possibility is that the user enters a PIN on a PIN pad on the token itself, or
2869 on the slot device. To initialize the normal user's PIN on a token with such a protected authentication
2870 path, the *pPin* parameter to **C_InitPIN** should be NULL_PTR. During the execution of **C_InitPIN**, the SO
2871 will enter the new PIN through the protected authentication path.

2872 If the token has a protected authentication path other than a PIN pad, then it is token-dependent whether
2873 or not **C_InitPIN** can be used to initialize the normal user's token access.

2874 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
2875 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
2876 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_PIN_INVALID,
2877 CKR_PIN_LEN_RANGE, CKR_SESSION_CLOSED, CKR_SESSION_READ_ONLY,
2878 CKR_SESSION_HANDLE_INVALID, CKR_TOKEN_WRITE_PROTECTED,
2879 CKR_USER_NOT_LOGGED_IN, CKR_ARGUMENTS_BAD.

2880 Example:

```
2881 CK_SESSION_HANDLE hSession;
2882 CK_UTF8CHAR newPin[] = {"NewPIN"};
2883 CK_RV rv;
2884
2885 rv = C_InitPIN(hSession, newPin, sizeof(newPin)-1);
2886 if (rv == CKR_OK) {
2887 .
2888 .
2889 }
```

2890 5.5.9 C_SetPIN

```
2891 CK_DECLARE_FUNCTION(CK_RV, C_SetPIN) (
2892     CK_SESSION_HANDLE hSession,
2893     CK_UTF8CHAR_PTR pOldPin,
2894     CK ULONG ulOldLen,
2895     CK_UTF8CHAR_PTR pNewPin,
2896     CK ULONG ulNewLen
2897 );
```

2898 **C_SetPIN** modifies the PIN of the user that is currently logged in, or the CKU_USER PIN if the session is
2899 not logged in. *hSession* is the session's handle; *pOldPin* points to the old PIN; *ulOldLen* is the length in
2900 bytes of the old PIN; *pNewPin* points to the new PIN; *ulNewLen* is the length in bytes of the new PIN. This
2901 standard allows PIN values to contain any valid UTF8 character, but the token may impose subset
2902 restrictions.

2903 **C_SetPIN** can only be called in the "R/W Public Session" state, "R/W SO Functions" state, or "R/W User
2904 Functions" state. An attempt to call it from a session in any other state fails with error
2905 CKR_SESSION_READ_ONLY.

2906 If the token has a "protected authentication path", as indicated by the
2907 CKF_PROTECTED_AUTHENTICATION_PATH flag in its **CK_TOKEN_INFO** being set, then that means
2908 that there is some way for a user to be authenticated to the token without having to send a PIN through
2909 the Cryptoki library. One such possibility is that the user enters a PIN on a PIN pad on the token itself, or
2910 on the slot device. To modify the current user's PIN on a token with such a protected authentication path,
2911 the *pOldPin* and *pNewPin* parameters to **C_SetPIN** should be NULL_PTR. During the execution of
2912 **C_SetPIN**, the current user will enter the old PIN and the new PIN through the protected authentication
2913 path. It is not specified how the PIN pad should be used to enter two PINs; this varies.

2914 If the token has a protected authentication path other than a PIN pad, then it is token-dependent whether
2915 or not **C_SetPIN** can be used to modify the current user's PIN.

2916 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
2917 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
2918 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_PIN_INCORRECT,
2919 CKR_PIN_INVALID, CKR_PIN_LEN_RANGE, CKR_PIN_LOCKED, CKR_SESSION_CLOSED,
2920 CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY,
2921 CKR_TOKEN_WRITE_PROTECTED, CKR_ARGUMENTS_BAD.

2922 Example:

```
2923 CK_SESSION_HANDLE hSession;
2924 CK_UTF8CHAR oldPin[] = {"OldPIN"};
2925 CK_UTF8CHAR newPin[] = {"NewPIN"};
2926 CK_RV rv;
2927
2928 rv = C_SetPIN(
2929     hSession, oldPin, sizeof(oldPin)-1, newPin, sizeof(newPin)-1);
2930 if (rv == CKR_OK) {
2931     .
2932     .
2933 }
```

2934 5.6 Session management functions

2935 A typical application might perform the following series of steps to make use of a token (note that there
2936 are other reasonable sequences of events that an application might perform):

- 2937 1. Select a token.
- 2938 2. Make one or more calls to **C_OpenSession** to obtain one or more sessions with the token.
- 2939 3. Call **C_Login** to log the user into the token. Since all sessions an application has with a token have a
2940 shared login state, **C_Login** only needs to be called for one of the sessions.
- 2941 4. Perform cryptographic operations using the sessions with the token.
- 2942 5. Call **C_CloseSession** once for each session that the application has with the token, or call
2943 **C_CloseAllSessions** to close all the application's sessions simultaneously.

2944 As has been observed, an application may have concurrent sessions with more than one token. It is also
2945 possible for a token to have concurrent sessions with more than one application.
2946 Cryptoki provides the following functions for session management:

2947 **5.6.1 C_OpenSession**

```
2948 CK_DECLARE_FUNCTION(CK_RV, C_OpenSession) (
2949     CK_SLOT_ID slotID,
2950     CK_FLAGS flags,
2951     CK_VOID_PTR pApplication,
2952     CK_NOTIFY Notify,
2953     CK_SESSION_HANDLE_PTR phSession
2954 );
```

2955 **C_OpenSession** opens a session between an application and a token in a particular slot. *slotID* is the
2956 slot's ID; *flags* indicates the type of session; *pApplication* is an application-defined pointer to be passed to
2957 the notification callback; *Notify* is the address of the notification callback function (see Section 5.21);
2958 *phSession* points to the location that receives the handle for the new session.

2959 When opening a session with **C_OpenSession**, the *flags* parameter consists of the logical OR of zero or
2960 more bit flags defined in the **CK_SESSION_INFO** data type. For legacy reasons, the
2961 **CKF_SERIAL_SESSION** bit MUST always be set; if a call to **C_OpenSession** does not have this bit set,
2962 the call should return unsuccessfully with the error code
2963 **CKR_SESSION_PARALLEL_NOT_SUPPORTED**.

2964 There may be a limit on the number of concurrent sessions an application may have with the token, which
2965 may depend on whether the session is "read-only" or "read/write". An attempt to open a session which
2966 does not succeed because there are too many existing sessions of some type should return
2967 **CKR_SESSION_COUNT**.

2968 If the token is write-protected (as indicated in the **CK_TOKEN_INFO** structure), then only read-only
2969 sessions may be opened with it.

2970 If the application calling **C_OpenSession** already has a R/W SO session open with the token, then any
2971 attempt to open a R/O session with the token fails with error code
2972 **CKR_SESSION_READ_WRITE_SO_EXISTS** (see [PKCS11-UG] for further details).

2973 The *Notify* callback function is used by Cryptoki to notify the application of certain events. If the
2974 application does not wish to support callbacks, it should pass a value of **NULL_PTR** as the *Notify*
2975 parameter. See Section 5.21 for more information about application callbacks.

2976 Return values: **CKR_CRYPTOKI_NOT_INITIALIZED**, **CKR_DEVICE_ERROR**, **CKR_DEVICE_MEMORY**,
2977 **CKR_DEVICE_REMOVED**, **CKR_FUNCTION_FAILED**, **CKR_GENERAL_ERROR**,
2978 **CKR_HOST_MEMORY**, **CKR_OK**, **CKR_SESSION_COUNT**,
2979 **CKR_SESSION_PARALLEL_NOT_SUPPORTED**, **CKR_SESSION_READ_WRITE_SO_EXISTS**,
2980 **CKR_SLOT_ID_INVALID**, **CKR_TOKEN_NOT_PRESENT**, **CKR_TOKEN_NOT_RECOGNIZED**,
2981 **CKR_TOKEN_WRITE_PROTECTED**, **CKR_ARGUMENTS_BAD**.

2982 Example: see **C_CloseSession**.

2983 **5.6.2 C_CloseSession**

```
2984 CK_DECLARE_FUNCTION(CK_RV, C_CloseSession) (
2985     CK_SESSION_HANDLE hSession
2986 );
```

2987 **C_CloseSession** closes a session between an application and a token. *hSession* is the session's
2988 handle.

2989 When a session is closed, all session objects created by the session are destroyed automatically, even if
2990 the application has other sessions "using" the objects (see [PKCS11-UG] for further details).

2991 If this function is successful and it closes the last session between the application and the token, the login
2992 state of the token for the application returns to public sessions. Any new sessions to the token opened by
2993 the application will be either R/O Public or R/W Public sessions.
2994 Depending on the token, when the last open session any application has with the token is closed, the
2995 token may be "ejected" from its reader (if this capability exists).
2996 Despite the fact this **C_CloseSession** is supposed to close a session, the return value
2997 CKR_SESSION_CLOSED is an error return. It actually indicates the (probably somewhat unlikely) event
2998 that while this function call was executing, another call was made to **C_CloseSession** to close this
2999 particular session, and that call finished executing first. Such uses of sessions are a bad idea, and
3000 Cryptoki makes little promise of what will occur in general if an application indulges in this sort of
3001 behavior.
3002 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
3003 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3004 CKR_HOST_MEMORY, CKR_OK, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.
3005 Example:

```
3006 CK_SLOT_ID slotID;
3007 CK_BYTE application;
3008 CK_NOTIFY MyNotify;
3009 CK_SESSION_HANDLE hSession;
3010 CK_RV rv;
3011 .
3012 .
3013 .
3014 application = 17;
3015 MyNotify = &EncryptionSessionCallback;
3016 rv = C_OpenSession(
3017     slotID, CKF_SERIAL_SESSION | CKF_RW_SESSION,
3018             (CK_VOID_PTR) &application, MyNotify,
3019             &hSession);
3020 if (rv == CKR_OK) {
3021 .
3022 .
3023     C_CloseSession(hSession);
3024 }
```

3025 **5.6.3 C_CloseAllSessions**

```
3026 CK_DECLARE_FUNCTION(CK_RV, C_CloseAllSessions) (
3027     CK_SLOT_ID slotID
3028 );
```

3029 **C_CloseAllSessions** closes all sessions an application has with a token. *slotID* specifies the token's slot.
3030 When a session is closed, all session objects created by the session are destroyed automatically.
3031 After successful execution of this function, the login state of the token for the application returns to public
3032 sessions. Any new sessions to the token opened by the application will be either R/O Public or R/W
3033 Public sessions.
3034 Depending on the token, when the last open session any application has with the token is closed, the
3035 token may be "ejected" from its reader (if this capability exists).

```
3036 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,  
3037 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,  
3038 CKR_HOST_MEMORY, CKR_OK, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT.
```

3039 Example:

```
3040 CK_SLOT_ID slotID;  
3041 CK_RV rv;  
3042 .  
3043 .  
3044 rv = C_CloseAllSessions(slotID);
```

3046 5.6.4 C_GetSessionInfo

```
3047 CK_DECLARE_FUNCTION(CK_RV, C_GetSessionInfo) (  
3048     CK_SESSION_HANDLE hSession,  
3049     CK_SESSION_INFO_PTR pInfo  
3050 );
```

3051 **C_GetSessionInfo** obtains information about a session. *hSession* is the session's handle; *pInfo* points to
3052 the location that receives the session information.

3053 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
3054 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3055 CKR_HOST_MEMORY, CKR_OK, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
3056 CKR_ARGUMENTS_BAD.

3057 Example:

```
3058 CK_SESSION_HANDLE hSession;  
3059 CK_SESSION_INFO info;  
3060 CK_RV rv;  
3061 .  
3062 .  
3063 .  
3064 rv = C_GetSessionInfo(hSession, &info);  
3065 if (rv == CKR_OK) {  
3066     if (info.state == CKS_RW_USER_FUNCTIONS) {  
3067         .  
3068         .  
3069     }  
3070     .  
3071     .  
3072 }
```

3073 5.6.5 C_SessionCancel

```
3074 CK_DECLARE_FUNCTION(CK_RV, C_SessionCancel) (  
3075     CK_SESSION_HANDLE hSession  
3076     CK_FLAGS flags  
3077 );
```

3078 **C_SessionCancel** terminates active session based operations. *hSession* is the session's handle; *flags*
3079 indicates the operations to cancel.
3080 To identify which operation(s) should be terminated, the *flags* parameter should be assigned the logical
3081 bitwise OR of one or more of the bit flags defined in the **CK_MECHANISM_INFO** structure.
3082 If no flags are set, the session state will not be modified and CKR_OK will be returned.
3083 If a flag is set for an operation that has not been initialized in the session, the operation flag will be
3084 ignored and **C_SessionCancel** will behave as if the operation flag was not set.
3085 If any of the operations indicated by the *flags* parameter cannot be cancelled,
3086 CKR_OPERATION_CANCEL_FAILED must be returned. If multiple operation flags were set and
3087 CKR_OPERATION_CANCEL_FAILED is returned, this function does not provide any information about
3088 which operation(s) could not be cancelled. If an application desires to know if any single operation could
3089 not be cancelled, the application should not call **C_SessionCancel** with multiple flags set.
3090 If **C_SessionCancel** is called from an application callback (see Section 5.21), no action will be taken by
3091 the library and CKR_FUNCTION_FAILED must be returned.
3092 If **C_SessionCancel** is used to cancel one half of a dual-function operation, the remaining operation
3093 should still be left in an active state. However, it is expected that some Cryptoki implementations may not
3094 support this and return CKR_OPERATION_CANCEL_FAILED unless flags for both operations are
3095 provided.
3096 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
3097 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3098 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_CANCEL_FAILED,
3099 CKR_TOKEN_NOT_PRESENT.

3100 Example:

```
3101 CK_SESSION_HANDLE hSession;
3102 CK_RV rv;
3103
3104 rv = C_EncryptInit(hSession, &mechanism, hKey);
3105 if (rv != CKR_OK)
3106 {
3107     .
3108     .
3109 }
3110
3111 rv = C_SessionCancel (hSession, CKF_ENCRYPT);
3112 if (rv != CKR_OK)
3113 {
3114     .
3115     .
3116 }
3117
3118 rv = C_EncryptInit(hSession, &mechanism, hKey);
3119 if (rv != CKR_OK)
3120 {
3121     .
3122     .
3123 }
```

3124

3125

3126

3127

3128 Below are modifications to existing API descriptions to allow an alternate method of cancelling individual
3129 operations. The additional text is highlighted.

3130

5.6.6 C_GetOperationState

```
3131 CK_DECLARE_FUNCTION(CK_RV, C_GetOperationState) (
3132     CK_SESSION_HANDLE hSession,
3133     CK_BYTE_PTR pOperationState,
3134     CK ULONG_PTR pulOperationStateLen
3135 );
```

3136 **C_GetOperationState** obtains a copy of the cryptographic operations state of a session, encoded as a
3137 string of bytes. *hSession* is the session's handle; *pOperationState* points to the location that receives the
3138 state; *pulOperationStateLen* points to the location that receives the length in bytes of the state.

3139 Although the saved state output by **C_GetOperationState** is not really produced by a "cryptographic
3140 mechanism", **C_GetOperationState** nonetheless uses the convention described in Section 5.2 on
3141 producing output.

3142 Precisely what the "cryptographic operations state" this function saves is varies from token to token;
3143 however, this state is what is provided as input to **C_SetOperationState** to restore the cryptographic
3144 activities of a session.

3145 Consider a session which is performing a message digest operation using SHA-1 (*i.e.*, the session is
3146 using the **CKM_SHA_1** mechanism). Suppose that the message digest operation was initialized
3147 properly, and that precisely 80 bytes of data have been supplied so far as input to SHA-1. The
3148 application now wants to "save the state" of this digest operation, so that it can continue it later. In this
3149 particular case, since SHA-1 processes 512 bits (64 bytes) of input at a time, the cryptographic
3150 operations state of the session most likely consists of three distinct parts: the state of SHA-1's 160-bit
3151 internal chaining variable; the 16 bytes of unprocessed input data; and some administrative data
3152 indicating that this saved state comes from a session which was performing SHA-1 hashing. Taken
3153 together, these three pieces of information suffice to continue the current hashing operation at a later
3154 time.

3155 Consider next a session which is performing an encryption operation with DES (a block cipher with a
3156 block size of 64 bits) in CBC (cipher-block chaining) mode (*i.e.*, the session is using the **CKM_DES_CBC**
3157 mechanism). Suppose that precisely 22 bytes of data (in addition to an IV for the CBC mode) have been
3158 supplied so far as input to DES, which means that the first two 8-byte blocks of ciphertext have already
3159 been produced and output. In this case, the cryptographic operations state of the session most likely
3160 consists of three or four distinct parts: the second 8-byte block of ciphertext (this will be used for cipher-
3161 block chaining to produce the next block of ciphertext); the 6 bytes of data still awaiting encryption; some
3162 administrative data indicating that this saved state comes from a session which was performing DES
3163 encryption in CBC mode; and possibly the DES key being used for encryption (see **C_SetOperationState**
3164 for more information on whether or not the key is present in the saved state).

3165 If a session is performing two cryptographic operations simultaneously (see Section 5.14), then the
3166 cryptographic operations state of the session will contain all the necessary information to restore both
3167 operations.

3168 An attempt to save the cryptographic operations state of a session which does not currently have some
3169 active savable cryptographic operation(s) (encryption, decryption, digesting, signing without message
3170 recovery, verification without message recovery, or some legal combination of two of these) should fail
3171 with the error **CKR_OPERATION_NOT_INITIALIZED**.

3172 An attempt to save the cryptographic operations state of a session which is performing an appropriate
3173 cryptographic operation (or two), but which cannot be satisfied for any of various reasons (certain
3174 necessary state information and/or key information can't leave the token, for example) should fail with the
3175 error **CKR_STATE_UNSAVEABLE**.

3176 Return values: CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED,
3177 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
3178 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
3179 CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
3180 CKR_STATE_UNSAVEABLE, CKR_ARGUMENTS_BAD.
3181 Example: see **C_SetOperationState**.

3182 **5.6.7 C_SetOperationState**

```
3183 CK_DECLARE_FUNCTION(CK_RV, C_SetOperationState) (
3184     CK_SESSION_HANDLE hSession,
3185     CK_BYTE_PTR pOperationState,
3186     CK ULONG ulOperationStateLen,
3187     CK_OBJECT_HANDLE hEncryptionKey,
3188     CK_OBJECT_HANDLE hAuthenticationKey
3189 );
```

3190 **C_SetOperationState** restores the cryptographic operations state of a session from a string of bytes
3191 obtained with **C_GetOperationState**. *hSession* is the session's handle; *pOperationState* points to the
3192 location holding the saved state; *ulOperationStateLen* holds the length of the saved state;
3193 *hEncryptionKey* holds a handle to the key which will be used for an ongoing encryption or decryption
3194 operation in the restored session (or 0 if no encryption or decryption key is needed, either because no
3195 such operation is ongoing in the stored session or because all the necessary key information is present in
3196 the saved state); *hAuthenticationKey* holds a handle to the key which will be used for an ongoing
3197 signature, MACing, or verification operation in the restored session (or 0 if no such key is needed, either
3198 because no such operation is ongoing in the stored session or because all the necessary key information
3199 is present in the saved state).

3200 The state need not have been obtained from the same session (the "source session") as it is being
3201 restored to (the "destination session"). However, the source session and destination session should have
3202 a common session state (e.g., CKS_RW_USER_FUNCTIONS), and should be with a common token.
3203 There is also no guarantee that cryptographic operations state may be carried across logins, or across
3204 different Cryptoki implementations.

3205 If **C_SetOperationState** is supplied with alleged saved cryptographic operations state which it can
3206 determine is not valid saved state (or is cryptographic operations state from a session with a different
3207 session state, or is cryptographic operations state from a different token), it fails with the error
3208 CKR_SAVED_STATE_INVALID.

3209 Saved state obtained from calls to **C_GetOperationState** may or may not contain information about keys
3210 in use for ongoing cryptographic operations. If a saved cryptographic operations state has an ongoing
3211 encryption or decryption operation, and the key in use for the operation is not saved in the state, then it
3212 MUST be supplied to **C_SetOperationState** in the *hEncryptionKey* argument. If it is not, then
3213 **C_SetOperationState** will fail and return the error CKR_KEY_NEEDED. If the key in use for the
3214 operation is saved in the state, then it can be supplied in the *hEncryptionKey* argument, but this is not
3215 required.

3216 Similarly, if a saved cryptographic operations state has an ongoing signature, MACing, or verification
3217 operation, and the key in use for the operation is not saved in the state, then it MUST be supplied to
3218 **C_SetOperationState** in the *hAuthenticationKey* argument. If it is not, then **C_SetOperationState** will
3219 fail with the error CKR_KEY_NEEDED. If the key in use for the operation is saved in the state, then it can
3220 be supplied in the *hAuthenticationKey* argument, but this is not required.

3221 If an *irrelevant* key is supplied to **C_SetOperationState** call (e.g., a nonzero key handle is submitted in
3222 the *hEncryptionKey* argument, but the saved cryptographic operations state supplied does not have an
3223 ongoing encryption or decryption operation, then **C_SetOperationState** fails with the error
3224 CKR_KEY_NOT_NEEDED.

3225 If a key is supplied as an argument to **C_SetOperationState**, and **C_SetOperationState** can somehow
3226 detect that this key was not the key being used in the source session for the supplied cryptographic

3227 operations state (it may be able to detect this if the key or a hash of the key is present in the saved state,
3228 for example), then **C_SetOperationState** fails with the error CKR_KEY_CHANGED.

3229 An application can look at the **CKF_RESTORE_KEY_NOT_NEEDED** flag in the flags field of the
3230 **CK_TOKEN_INFO** field for a token to determine whether or not it needs to supply key handles to
3231 **C_SetOperationState** calls. If this flag is true, then a call to **C_SetOperationState** never needs a key
3232 handle to be supplied to it. If this flag is false, then at least some of the time, **C_SetOperationState**
3233 requires a key handle, and so the application should probably *always* pass in any relevant key handles
3234 when restoring cryptographic operations state to a session.

3235 **C_SetOperationState** can successfully restore cryptographic operations state to a session even if that
3236 session has active cryptographic or object search operations when **C_SetOperationState** is called (the
3237 ongoing operations are abruptly cancelled).

3238 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
3239 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3240 CKR_HOST_MEMORY, CKR_KEY_CHANGED, CKR_KEY_NEEDED, CKR_KEY_NOT_NEEDED,
3241 CKR_OK, CKR_SAVED_STATE_INVALID, CKR_SESSION_CLOSED,
3242 CKR_SESSION_HANDLE_INVALID, CKR_ARGUMENTS_BAD.

3243 Example:

```
3244 CK_SESSION_HANDLE hSession;
3245 CK_MECHANISM digestMechanism;
3246 CK_BYTE_PTR pState;
3247 CK_ULONG ulStateLen;
3248 CK_BYTE data1[] = {0x01, 0x03, 0x05, 0x07};
3249 CK_BYTE data2[] = {0x02, 0x04, 0x08};
3250 CK_BYTE data3[] = {0x10, 0x0F, 0x0E, 0x0D, 0x0C};
3251 CK_BYTE pDigest[20];
3252 CK_ULONG ulDigestLen;
3253 CK_RV rv;

3254 .
3255 .
3256 .
3257 /* Initialize hash operation */
3258 rv = C_DigestInit(hSession, &digestMechanism);
3259 assert(rv == CKR_OK);

3260 .
3261 /* Start hashing */
3262 rv = C_DigestUpdate(hSession, data1, sizeof(data1));
3263 assert(rv == CKR_OK);

3264 .
3265 /* Find out how big the state might be */
3266 rv = C_GetOperationState(hSession, NULL_PTR, &ulStateLen);
3267 assert(rv == CKR_OK);

3268 .
3269 /* Allocate some memory and then get the state */
3270 pState = (CK_BYTE_PTR) malloc(ulStateLen);
3271 rv = C_GetOperationState(hSession, pState, &ulStateLen);
```

```

3273 /* Continue hashing */
3274 rv = C_DigestUpdate(hSession, data2, sizeof(data2));
3275 assert(rv == CKR_OK);
3276
3277 /* Restore state. No key handles needed */
3278 rv = C_SetOperationState(hSession, pState, ulStateLen, 0, 0);
3279 assert(rv == CKR_OK);
3280
3281 /* Continue hashing from where we saved state */
3282 rv = C_DigestUpdate(hSession, data3, sizeof(data3));
3283 assert(rv == CKR_OK);
3284
3285 /* Conclude hashing operation */
3286 ulDigestLen = sizeof(pDigest);
3287 rv = C_DigestFinal(hSession, pDigest, &ulDigestLen);
3288 if (rv == CKR_OK) {
3289     /* pDigest[] now contains the hash of 0x01030507100F0E0D0C */
3290     .
3291     .
3292 }

```

3293 5.6.8 C_Login

```

3294 CK_DECLARE_FUNCTION(CK_RV, C_Login) (
3295     CK_SESSION_HANDLE hSession,
3296     CK_USER_TYPE userType,
3297     CK_UTF8CHAR_PTR pPin,
3298     CK_ULONG ulPinLen
3299 );

```

3300 **C_Login** logs a user into a token. *hSession* is a session handle; *userType* is the user type; *pPin* points to
 3301 the user's PIN; *ulPinLen* is the length of the PIN. This standard allows PIN values to contain any valid
 3302 UTF8 character, but the token may impose subset restrictions.

3303 When the user type is either CKU_SO or CKU_USER, if the call succeeds, each of the application's
 3304 sessions will enter either the "R/W SO Functions" state, the "R/W User Functions" state, or the "R/O User
 3305 Functions" state. If the user type is CKU_CONTEXT_SPECIFIC, the behavior of **C_Login** depends on the
 3306 context in which it is called. Improper use of this user type will result in a return value
 3307 CKR_OPERATION_NOT_INITIALIZED..

3308 If the token has a "protected authentication path", as indicated by the
 3309 **CKF_PROTECTED_AUTHENTICATION_PATH** flag in its **CK_TOKEN_INFO** being set, then that means
 3310 that there is some way for a user to be authenticated to the token without having to send a PIN through
 3311 the Cryptoki library. One such possibility is that the user enters a PIN on a PIN pad on the token itself, or
 3312 on the slot device. Or the user might not even use a PIN—authentication could be achieved by some
 3313 fingerprint-reading device, for example. To log into a token with a protected authentication path, the *pPin*
 3314 parameter to **C_Login** should be **NULL_PTR**. When **C_Login** returns, whatever authentication method
 3315 supported by the token will have been performed; a return value of **CKR_OK** means that the user was
 3316 successfully authenticated, and a return value of **CKR_PIN_INCORRECT** means that the user was
 3317 denied access.

3318 If there are any active cryptographic or object finding operations in an application's session, and then
3319 **C_Login** is successfully executed by that application, it may or may not be the case that those operations
3320 are still active. Therefore, before logging in, any active operations should be finished.
3321 If the application calling **C_Login** has a R/O session open with the token, then it will be unable to log the
3322 SO into a session (see [PKCS11-UG] for further details). An attempt to do this will result in the error code
3323 CKR_SESSION_READ_ONLY_EXISTS.
3324 **C_Login** may be called repeatedly, without intervening **C_Logout** calls, if (and only if) a key with the
3325 CKA_ALWAYS_AUTHENTICATE attribute set to CK_TRUE exists, and the user needs to do
3326 cryptographic operation on this key. See further Section 4.9.
3327 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
3328 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
3329 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3330 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_PIN_INCORRECT,
3331 CKR_PIN_LOCKED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
3332 CKR_SESSION_READ_ONLY_EXISTS, CKR_USER_ALREADY_LOGGED_IN,
3333 CKR_USER_ANOTHER_ALREADY_LOGGED_IN, CKR_USER_PIN_NOT_INITIALIZED,
3334 CKR_USER_TOO_MANY_TYPES, CKR_USER_TYPE_INVALID.

3335 Example: see **C_Logout**.

3336 5.6.9 **C_LoginUser**

```
3337 CK_DECLARE_FUNCTION(CK_RV, C_LoginUser) (
3338     CK_SESSION_HANDLE hSession,
3339     CK_USER_TYPE userType,
3340     CK_UTF8CHAR_PTR pPin,
3341     CK ULONG ulPinLen,
3342     CK_UTF8CHAR_PTR pUsername,
3343     CK ULONG ulUsernameLen
3344 );
```

3345 **C_LoginUser** logs a user into a token. *hSession* is a session handle; *userType* is the user type; *pPin*
3346 points to the user's PIN; *ulPinLen* is the length of the PIN, *pUsername* points to the user name,
3347 *ulUsernameLen* is the length of the user name. This standard allows PIN and user name values to
3348 contain any valid UTF8 character, but the token may impose subset restrictions.

3349 When the user type is either CKU_SO or CKU_USER, if the call succeeds, each of the application's
3350 sessions will enter either the "R/W SO Functions" state, the "R/W User Functions" state, or the "R/O User
3351 Functions" state. If the user type is CKU_CONTEXT_SPECIFIC, the behavior of **C_LoginUser** depends
3352 on the context in which it is called. Improper use of this user type will result in a return value
3353 CKR_OPERATION_NOT_INITIALIZED.

3354 If the token has a "protected authentication path", as indicated by the
3355 CKF_PROTECTED_AUTHENTICATION_PATH flag in its CK_TOKEN_INFO being set, then that means
3356 that there is some way for a user to be authenticated to the token without having to send a PIN through
3357 the Cryptoki library. One such possibility is that the user enters a PIN on a PIN pad on the token itself, or
3358 on the slot device. The user might not even use a PIN—authentication could be achieved by some
3359 fingerprint-reading device, for example. To log into a token with a protected authentication path, the *pPin*
3360 parameter to **C_LoginUser** should be NULL_PTR. When **C_LoginUser** returns, whatever authentication
3361 method supported by the token will have been performed; a return value of CKR_OK means that the user
3362 was successfully authenticated, and a return value of CKR_PIN_INCORRECT means that the user was
3363 denied access.

3364 If there are any active cryptographic or object finding operations in an application's session, and then
3365 **C_LoginUser** is successfully executed by that application, it may or may not be the case that those
3366 operations are still active. Therefore, before logging in, any active operations should be finished.

3367 If the application calling **C_LoginUser** has a R/O session open with the token, then it will be unable to log
3368 the SO into a session (see [PKCS11-UG] for further details). An attempt to do this will result in the error
3369 code CKR_SESSION_READ_ONLY_EXISTS.

3370 **C_LoginUser** may be called repeatedly, without intervening **C_Logout** calls, if (and only if) a key with the
3371 CKA_ALWAYS_AUTHENTICATE attribute set to CK_TRUE exists, and the user needs to do
3372 cryptographic operation on this key. See further Section 4.9.
3373 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
3374 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
3375 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3376 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_PIN_INCORRECT,
3377 CKR_PIN_LOCKED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
3378 CKR_SESSION_READ_ONLY_EXISTS, CKR_USER_ALREADY_LOGGED_IN,
3379 CKR_USER_ANOTHER_ALREADY_LOGGED_IN, CKR_USER_PIN_NOT_INITIALIZED,
3380 CKR_USER_TOO_MANY_TYPES, CKR_USER_TYPE_INVALID.

3381 Example:

```
3382 CK_SESSION_HANDLE hSession;
3383 CK_UTF8CHAR userPin[] = {"MyPIN"};
3384 CK_UTF8CHAR userName[] = {"MyUserName"};
3385 CK_RV rv;
3386
3387 rv = C_LoginUser(hSession, CKU_USER, userPin, sizeof(userPin)-1, userName,
3388 sizeof(userName)-1);
3389 if (rv == CKR_OK) {
3390 .
3391 .
3392 rv = C_Logout(hSession);
3393 if (rv == CKR_OK) {
3394 .
3395 .
3396 }
3397 }
```

3398 5.6.10 **C_Logout**

```
3399 CK_DECLARE_FUNCTION(CK_RV, C_Logout) (
3400     CK_SESSION_HANDLE hSession
3401 );
```

3402 **C_Logout** logs a user out from a token. *hSession* is the session's handle.

3403 Depending on the current user type, if the call succeeds, each of the application's sessions will enter
3404 either the "R/W Public Session" state or the "R/O Public Session" state.

3405 When **C_Logout** successfully executes, any of the application's handles to private objects become invalid
3406 (even if a user is later logged back into the token, those handles remain invalid). In addition, all private
3407 session objects from sessions belonging to the application are destroyed.

3408 If there are any active cryptographic or object-finding operations in an application's session, and then
3409 **C_Logout** is successfully executed by that application, it may or may not be the case that those
3410 operations are still active. Therefore, before logging out, any active operations should be finished.

3411 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
3412 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3413 CKR_HOST_MEMORY, CKR_OK, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
3414 CKR_USER_NOT_LOGGED_IN.

3415 Example:

```

3416 CK_SESSION_HANDLE hSession;
3417 CK_UTF8CHAR userPin[] = {"MyPIN"};
3418 CK_RV rv;
3419
3420 rv = C_Login(hSession, CKU_USER, userPin, sizeof(userPin)-1);
3421 if (rv == CKR_OK) {
3422 .
3423 .
3424     rv = C_Logout(hSession);
3425     if (rv == CKR_OK) {
3426 .
3427 .
3428 }
3429 }
```

3430 5.7 Object management functions

3431 Cryptoki provides the following functions for managing objects. Additional functions provided specifically
 3432 for managing key objects are described in Section 5.18.

3433 5.7.1 C_CreateObject

```

3434 CK_DECLARE_FUNCTION(CK_RV, C_CreateObject) (
3435     CK_SESSION_HANDLE hSession,
3436     CK_ATTRIBUTE_PTR pTemplate,
3437     CK ULONG ulCount,
3438     CK_OBJECT_HANDLE_PTR phObject
3439 );
```

3440 **C_CreateObject** creates a new object. *hSession* is the session's handle; *pTemplate* points to the object's
 3441 template; *ulCount* is the number of attributes in the template; *phObject* points to the location that receives
 3442 the new object's handle.

3443 If a call to **C_CreateObject** cannot support the precise template supplied to it, it will fail and return without
 3444 creating any object.

3445 If **C_CreateObject** is used to create a key object, the key object will have its **CKA_LOCAL** attribute set to
 3446 **CK_FALSE**. If that key object is a secret or private key then the new key will have the
 3447 **CKA_ALWAYS_SENSITIVE** attribute set to **CK_FALSE**, and the **CKA_NEVER_EXTRACTABLE**
 3448 attribute set to **CK_FALSE**.

3449 Only session objects can be created during a read-only session. Only public objects can be created
 3450 unless the normal user is logged in.

3451 Whenever an object is created, a value for **CKA_UNIQUE_ID** is generated and assigned to the new
 3452 object (See Section 4.4.1).

3453 Return values: **CKR_ARGUMENTS_BAD**, **CKR_ATTRIBUTE_READ_ONLY**,
 3454 **CKR_ATTRIBUTE_TYPE_INVALID**, **CKR_ATTRIBUTE_VALUE_INVALID**,
 3455 **CKR_CRYPTOKI_NOT_INITIALIZED**, **CKR_CURVE_NOT_SUPPORTED**, **CKR_DEVICE_ERROR**,
 3456 **CKR_DEVICE_MEMORY**, **CKR_DEVICE_REMOVED**, **CKR_DOMAIN_PARAMS_INVALID**,
 3457 **CKR_FUNCTION_FAILED**, **CKR_GENERAL_ERROR**, **CKR_HOST_MEMORY**, **CKR_OK**,
 3458 **CKR_PIN_EXPIRED**, **CKR_SESSION_CLOSED**, **CKR_SESSION_HANDLE_INVALID**,
 3459 **CKR_SESSION_READ_ONLY**, **CKR_TEMPLATE_INCOMPLETE**, **CKR_TEMPLATE_INCONSISTENT**,
 3460 **CKR_TOKEN_WRITE_PROTECTED**, **CKR_USER_NOT_LOGGED_IN**.

3461 Example:

```

3462 CK_SESSION_HANDLE hSession;
3463 CK_OBJECT_HANDLE
3464     hData,
3465     hCertificate,
3466     hKey;
3467 CK_OBJECT_CLASS
3468     dataClass = CKO_DATA,
3469     certificateClass = CKO_CERTIFICATE,
3470     keyClass = CKO_PUBLIC_KEY;
3471 CK_KEY_TYPE keyType = CKK_RSA;
3472 CK_UTF8CHAR application[] = {"My Application"};
3473 CK_BYTE dataValue[] = {...};
3474 CK_BYTE subject[] = {...};
3475 CK_BYTE id[] = {...};
3476 CK_BYTE certificateValue[] = {...};
3477 CK_BYTE modulus[] = {...};
3478 CK_BYTE exponent[] = {...};
3479 CK_BBOOL true = CK_TRUE;
3480 CK_ATTRIBUTE dataTemplate[] = {
3481     {CKA_CLASS, &dataClass, sizeof(dataClass)},
3482     {CKA_TOKEN, &true, sizeof(true)},
3483     {CKA_APPLICATION, application, sizeof(application)-1},
3484     {CKA_VALUE, dataValue, sizeof(dataValue)}
3485 };
3486 CK_ATTRIBUTE certificateTemplate[] = {
3487     {CKA_CLASS, &certificateClass, sizeof(certificateClass)},
3488     {CKA_TOKEN, &true, sizeof(true)},
3489     {CKA_SUBJECT, subject, sizeof(subject)},
3490     {CKA_ID, id, sizeof(id)},
3491     {CKA_VALUE, certificateValue, sizeof(certificateValue)}
3492 };
3493 CK_ATTRIBUTE keyTemplate[] = {
3494     {CKA_CLASS, &keyClass, sizeof(keyClass)},
3495     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
3496     {CKA_WRAP, &true, sizeof(true)},
3497     {CKA_MODULUS, modulus, sizeof(modulus)},
3498     {CKA_PUBLIC_EXPONENT, exponent, sizeof(exponent)}
3499 };
3500 CK_RV rv;
3501 .
3502 .
3503 /* Create a data object */

```

```

3505     rv = C_CreateObject(hSession, dataTemplate, 4, &hData);
3506     if (rv == CKR_OK) {
3507         .
3508         .
3509     }
3510
3511     /* Create a certificate object */
3512     rv = C_CreateObject(
3513         hSession, certificateTemplate, 5, &hCertificate);
3514     if (rv == CKR_OK) {
3515         .
3516         .
3517     }
3518
3519     /* Create an RSA public key object */
3520     rv = C_CreateObject(hSession, keyTemplate, 5, &hKey);
3521     if (rv == CKR_OK) {
3522         .
3523         .
3524     }

```

3525 5.7.2 C_CopyObject

```

3526 CK_DECLARE_FUNCTION(CK_RV, C_CopyObject) (
3527     CK_SESSION_HANDLE hSession,
3528     CK_OBJECT_HANDLE hObject,
3529     CK_ATTRIBUTE_PTR pTemplate,
3530     CK ULONG ulCount,
3531     CK_OBJECT_HANDLE_PTR phNewObject
3532 );

```

3533 **C_CopyObject** copies an object, creating a new object for the copy. *hSession* is the session's handle;
 3534 *hObject* is the object's handle; *pTemplate* points to the template for the new object; *ulCount* is the number
 3535 of attributes in the template; *phNewObject* points to the location that receives the handle for the copy of
 3536 the object.

3537 The template may specify new values for any attributes of the object that can ordinarily be modified (e.g.,
 3538 in the course of copying a secret key, a key's **CKA_EXTRACTABLE** attribute may be changed from
 3539 CK_TRUE to CK_FALSE, but not the other way around. If this change is made, the new key's
 3540 **CKA_NEVER_EXTRACTABLE** attribute will have the value CK_FALSE. Similarly, the template may
 3541 specify that the new key's **CKA_SENSITIVE** attribute be CK_TRUE; the new key will have the same
 3542 value for its **CKA_ALWAYS_SENSITIVE** attribute as the original key). It may also specify new values of
 3543 the **CKA_TOKEN** and **CKA_PRIVATE** attributes (e.g., to copy a session object to a token object). If the
 3544 template specifies a value of an attribute which is incompatible with other existing attributes of the object,
 3545 the call fails with the return code CKR_TEMPLATE_INCONSISTENT.

3546 If a call to **C_CopyObject** cannot support the precise template supplied to it, it will fail and return without
 3547 creating any object. If the object indicated by *hObject* has its **CKA_COPYABLE** attribute set to
 3548 CK_FALSE, **C_CopyObject** will return CKR_ACTION_PROHIBITED.

3549 Whenever an object is copied, a new value for **CKA_UNIQUE_ID** is generated and assigned to the new
 3550 object (See Section 4.4.1).

3551 Only session objects can be created during a read-only session. Only public objects can be created
3552 unless the normal user is logged in.
3553 Return values: , CKR_ACTION_PROHIBITED, CKR_ARGUMENTS_BAD,
3554 CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID,
3555 CKR_ATTRIBUTE_VALUE_INVALID, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR,
3556 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED,
3557 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OBJECT_HANDLE_INVALID, CKR_OK,
3558 CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
3559 CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCONSISTENT,
3560 CKR_TOKEN_WRITE_PROTECTED, CKR_USER_NOT_LOGGED_IN.

3561 Example:

```
3562 CK_SESSION_HANDLE hSession;
3563 CK_OBJECT_HANDLE hKey, hNewKey;
3564 CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;
3565 CK_KEY_TYPE keyType = CKK_DES;
3566 CK_BYTE id[] = {...};
3567 CK_BYTE keyValue[] = {...};
3568 CK_BBOOL false = CK_FALSE;
3569 CK_BBOOL true = CK_TRUE;
3570 CK_ATTRIBUTE keyTemplate[] = {
3571     {CKA_CLASS, &keyClass, sizeof(keyClass)},
3572     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
3573     {CKA_TOKEN, &false, sizeof(false)},
3574     {CKA_ID, id, sizeof(id)},
3575     {CKA_VALUE, keyValue, sizeof(keyValue)}}
3576 };
3577 CK_ATTRIBUTE copyTemplate[] = {
3578     {CKA_TOKEN, &true, sizeof(true)}}
3579 };
3580 CK_RV rv;
3581 .
3582 .
3583 .
3584 /* Create a DES secret key session object */
3585 rv = C_CreateObject(hSession, keyTemplate, 5, &hKey);
3586 if (rv == CKR_OK) {
3587     /* Create a copy which is a token object */
3588     rv = C_CopyObject(hSession, hKey, copyTemplate, 1, &hNewKey);
3589     .
3590     .
3591 }
```

3592 5.7.3 C_DestroyObject

```
3593 CK_DECLARE_FUNCTION(CK_RV, C_DestroyObject) (
3594     CK_SESSION_HANDLE hSession,
```

```

3595     CK_OBJECT_HANDLE hObject
3596 ) ;

```

3597 **C_DestroyObject** destroys an object. *hSession* is the session's handle; and *hObject* is the object's handle.

3598 Only session objects can be destroyed during a read-only session. Only public objects can be destroyed unless the normal user is logged in.

3601 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.

3604 Return values: CKR_ACTION_PROHIBITED, CKR_CRYPTOKI_NOT_INITIALIZED,
 3605 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
 3606 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
 3607 CKR_OBJECT_HANDLE_INVALID, CKR_OK, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
 3608 CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY,
 3609 CKR_TOKEN_WRITE_PROTECTED.

3610 Example: see **C_GetObjectSize**.

3611 5.7.4 C_GetObjectSize

```

3612 CK_DECLARE_FUNCTION(CK_RV, C_GetObjectSize) (
3613     CK_SESSION_HANDLE hSession,
3614     CK_OBJECT_HANDLE hObject,
3615     CK_ULONG_PTR pulSize
3616 );

```

3617 **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.

3619 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.

3623 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
 3624 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
 3625 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
 3626 CKR_INFORMATION_SENSITIVE, CKR_OBJECT_HANDLE_INVALID, CKR_OK,
 3627 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

3628 Example:

```

3629 CK_SESSION_HANDLE hSession;
3630 CK_OBJECT_HANDLE hObject;
3631 CK_OBJECT_CLASS dataClass = CKO_DATA;
3632 CK_UTF8CHAR application[] = {"My Application"};
3633 CK_BYTE value[] = {...};
3634 CK_BBOOL true = CK_TRUE;
3635 CK_ATTRIBUTE template[] = {
3636     {CKA_CLASS, &dataClass, sizeof(dataClass)},
3637     {CKA_TOKEN, &true, sizeof(true)},
3638     {CKA_APPLICATION, application, sizeof(application)-1},
3639     {CKA_VALUE, value, sizeof(value)}
3640 };
3641 CK_ULONG ulSize;

```

```

3642 CK_RV rv;
3643 .
3644 .
3645 .
3646 rv = C_CreateObject(hSession, template, 4, &hObject);
3647 if (rv == CKR_OK) {
3648     rv = C_GetObjectSize(hSession, hObject, &ulSize);
3649     if (rv != CKR_INFORMATION_SENSITIVE) {
3650         .
3651         .
3652     }
3653     .
3654     rv = C_DestroyObject(hSession, hObject);
3655     .
3656     .
3657 }

```

3658 5.7.5 C_GetAttributeValue

```

3659 CK_DECLARE_FUNCTION(CK_RV, C_GetAttributeValue) (
3660     CK_SESSION_HANDLE hSession,
3661     CK_OBJECT_HANDLE hObject,
3662     CK_ATTRIBUTE_PTR pTemplate,
3663     CK_ULONG ulCount
3664 );

```

3665 **C_GetAttributeValue** obtains the value of one or more attributes of an object. *hSession* is the session's
 3666 handle; *hObject* is the object's handle; *pTemplate* points to a template that specifies which attribute
 3667 values are to be obtained, and receives the attribute values; *ulCount* is the number of attributes in the
 3668 template.

3669 For each (*type*, *pValue*, *ulValueLen*) triple in the template, **C_GetAttributeValue** performs the following
 3670 algorithm:

- 3671 1. If the specified attribute (i.e., the attribute specified by the type field) for the object cannot be revealed
 3672 because the object is sensitive or unextractable, then the *ulValueLen* field in that triple is modified to
 3673 hold the value CK_UNAVAILABLE_INFORMATION.
- 3674 2. Otherwise, if the specified value for the object is invalid (the object does not possess such an
 3675 attribute), then the *ulValueLen* field in that triple is modified to hold the value
 3676 CK_UNAVAILABLE_INFORMATION.
- 3677 3. Otherwise, if the *pValue* field has the value NULL_PTR, then the *ulValueLen* field is modified to hold
 3678 the exact length of the specified attribute for the object.
- 3679 4. Otherwise, if the length specified in *ulValueLen* is large enough to hold the value of the specified
 3680 attribute for the object, then that attribute is copied into the buffer located at *pValue*, and the
 3681 *ulValueLen* field is modified to hold the exact length of the attribute.
- 3682 5. Otherwise, the *ulValueLen* field is modified to hold the value CK_UNAVAILABLE_INFORMATION.

3683 If case 1 applies to any of the requested attributes, then the call should return the value
 3684 CKR_ATTRIBUTE_SENSITIVE. If case 2 applies to any of the requested attributes, then the call should
 3685 return the value CKR_ATTRIBUTE_TYPE_INVALID. If case 5 applies to any of the requested attributes,
 3686 then the call should return the value CKR_BUFFER_TOO_SMALL. As usual, if more than one of these
 3687 error codes is applicable, Cryptoki may return any of them. Only if none of them applies to any of the
 3688 requested attributes will CKR_OK be returned.

3689 In the special case of an attribute whose value is an array of attributes, for example
 3690 CKA_WRAP_TEMPLATE, where it is passed in with pValue not NULL, the length specified in ulValueLen
 3691 MUST be large enough to hold all attributes in the array. If the pValue of elements within the array is
 3692 NULL_PTR then the ulValueLen of elements within the array will be set to the required length. If the
 3693 pValue of elements within the array is not NULL_PTR, then the ulValueLen element of attributes within
 3694 the array MUST reflect the space that the corresponding pValue points to, and pValue is filled in if there is
 3695 sufficient room. Therefore it is important to initialize the contents of a buffer before calling
 3696 C_GetAttributeValue to get such an array value. Note that the type element of attributes within the array
 3697 MUST be ignored on input and MUST be set on output. If any ulValueLen within the array isn't large
 3698 enough, it will be set to CK_UNAVAILABLE_INFORMATION and the function will return
 3699 CKR_BUFFER_TOO_SMALL, as it does if an attribute in the pTemplate argument has ulValueLen too
 3700 small. Note that any attribute whose value is an array of attributes is identifiable by virtue of the attribute
 3701 type having the CKF_ARRAY_ATTRIBUTE bit set.
 3702 Note that the error codes CKR_ATTRIBUTE_SENSITIVE, CKR_ATTRIBUTE_TYPE_INVALID, and
 3703 CKR_BUFFER_TOO_SMALL do not denote true errors for **C_GetAttributeValue**. If a call to
 3704 **C_GetAttributeValue** returns any of these three values, then the call MUST nonetheless have processed
 3705 every attribute in the template supplied to **C_GetAttributeValue**. Each attribute in the template whose
 3706 value *can be* returned by the call to **C_GetAttributeValue** *will be* returned by the call to
 3707 **C_GetAttributeValue**.
 3708 Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_SENSITIVE,
 3709 CKR_ATTRIBUTE_TYPE_INVALID, CKR_BUFFER_TOO_SMALL,
 3710 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
 3711 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
 3712 CKR_HOST_MEMORY, CKR_OBJECT_HANDLE_INVALID, CKR_OK, CKR_SESSION_CLOSED,
 3713 CKR_SESSION_HANDLE_INVALID.

3714 Example:

```

3715 CK_SESSION_HANDLE hSession;
3716 CK_OBJECT_HANDLE hObject;
3717 CK_BYTE_PTR pModulus, pExponent;
3718 CK_ATTRIBUTE template[] = {
3719   {CKA_MODULUS, NULL_PTR, 0},
3720   {CKA_PUBLIC_EXPONENT, NULL_PTR, 0}
3721 };
3722 CK_RV rv;
3723 .
3724 .
3725 .
3726 rv = C_GetAttributeValue(hSession, hObject, template, 2);
3727 if (rv == CKR_OK) {
3728   pModulus = (CK_BYTE_PTR) malloc(template[0].ulValueLen);
3729   template[0].pValue = pModulus;
3730   /* template[0].ulValueLen was set by C_GetAttributeValue */
3731
3732   pExponent = (CK_BYTE_PTR) malloc(template[1].ulValueLen);
3733   template[1].pValue = pExponent;
3734   /* template[1].ulValueLen was set by C_GetAttributeValue */
3735
3736   rv = C_GetAttributeValue(hSession, hObject, template, 2);

```

```
3737     if (rv == CKR_OK) {
3738         .
3739         .
3740     }
3741     free(pModulus);
3742     free(pExponent);
3743 }
```

3744 5.7.6 C_SetAttributeValue

```
3745 CK_DECLARE_FUNCTION(CK_RV, C_SetAttributeValue) (
3746     CK_SESSION_HANDLE hSession,
3747     CK_OBJECT_HANDLE hObject,
3748     CK_ATTRIBUTE_PTR pTemplate,
3749     CK_ULONG ulCount
3750 );
```

3751 **C_SetAttributeValue** modifies the value of one or more attributes of an object. *hSession* is the session's
3752 handle; *hObject* is the object's handle; *pTemplate* points to a template that specifies which attribute
3753 values are to be modified and their new values; *ulCount* is the number of attributes in the template.

3754 Certain objects may not be modified. Calling C_SetAttributeValue on such objects will result in the
3755 CKR_ACTION_PROHIBITED error code. An application can consult the object's CKA_MODIFIABLE
3756 attribute to determine if an object may be modified or not.

3757 Only session objects can be modified during a read-only session.

3758 The template may specify new values for any attributes of the object that can be modified. If the template
3759 specifies a value of an attribute which is incompatible with other existing attributes of the object, the call
3760 fails with the return code CKR_TEMPLATE_INCONSISTENT.

3761 Not all attributes can be modified; see Section 4.1.2 for more details.

3762 Return values: CKR_ACTION_PROHIBITED, CKR_ARGUMENTS_BAD,
3763 CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID,
3764 CKR_ATTRIBUTE_VALUE_INVALID, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR,
3765 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED,
3766 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OBJECT_HANDLE_INVALID, CKR_OK,
3767 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY,
3768 CKR_TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED,
3769 CKR_USER_NOT_LOGGED_IN.

3770 Example:

```
3771 CK_SESSION_HANDLE hSession;
3772 CK_OBJECT_HANDLE hObject;
3773 CK_UTF8CHAR label[] = {"New label"};
3774 CK_ATTRIBUTE template[] = {
3775     {CKA_LABEL, label, sizeof(label)-1}
3776 };
3777 CK_RV rv;
3778
3779 .
3780 .
3781 rv = C_SetAttributeValue(hSession, hObject, template, 1);
3782 if (rv == CKR_OK) {
```

```
3783 .  
3784 .  
3785 }
```

3786 5.7.7 C_FindObjectsInit

```
3787 CK_DECLARE_FUNCTION(CK_RV, C_FindObjectsInit) (  
3788     CK_SESSION_HANDLE hSession,  
3789     CK_ATTRIBUTE_PTR pTemplate,  
3790     CK ULONG ulCount  
3791 );
```

3792 **C_FindObjectsInit** initializes a search for token and session objects that match a template. *hSession* is
3793 the session's handle; *pTemplate* points to a search template that specifies the attribute values to match;
3794 *ulCount* is the number of attributes in the search template. The matching criterion is an exact byte-for-
3795 byte match with all attributes in the template. To find all objects, set *ulCount* to 0.

3796 After calling **C_FindObjectsInit**, the application may call **C_FindObjects** one or more times to obtain
3797 handles for objects matching the template, and then eventually call **C_FindObjectsFinal** to finish the
3798 active search operation. At most one search operation may be active at a given time in a given session.

3799 The object search operation will only find objects that the session can view. For example, an object
3800 search in an "R/W Public Session" will not find any private objects (even if one of the attributes in the
3801 search template specifies that the search is for private objects).

3802 If a search operation is active, and objects are created or destroyed which fit the search template for the
3803 active search operation, then those objects may or may not be found by the search operation. Note that
3804 this means that, under these circumstances, the search operation may return invalid object handles.

3805 Even though **C_FindObjectsInit** can return the values CKR_ATTRIBUTE_TYPE_INVALID and
3806 CKR_ATTRIBUTE_VALUE_INVALID, it is not required to. For example, if it is given a search template
3807 with nonexistent attributes in it, it can return CKR_ATTRIBUTE_TYPE_INVALID, or it can initialize a
3808 search operation which will match no objects and return CKR_OK.

3809 If the CKA_UNIQUE_ID attribute is present in the search template, either zero or one objects will be
3810 found, since at most one object can have any particular CKA_UNIQUE_ID value.

3811 Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_TYPE_INVALID,
3812 CKR_ATTRIBUTE_VALUE_INVALID, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR,
3813 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED,
3814 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE,
3815 CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

3816 Example: see **C_FindObjectsFinal**.

3817 5.7.8 C_FindObjects

```
3818 CK_DECLARE_FUNCTION(CK_RV, C_FindObjects) (  
3819     CK_SESSION_HANDLE hSession,  
3820     CK_OBJECT_HANDLE_PTR phObject,  
3821     CK ULONG ulMaxObjectCount,  
3822     CK ULONG_PTR pulObjectCount  
3823 );
```

3824 **C_FindObjects** continues a search for token and session objects that match a template, obtaining
3825 additional object handles. *hSession* is the session's handle; *phObject* points to the location that receives
3826 the list (array) of additional object handles; *ulMaxObjectCount* is the maximum number of object handles
3827 to be returned; *pulObjectCount* points to the location that receives the actual number of object handles
3828 returned.

3829 If there are no more objects matching the template, then the location that *pulObjectCount* points to
3830 receives the value 0.

3831 The search MUST have been initialized with **C_FindObjectsInit**.
3832 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
3833 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
3834 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
3835 CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.
3836 Example: see **C_FindObjectsFinal**.

3837 **5.7.9 C_FindObjectsFinal**

```
3838 CK_DECLARE_FUNCTION(CK_RV, C_FindObjectsFinal) (
3839     CK_SESSION_HANDLE hSession
3840 );
```

3841 **C_FindObjectsFinal** terminates a search for token and session objects. *hSession* is the session's
3842 handle.

3843 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
3844 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3845 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
3846 CKR_SESSION_HANDLE_INVALID.

3847 Example:

```
3848 CK_SESSION_HANDLE hSession;
3849 CK_OBJECT_HANDLE hObject;
3850 CK_ULONG ulObjectCount;
3851 CK_RV rv;
3852 .
3853 .
3854 .
3855 rv = C_FindObjectsInit(hSession, NULL_PTR, 0);
3856 assert(rv == CKR_OK);
3857 while (1) {
3858     rv = C_FindObjects(hSession, &hObject, 1, &ulObjectCount);
3859     if (rv != CKR_OK || ulObjectCount == 0)
3860         break;
3861     .
3862     .
3863 }
3864 .
3865 rv = C_FindObjectsFinal(hSession);
3866 assert(rv == CKR_OK);
```

3867 **5.8 Encryption functions**

3868 Cryptoki provides the following functions for encrypting data:

3869 **5.8.1 C_EncryptInit**

```
3870 CK_DECLARE_FUNCTION(CK_RV, C_EncryptInit) (
3871     CK_SESSION_HANDLE hSession,
3872     CK_MECHANISM_PTR pMechanism,
```

```

3873     CK_OBJECT_HANDLE hKey
3874 ) ;
3875
3876 C_EncryptInit initializes an encryption operation. hSession is the session's handle; pMechanism points
3877 to the encryption mechanism; hKey is the handle of the encryption key.
3878
3879 The CKA_ENCRYPT attribute of the encryption key, which indicates whether the key supports
3880 encryption, MUST be CK_TRUE.
3881
3882 After calling C_EncryptInit, the application can either call C_Encrypt to encrypt data in a single part; or
3883 call C_EncryptUpdate zero or more times, followed by C_EncryptFinal, to encrypt data in multiple parts.
3884 The encryption operation is active until the application uses a call to C_Encrypt or C_EncryptFinal to
3885 actually obtain the final piece of ciphertext. To process additional data (in single or multiple parts), the
3886 application MUST call C_EncryptInit again.
3887
3888 C_EncryptInit can be called with pMechanism set to NULL_PTR to terminate an active encryption
3889 operation. If an active operation operations cannot be cancelled, CKR_OPERATION_CANCEL_FAILED
3890 must be returned.
3891
3892 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
3893 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
3894 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED,
3895 CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT,
3896 CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK,
3897 CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
3898 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
3899 CKR_OPERATION_CANCEL_FAILED.
3900
3901 Example: see C_EncryptFinal.

```

3896 5.8.2 C_Encrypt

```

3897 CK_DECLARE_FUNCTION(CK_RV, C_Encrypt) (
3898     CK_SESSION_HANDLE hSession,
3899     CK_BYTE_PTR pData,
3900     CK ULONG ulDataLen,
3901     CK_BYTE_PTR pEncryptedData,
3902     CK ULONG_PTR pulEncryptedDataLen
3903 ) ;
3904
3905 C_Encrypt encrypts single-part data. hSession is the session's handle; pData points to the data;
3906 ulDataLen is the length in bytes of the data; pEncryptedData points to the location that receives the
3907 encrypted data; pulEncryptedDataLen points to the location that holds the length in bytes of the encrypted
3908 data.
3909
3910 C_Encrypt uses the convention described in Section 5.2 on producing output.
3911 The encryption operation MUST have been initialized with C_EncryptInit. A call to C_Encrypt always
3912 terminates the active encryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a
3913 successful call (i.e., one which returns CKR_OK) to determine the length of the buffer needed to hold the
3914 ciphertext.
3915
3916 C_Encrypt cannot be used to terminate a multi-part operation, and MUST be called after C_EncryptInit
3917 without intervening C_EncryptUpdate calls.
3918
3919 For some encryption mechanisms, the input plaintext data has certain length constraints (either because
3920 the mechanism can only encrypt relatively short pieces of plaintext, or because the mechanism's input
3921 data MUST consist of an integral number of blocks). If these constraints are not satisfied, then
3922 C_Encrypt will fail with return code CKR_DATA_LEN_RANGE.
3923
3924 The plaintext and ciphertext can be in the same place, i.e., it is OK if pData and pEncryptedData point to
3925 the same location.
3926
3927 For most mechanisms, C_Encrypt is equivalent to a sequence of C_EncryptUpdate operations followed
3928 by C_EncryptFinal.

```

3923 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
3924 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE,
3925 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
3926 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3927 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
3928 CKR_SESSION_HANDLE_INVALID.

3929 Example: see **C_EncryptFinal** for an example of similar functions.

3930 5.8.3 C_EncryptUpdate

```
3931 CK_DECLARE_FUNCTION(CK_RV, C_EncryptUpdate) (
3932     CK_SESSION_HANDLE hSession,
3933     CK_BYTE_PTR pPart,
3934     CK_ULONG ulPartLen,
3935     CK_BYTE_PTR pEncryptedPart,
3936     CK_ULONG_PTR pulEncryptedPartLen
3937 );
```

3938 **C_EncryptUpdate** continues a multiple-part encryption operation, processing another data part.
3939 *hSession* is the session's handle; *pPart* points to the data part; *ulPartLen* is the length of the data part;
3940 *pEncryptedPart* points to the location that receives the encrypted data part; *pulEncryptedPartLen* points
3941 to the location that holds the length in bytes of the encrypted data part.

3942 **C_EncryptUpdate** uses the convention described in Section 5.2 on producing output.

3943 The encryption operation MUST have been initialized with **C_EncryptInit**. This function may be called
3944 any number of times in succession. A call to **C_EncryptUpdate** which results in an error other than
3945 CKR_BUFFER_TOO_SMALL terminates the current encryption operation.

3946 The plaintext and ciphertext can be in the same place, *i.e.*, it is OK if *pPart* and *pEncryptedPart* point to
3947 the same location.

3948 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
3949 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR,
3950 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED,
3951 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
3952 CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

3953 Example: see **C_EncryptFinal**.

3954 5.8.4 C_EncryptFinal

```
3955 CK_DECLARE_FUNCTION(CK_RV, C_EncryptFinal) (
3956     CK_SESSION_HANDLE hSession,
3957     CK_BYTE_PTR pLastEncryptedPart,
3958     CK_ULONG_PTR pulLastEncryptedPartLen
3959 );
```

3960 **C_EncryptFinal** finishes a multiple-part encryption operation. *hSession* is the session's handle;
3961 *pLastEncryptedPart* points to the location that receives the last encrypted data part, if any;
3962 *pulLastEncryptedPartLen* points to the location that holds the length of the last encrypted data part.

3963 **C_EncryptFinal** uses the convention described in Section 5.2 on producing output.

3964 The encryption operation MUST have been initialized with **C_EncryptInit**. A call to **C_EncryptFinal**
3965 always terminates the active encryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a
3966 successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the
3967 ciphertext.

3968 For some multi-part encryption mechanisms, the input plaintext data has certain length constraints,
3969 because the mechanism's input data MUST consist of an integral number of blocks. If these constraints
3970 are not satisfied, then **C_EncryptFinal** will fail with return code CKR_DATA_LEN_RANGE.

3971 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
3972 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR,
3973 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED,
3974 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
3975 CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.
3976 Example:

```
3977 #define PLAINTEXT_BUF_SZ 200
3978 #define CIPHERTEXT_BUF_SZ 256
3979
3980 CK_ULONG firstPieceLen, secondPieceLen;
3981 CK_SESSION_HANDLE hSession;
3982 CK_OBJECT_HANDLE hKey;
3983 CK_BYTE iv[8];
3984 CK_MECHANISM mechanism = {
3985     CKM_DES_CBC_PAD, iv, sizeof(iv)
3986 };
3987 CK_BYTE data[PLAINTEXT_BUF_SZ];
3988 CK_BYTE encryptedData[CIPHERTEXT_BUF_SZ];
3989 CK_ULONG ulEncryptedData1Len;
3990 CK_ULONG ulEncryptedData2Len;
3991 CK_ULONG ulEncryptedData3Len;
3992 CK_RV rv;
3993
3994 .
3995 .
3996 firstPieceLen = 90;
3997 secondPieceLen = PLAINTEXT_BUF_SZ-firstPieceLen;
3998 rv = C_EncryptInit(hSession, &mechanism, hKey);
3999 if (rv == CKR_OK) {
4000     /* Encrypt first piece */
4001     ulEncryptedData1Len = sizeof(encryptedData);
4002     rv = C_EncryptUpdate(
4003         hSession,
4004         &data[0], firstPieceLen,
4005         &encryptedData[0], &ulEncryptedData1Len);
4006     if (rv != CKR_OK) {
4007         .
4008         .
4009     }
4010
4011     /* Encrypt second piece */
4012     ulEncryptedData2Len = sizeof(encryptedData)-ulEncryptedData1Len;
4013     rv = C_EncryptUpdate(
4014         hSession,
```

```

4015     &data[firstPieceLen], secondPieceLen,
4016     &encryptedData[ulEncryptedData1Len], &ulEncryptedData2Len);
4017     if (rv != CKR_OK) {
4018     .
4019     .
4020   }
4021
4022   /* Get last little encrypted bit */
4023   ulEncryptedData3Len =
4024     sizeof(encryptedData)-ulEncryptedData1Len-ulEncryptedData2Len;
4025   rv = C_EncryptFinal(
4026     hSession,
4027     &encryptedData[ulEncryptedData1Len+ulEncryptedData2Len],
4028     &ulEncryptedData3Len);
4029   if (rv != CKR_OK) {
4030     .
4031     .
4032   }
4033 }
```

4034 5.9 Message-based encryption functions

4035 Message-based encryption refers to the process of encrypting multiple messages using the same
 4036 encryption mechanism and encryption key. The encryption mechanism can be either an authenticated
 4037 encryption with associated data (AEAD) algorithm or a pure encryption algorithm.

4038 Cryptoki provides the following functions for message-based encryption:

4039 5.9.1 C_MessageEncryptInit

```

4040 CK_DECLARE_FUNCTION(CK_RV, C_MessageEncryptInit) (
4041   CK_SESSION_HANDLE hSession,
4042   CK_MECHANISM_PTR pMechanism,
4043   CK_OBJECT_HANDLE hKey
4044 );
```

4045 **C_MessageEncryptInit** prepares a session for one or more encryption operations that use the same
 4046 encryption mechanism and encryption key. *hSession* is the session's handle; *pMechanism* points to the
 4047 encryption mechanism; *hKey* is the handle of the encryption key.

4048 The **CKA_ENCRYPT** attribute of the encryption key, which indicates whether the key supports encryption,
 4049 MUST be **CK_TRUE**.

4050 After calling **C_MessageEncryptInit**, the application can either call **C_EncryptMessage** to encrypt a
 4051 message in a single part, or call **C_EncryptMessageBegin**, followed by **C_EncryptMessageNext** one or
 4052 more times, to encrypt a message in multiple parts. This may be repeated several times. The message-
 4053 based encryption process is active until the application calls **C_MessageEncryptFinal** to finish the
 4054 message-based encryption process.

4055 **C_MessageEncryptInit** can be called with *pMechanism* set to **NULL_PTR** to terminate a message-based
 4056 encryption process. If a multi-part message encryption operation is active, it will also be terminated. If an
 4057 active operation has been initialized and it cannot be cancelled, **CKR_OPERATION_CANCEL_FAILED**
 4058 must be returned.

4059 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4060 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
4061 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED,
4062 CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT,
4063 CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK,
4064 CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
4065 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
4066 CKR_OPERATION_CANCEL FAILED.

4067 5.9.2 C_EncryptMessage

```
4068 CK_DECLARE_FUNCTION(CK_RV, C_EncryptMessage) (
4069     CK_SESSION_HANDLE hSession,
4070     CK_VOID_PTR pParameter,
4071     CK_ULONG ulParameterLen,
4072     CK_BYTE_PTR pAssociatedData,
4073     CK_ULONG ulAssociatedDataLen,
4074     CK_BYTE_PTR pPlaintext,
4075     CK_ULONG ulPlaintextLen,
4076     CK_BYTE_PTR pCiphertext,
4077     CK_ULONG_PTR pulCiphertextLen
4078 );
```

4079 **C_EncryptMessage** encrypts a message in a single part. *hSession* is the session's handle; *pParameter*
4080 and *ulParameterLen* specify any mechanism-specific parameters for the message encryption operation;
4081 *pAssociatedData* and *ulAssociatedDataLen* specify the associated data for an AEAD mechanism;
4082 *pPlaintext* points to the plaintext data; *ulPlaintextLen* is the length in bytes of the plaintext data;
4083 *pCiphertext* points to the location that receives the encrypted data; *pulCiphertextLen* points to the location
4084 that holds the length in bytes of the encrypted data.

4085 Typically, *pParameter* is an initialization vector (IV) or nonce. Depending on the mechanism parameter
4086 passed to **C_MessageEncryptInit**, *pParameter* may be either an input or an output parameter. For
4087 example, if the mechanism parameter specifies an IV generator mechanism, the IV generated by the IV
4088 generator will be output to the *pParameter* buffer.

4089 If the encryption mechanism is not AEAD, *pAssociatedData* and *ulAssociatedDataLen* are not used and
4090 should be set to (NULL, 0).

4091 **C_EncryptMessage** uses the convention described in Section 5.2 on producing output.

4092 The message-based encryption process MUST have been initialized with **C_MessageEncryptInit**. A call
4093 to **C_EncryptMessage** begins and terminates a message encryption operation.

4094 **C_EncryptMessage** cannot be called in the middle of a multi-part message encryption operation.

4095 For some encryption mechanisms, the input plaintext data has certain length constraints (either because
4096 the mechanism can only encrypt relatively short pieces of plaintext, or because the mechanism's input
4097 data MUST consist of an integral number of blocks). If these constraints are not satisfied, then
4098 **C_EncryptMessage** will fail with return code CKR_DATA_LEN_RANGE. The plaintext and ciphertext can
4099 be in the same place, i.e., it is OK if *pPlaintext* and *pCiphertext* point to the same location.

4100 For most mechanisms, **C_EncryptMessage** is equivalent to **C_EncryptMessageBegin** followed by a
4101 sequence of **C_EncryptMessageNext** operations.

4102 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4103 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE,
4104 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4105 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4106 CKR_HOST_MEMORY, CKR_OK, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

4107 5.9.3 C_EncryptMessageBegin

```
4108 CK_DECLARE_FUNCTION(CK_RV, C_EncryptMessageBegin) (
```

```
4109     CK_SESSION_HANDLE hSession,  
4110     CK_VOID_PTR pParameter,  
4111     CK ULONG ulParameterLen,  
4112     CK_BYTE_PTR pAssociatedData,  
4113     CK ULONG ulAssociatedDataLen  
4114 );
```

4115 **C_EncryptMessageBegin** begins a multiple-part message encryption operation. *hSession* is the
4116 session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the
4117 message encryption operation; *pAssociatedData* and *ulAssociatedDataLen* specify the associated data
4118 for an AEAD mechanism.

4119 Typically, *pParameter* is an initialization vector (IV) or nonce. Depending on the mechanism parameter
4120 passed to **C_MessageEncryptInit**, *pParameter* may be either an input or an output parameter. For
4121 example, if the mechanism parameter specifies an IV generator mechanism, the IV generated by the IV
4122 generator will be output to the *pParameter* buffer.

4123 If the mechanism is not AEAD, *pAssociatedData* and *ulAssociatedDataLen* are not used and should be
4124 set to (NULL, 0).

4125 After calling **C_EncryptMessageBegin**, the application should call **C_EncryptMessageNext** one or
4126 more times to encrypt the message in multiple parts. The message encryption operation is active until the
4127 application uses a call to **C_EncryptMessageNext** with flags=CKF_END_OF_MESSAGE to actually
4128 obtain the final piece of ciphertext. To process additional messages (in single or multiple parts), the
4129 application MUST call **C_EncryptMessage** or **C_EncryptMessageBegin** again.

4130 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4131 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
4132 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE,
4133 CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
4134 CKR_USER_NOT_LOGGED_IN.

4135 5.9.4 C_EncryptMessageNext

```
4136 CK_DECLARE_FUNCTION(CK_RV, C_EncryptMessageNext) (  
4137     CK_SESSION_HANDLE hSession,  
4138     CK_VOID_PTR pParameter,  
4139     CK ULONG ulParameterLen,  
4140     CK_BYTE_PTR pPlaintextPart,  
4141     CK ULONG ulPlaintextPartLen,  
4142     CK_BYTE_PTR pCiphertextPart,  
4143     CK ULONG_PTR pulCiphertextPartLen,  
4144     CK_FLAGS flags  
4145 );
```

4146 **C_EncryptMessageNext** continues a multiple-part message encryption operation, processing another
4147 message part. *hSession* is the session's handle; *pParameter* and *ulParameterLen* specify any
4148 mechanism-specific parameters for the message encryption operation; *pPlaintextPart* points to the
4149 plaintext message part; *ulPlaintextPartLen* is the length of the plaintext message part; *pCiphertextPart*
4150 points to the location that receives the encrypted message part; *pulCiphertextPartLen* points to the
4151 location that holds the length in bytes of the encrypted message part; *flags* is set to 0 if there is more
4152 plaintext data to follow, or set to CKF_END_OF_MESSAGE if this is the last plaintext part.

4153 Typically, *pParameter* is an initialization vector (IV) or nonce. Depending on the mechanism parameter
4154 passed to **C_EncryptMessageNext**, *pParameter* may be either an input or an output parameter. For
4155 example, if the mechanism parameter specifies an IV generator mechanism, the IV generated by the IV
4156 generator will be output to the *pParameter* buffer.

4157 **C_EncryptMessageNext** uses the convention described in Section 5.2 on producing output.

4158 The message encryption operation MUST have been started with **C_EncryptMessageBegin**. This
4159 function may be called any number of times in succession. A call to **C_EncryptMessageNext** with flags=0
4160 which results in an error other than CKR_BUFFER_TOO_SMALL terminates the current message

4161 encryption operation. A call to **C_EncryptMessageNext** with flags=CKF_END_OF_MESSAGE always
4162 terminates the active message encryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a
4163 successful call (i.e., one which returns **CKR_OK**) to determine the length of the buffer needed to hold the
4164 ciphertext.

4165 Although the last **C_EncryptMessageNext** call ends the encryption of a message, it does not finish the
4166 message-based encryption process. Additional **C_EncryptMessage** or **C_EncryptMessageBegin** and
4167 **C_EncryptMessageNext** calls may be made on the session.

4168 The plaintext and ciphertext can be in the same place, i.e., it is OK if *pPlaintextPart* and *pCiphertextPart*
4169 point to the same location.

4170 For some multi-part encryption mechanisms, the input plaintext data has certain length constraints,
4171 because the mechanism's input data MUST consist of an integral number of blocks. If these constraints
4172 are not satisfied when the final message part is supplied (i.e., with flags=CKF_END_OF_MESSAGE),
4173 then **C_EncryptMessageNext** will fail with return code CKR_DATA_LEN_RANGE.

4174 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4175 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR,
4176 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED,
4177 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
4178 CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

4179 5.9.5 C_MessageEncryptFinal

```
4180 CK_DECLARE_FUNCTION(CK_RV, C_MessageEncryptFinal) (
4181     CK_SESSION_HANDLE hSession
4182 );
```

4183 **C_MessageEncryptFinal** finishes a message-based encryption process. *hSession* is the session's
4184 handle.

4185 The message-based encryption process MUST have been initialized with **C_MessageEncryptInit**.

4186 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4187 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4188 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4189 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
4190 CKR_SESSION_HANDLE_INVALID.

4191 Example:

```
4192 #define PLAINTEXT_BUF_SZ 200
4193 #define AUTH_BUF_SZ 100
4194 #define CIPHERTEXT_BUF_SZ 256
4195
4196 CK_SESSION_HANDLE hSession;
4197 CK_OBJECT_HANDLE hKey;
4198 CK_BYTE iv[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
4199 CK_BYTE tag[16];
4200 CK_GCM_MESSAGE_PARAMS gcmParams = {
4201     iv,
4202     sizeof(iv) * 8,
4203     0,
4204     CKG_NO_GENERATE,
4205     tag,
4206     sizeof(tag) * 8
```

```

4207    } ;
4208    CK_MECHANISM mechanism = {
4209        CKM_AES_GCM, &gcmParams, sizeof(gcmParams)
4210    } ;
4211    CK_BYTE data[2][PLAINTEXT_BUF_SZ];
4212    CK_BYTE auth[2][AUTH_BUF_SZ];
4213    CK_BYTE encryptedData[2][CIPHERTEXT_BUF_SZ];
4214    CK_ULONG ulEncryptedDataLen, ulFirstEncryptedDataLen;
4215    CK ULONG firstPieceLen = PLAINTEXT_BUF_SZ / 2;
4216
4217    /* error handling is omitted for better readability */
4218    .
4219    .
4220    C_MessageEncryptInit(hSession, &mechanism, hKey);
4221    /* encrypt message en bloc with given IV */
4222    ulEncryptedDataLen = sizeof(encryptedData[0]);
4223    C_EncryptMessage(hSession,
4224        &gcmParams, sizeof(gcmParams),
4225        &auth[0][0], sizeof(auth[0]),
4226        &data[0][0], sizeof(data[0]),
4227        &encryptedData[0][0], &ulEncryptedDataLen);
4228    /* iv and tag are set now for message */
4229
4230    /* encrypt message in two steps with generated IV */
4231    gcmParams.ivGenerator = CKG_GENERATE;
4232    C_EncryptMessageBegin(hSession,
4233        &gcmParams, sizeof(gcmParams),
4234        &auth[1][0], sizeof(auth[1])
4235    );
4236    /* encrypt first piece */
4237    ulFirstEncryptedDataLen = sizeof(encryptedData[1]);
4238    C_EncryptMessageNext(hSession,
4239        &gcmParams, sizeof(gcmParams),
4240        &data[1][0], firstPieceLen,
4241        &encryptedData[1][0], &ulFirstEncryptedDataLen,
4242        0
4243    );
4244    /* encrypt second piece */
4245    ulEncryptedDataLen = sizeof(encryptedData[1]) - ulFirstEncryptedDataLen;
4246    C_EncryptMessageNext(hSession,
4247        &gcmParams, sizeof(gcmParams),
4248        &data[1][firstPieceLen], sizeof(data[1])-firstPieceLen,
4249        &encryptedData[1][ulFirstEncryptedDataLen], &ulEncryptedDataLen,

```

```

4250     CKF_END_OF_MESSAGE
4251 );
4252 /* tag is set now for message */
4253
4254 /* finalize */
4255 C_MessageEncryptFinal(hSession);

```

4256 5.10 Decryption functions

4257 Cryptoki provides the following functions for decrypting data:

4258 5.10.1 C_DecryptInit

```

4259 CK_DECLARE_FUNCTION(CK_RV, C_DecryptInit) (
4260     CK_SESSION_HANDLE hSession,
4261     CK_MECHANISM_PTR pMechanism,
4262     CK_OBJECT_HANDLE hKey
4263 );

```

4264 **C_DecryptInit** initializes a decryption operation. *hSession* is the session's handle; *pMechanism* points to
4265 the decryption mechanism; *hKey* is the handle of the decryption key.

4266 The **CKA_DECRYPT** attribute of the decryption key, which indicates whether the key supports
4267 decryption, MUST be CK_TRUE.

4268 After calling **C_DecryptInit**, the application can either call **C_Decrypt** to decrypt data in a single part; or
4269 call **C_DecryptUpdate** zero or more times, followed by **C_DecryptFinal**, to decrypt data in multiple parts.
4270 The decryption operation is active until the application uses a call to **C_Decrypt** or **C_DecryptFinal** to
4271 actually obtain the final piece of plaintext. To process additional data (in single or multiple parts), the
4272 application MUST call **C_DecryptInit** again.

4273 **C_DecryptInit** can be called with *pMechanism* set to NULL_PTR to terminate an active decryption
4274 operation. If an active operation cannot be cancelled, CKR_OPERATION_CANCEL_FAILED must be
4275 returned.

4276 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4277 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4278 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4279 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,
4280 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
4281 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
4282 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
4283 CKR_OPERATION_CANCEL_FAILED.

4284 Example: see **C_DecryptFinal**.

4285 5.10.2 C_Decrypt

```

4286 CK_DECLARE_FUNCTION(CK_RV, C_Decrypt) (
4287     CK_SESSION_HANDLE hSession,
4288     CK_BYTE_PTR pEncryptedData,
4289     CK_ULONG ulEncryptedDataLen,
4290     CK_BYTE_PTR pData,
4291     CK_ULONG_PTR pulDataLen
4292 );

```

4293 **C_Decrypt** decrypts encrypted data in a single part. *hSession* is the session's handle; *pEncryptedData*
4294 points to the encrypted data; *ulEncryptedDataLen* is the length of the encrypted data; *pData* points to the

4295 location that receives the recovered data; *pulDataLen* points to the location that holds the length of the
4296 recovered data.
4297 **C_Decrypt** uses the convention described in Section 5.2 on producing output.
4298 The decryption operation MUST have been initialized with **C_DecryptInit**. A call to **C_Decrypt** always
4299 terminates the active decryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a
4300 successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the
4301 plaintext.
4302 **C_Decrypt** cannot be used to terminate a multi-part operation, and MUST be called after **C_DecryptInit**
4303 without intervening **C_DecryptUpdate** calls.
4304 The ciphertext and plaintext can be in the same place, *i.e.*, it is OK if *pEncryptedData* and *pData* point to
4305 the same location.
4306 If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either
4307 CKR_ENCRYPTED_DATA_INVALID or CKR_ENCRYPTED_DATA_LEN_RANGE may be returned.
4308 For most mechanisms, **C_Decrypt** is equivalent to a sequence of **C_DecryptUpdate** operations followed
4309 by **C_DecryptFinal**.
4310 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4311 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4312 CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID,
4313 CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
4314 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
4315 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.
4316 Example: see **C_DecryptFinal** for an example of similar functions.

4317 5.10.3 C_DecryptUpdate

```
4318 CK_DECLARE_FUNCTION(CK_RV, C_DecryptUpdate) (
4319     CK_SESSION_HANDLE hSession,
4320     CK_BYTE_PTR pEncryptedPart,
4321     CK_ULONG ulEncryptedPartLen,
4322     CK_BYTE_PTR pPart,
4323     CK_ULONG_PTR pulPartLen
4324 );
```

4325 **C_DecryptUpdate** continues a multiple-part decryption operation, processing another encrypted data
4326 part. *hSession* is the session's handle; *pEncryptedPart* points to the encrypted data part;
4327 *ulEncryptedPartLen* is the length of the encrypted data part; *pPart* points to the location that receives the
4328 recovered data part; *pulPartLen* points to the location that holds the length of the recovered data part.

4329 **C_DecryptUpdate** uses the convention described in Section 5.2 on producing output.
4330 The decryption operation MUST have been initialized with **C_DecryptInit**. This function may be called
4331 any number of times in succession. A call to **C_DecryptUpdate** which results in an error other than
4332 CKR_BUFFER_TOO_SMALL terminates the current decryption operation.

4333 The ciphertext and plaintext can be in the same place, *i.e.*, it is OK if *pEncryptedPart* and *pPart* point to
4334 the same location.

4335 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4336 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4337 CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID,
4338 CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
4339 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
4340 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

4341 Example: See **C_DecryptFinal**.

4342 **5.10.4 C_DecryptFinal**

```
4343 CK_DECLARE_FUNCTION(CK_RV, C_DecryptFinal) (
4344     CK_SESSION_HANDLE hSession,
4345     CK_BYTE_PTR pLastPart,
4346     CK ULONG_PTR pulLastPartLen
4347 );
```

4348 **C_DecryptFinal** finishes a multiple-part decryption operation. *hSession* is the session's handle;
4349 *pLastPart* points to the location that receives the last recovered data part, if any; *pulLastPartLen* points to
4350 the location that holds the length of the last recovered data part.

4351 **C_DecryptFinal** uses the convention described in Section 5.2 on producing output.

4352 The decryption operation MUST have been initialized with **C_DecryptInit**. A call to **C_DecryptFinal**
4353 always terminates the active decryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a
4354 successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the
4355 plaintext.

4356 If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either
4357 CKR_ENCRYPTED_DATA_INVALID or CKR_ENCRYPTED_DATA_LEN_RANGE may be returned.

4358 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4359 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4360 CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID,
4361 CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
4362 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
4363 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

4364 Example:

```
4365 #define CIPHERTEXT_BUF_SZ 256
4366 #define PLAINTEXT_BUF_SZ 256
4367
4368 CK ULONG firstEncryptedPieceLen, secondEncryptedPieceLen;
4369 CK SESSION_HANDLE hSession;
4370 CK OBJECT_HANDLE hKey;
4371 CK BYTE iv[8];
4372 CK_MECHANISM mechanism = {
4373     CKM DES CBC PAD, iv, sizeof(iv)
4374 };
4375 CK BYTE data[PLAINTEXT_BUF_SZ];
4376 CK BYTE encryptedData[CIPHERTEXT_BUF_SZ];
4377 CK ULONG ulData1Len, ulData2Len, ulData3Len;
4378 CK RV rv;
4379 .
4380 .
4381 .
4382 firstEncryptedPieceLen = 90;
4383 secondEncryptedPieceLen = CIPHERTEXT_BUF_SZ - firstEncryptedPieceLen;
4384 rv = C_DecryptInit(hSession, &mechanism, hKey);
4385 if (rv == CKR_OK) {
4386     /* Decrypt first piece */
4387     ulData1Len = sizeof(data);
```

```

4388     rv = C_DecryptUpdate(
4389         hSession,
4390         &encryptedData[0], firstEncryptedPieceLen,
4391         &data[0], &ulData1Len);
4392     if (rv != CKR_OK) {
4393         .
4394         .
4395     }
4396
4397     /* Decrypt second piece */
4398     ulData2Len = sizeof(data)-ulData1Len;
4399     rv = C_DecryptUpdate(
4400         hSession,
4401         &encryptedData[firstEncryptedPieceLen],
4402         secondEncryptedPieceLen,
4403         &data[ulData1Len], &ulData2Len);
4404     if (rv != CKR_OK) {
4405         .
4406         .
4407     }
4408
4409     /* Get last little decrypted bit */
4410     ulData3Len = sizeof(data)-ulData1Len-ulData2Len;
4411     rv = C_DecryptFinal(
4412         hSession,
4413         &data[ulData1Len+ulData2Len], &ulData3Len);
4414     if (rv != CKR_OK) {
4415         .
4416         .
4417     }
4418 }
```

4419 5.11 Message-based decryption functions

4420 Message-based decryption refers to the process of decrypting multiple encrypted messages using the
 4421 same decryption mechanism and decryption key. The decryption mechanism can be either an
 4422 authenticated encryption with associated data (AEAD) algorithm or a pure encryption algorithm.
 4423 Cryptoki provides the following functions for message-based decryption.

4424 5.11.1 C_MessageDecryptInit

```

4425 CK_DECLARE_FUNCTION(CK_RV, C_MessageDecryptInit) (
4426     CK_SESSION_HANDLE hSession,
4427     CK_MECHANISM_PTR pMechanism,
4428     CK_OBJECT_HANDLE hKey
4429 );
```

4430 **C_MessageDecryptInit** initializes a message-based decryption process, preparing a session for one or
 4431 more decryption operations that use the same decryption mechanism and decryption key. *hSession* is
 4432 the session's handle; *pMechanism* points to the decryption mechanism; *hKey* is the handle of the
 4433 decryption key.
 4434 The CKA_DECRYPT attribute of the decryption key, which indicates whether the key supports decryption,
 4435 MUST be CK_TRUE.
 4436 After calling **C_MessageDecryptInit**, the application can either call **C_DecryptMessage** to decrypt an
 4437 encrypted message in a single part; or call **C_DecryptMessageBegin**, followed by
 4438 **C_DecryptMessageNext** one or more times, to decrypt an encrypted message in multiple parts. This
 4439 may be repeated several times. The message-based decryption process is active until the application
 4440 uses a call to **C_MessageDecryptFinal** to finish the message-based decryption process.
 4441 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
 4442 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
 4443 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
 4444 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,
 4445 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
 4446 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
 4447 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
 4448 CKR_OPERATION_CANCEL_FAILED.

4449 5.11.2 C_DecryptMessage

```

4450 CK_DECLARE_FUNCTION(CK_RV, C_DecryptMessage) (
4451   CK_SESSION_HANDLE hSession,
4452   CK_VOID_PTR pParameter,
4453   CK ULONG ulParameterLen,
4454   CK_BYTE_PTR pAssociatedData,
4455   CK ULONG ulAssociatedDataLen,
4456   CK_BYTE_PTR pCiphertext,
4457   CK ULONG ulCiphertextLen,
4458   CK_BYTE_PTR pPlaintext,
4459   CK ULONG_PTR pulPlaintextLen
4460 );

```

4461 **C_DecryptMessage** decrypts an encrypted message in a single part. *hSession* is the session's handle;
 4462 *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message decryption
 4463 operation; *pAssociatedData* and *ulAssociatedDataLen* specify the associated data for an AEAD
 4464 mechanism; *pCiphertext* points to the encrypted message; *ulCiphertextLen* is the length of the encrypted
 4465 message; *pPlaintext* points to the location that receives the recovered message; *pulPlaintextLen* points to
 4466 the location that holds the length of the recovered message.

4467 Typically, *pParameter* is an initialization vector (IV) or nonce. Unlike the *pParameter* parameter of
 4468 **C_EncryptMessage**, *pParameter* is always an input parameter.

4469 If the decryption mechanism is not AEAD, *pAssociatedData* and *ulAssociatedDataLen* are not used and
 4470 should be set to (NULL, 0).

4471 **C_DecryptMessage** uses the convention described in Section 5.2 on producing output.

4472 The message-based decryption process MUST have been initialized with **C_MessageDecryptInit**. A call
 4473 to **C_DecryptMessage** begins and terminates a message decryption operation.

4474 **C_DecryptMessage** cannot be called in the middle of a multi-part message decryption operation.

4475 The ciphertext and plaintext can be in the same place, i.e., it is OK if *pCiphertext* and *pPlaintext* point to
 4476 the same location.

4477 If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either
 4478 CKR_ENCRYPTED_DATA_INVALID or CKR_ENCRYPTED_DATA_LEN_RANGE may be returned.

4479 If the decryption mechanism is an AEAD algorithm and the authenticity of the associated data or
 4480 ciphertext cannot be verified, then CKR_AEAD_DECRYPT_FAILED is returned.

4481 For most mechanisms, **C_DecryptMessage** is equivalent to **C_DecryptMessageBegin** followed by a
4482 sequence of **C_DecryptMessageNext** operations.
4483 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4484 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4485 CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID,
4486 CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_AEAD_DECRYPT_FAILED,
4487 CKR_FUNCTION_CANCELED, CKR_FUNCTION FAILED, CKR_GENERAL_ERROR,
4488 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
4489 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
4490 CKR_OPERATION_CANCEL_FAILED.

4491 5.11.3 C_DecryptMessageBegin

```
4492 CK_DECLARE_FUNCTION(CK_RV, C_DecryptMessageBegin) (
4493     CK_SESSION_HANDLE hSession,
4494     CK_VOID_PTR pParameter,
4495     CK ULONG ulParameterLen,
4496     CK_BYTE_PTR pAssociatedData,
4497     CK ULONG ulAssociatedDataLen
4498 );
```

4499 **C_DecryptMessageBegin** begins a multiple-part message decryption operation. *hSession* is the
4500 session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the
4501 message decryption operation; *pAssociatedData* and *ulAssociatedDataLen* specify the associated data
4502 for an AEAD mechanism.

4503 Typically, *pParameter* is an initialization vector (IV) or nonce. Unlike the *pParameter* parameter of
4504 **C_EncryptMessageBegin**, *pParameter* is always an input parameter.

4505 If the decryption mechanism is not AEAD, *pAssociatedData* and *ulAssociatedDataLen* are not used and
4506 should be set to (NULL, 0).

4507 After calling **C_DecryptMessageBegin**, the application should call **C_DecryptMessageNext** one or
4508 more times to decrypt the encrypted message in multiple parts. The message decryption operation is
4509 active until the application uses a call to **C_DecryptMessageNext** with flags=CKF_END_OF_MESSAGE
4510 to actually obtain the final piece of plaintext. To process additional encrypted messages (in single or
4511 multiple parts), the application MUST call **C_DecryptMessage** or **C_DecryptMessageBegin** again.

4512 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4513 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4514 CKR_FUNCTION_CANCELED, CKR_FUNCTION FAILED, CKR_GENERAL_ERROR,
4515 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
4516 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

4517 5.11.4 C_DecryptMessageNext

```
4518 CK_DECLARE_FUNCTION(CK_RV, C_DecryptMessageNext) (
4519     CK_SESSION_HANDLE hSession,
4520     CK_VOID_PTR pParameter,
4521     CK ULONG ulParameterLen,
4522     CK_BYTE_PTR pCiphertextPart,
4523     CK ULONG ulCiphertextPartLen,
4524     CK_BYTE_PTR pPlaintextPart,
4525     CK ULONG_PTR pulPlaintextPartLen,
4526     CK_FLAGS flags
4527 );
```

4528 **C_DecryptMessageNext** continues a multiple-part message decryption operation, processing another
4529 encrypted message part. *hSession* is the session's handle; *pParameter* and *ulParameterLen* specify any
4530 mechanism-specific parameters for the message decryption operation; *pCiphertextPart* points to the

4531 encrypted message part; *ulCiphertextPartLen* is the length of the encrypted message part; *pPlaintextPart*
 4532 points to the location that receives the recovered message part; *pulPlaintextPartLen* points to the location
 4533 that holds the length of the recovered message part; flags is set to 0 if there is more ciphertext data to
 4534 follow, or set to CKF_END_OF_MESSAGE if this is the last ciphertext part.
 4535 Typically, *pParameter* is an initialization vector (IV) or nonce. Unlike the *pParameter* parameter of
 4536 **C_EncryptMessageNext**, *pParameter* is always an input parameter.
 4537 **C_DecryptMessageNext** uses the convention described in Section 5.2 on producing output.
 4538 The message decryption operation MUST have been started with **C_DecryptMessageBegin**. This
 4539 function may be called any number of times in succession. A call to **C_DecryptMessageNext** with
 4540 flags=0 which results in an error other than CKR_BUFFER_TOO_SMALL terminates the current message
 4541 decryption operation. A call to **C_DecryptMessageNext** with flags=CKF_END_OF_MESSAGE always
 4542 terminates the active message decryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a
 4543 successful call (i.e., one which returns CKR_OK) to determine the length of the buffer needed to hold the
 4544 plaintext.
 4545 The ciphertext and plaintext can be in the same place, i.e., it is OK if *pCiphertextPart* and *pPlaintextPart*
 4546 point to the same location.
 4547 Although the last **C_DecryptMessageNext** call ends the decryption of a message, it does not finish the
 4548 message-based decryption process. Additional **C_DecryptMessage** or **C_DecryptMessageBegin** and
 4549 **C_DecryptMessageNext** calls may be made on the session.
 4550 If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either
 4551 CKR_ENCRYPTED_DATA_INVALID or CKR_ENCRYPTED_DATA_LEN_RANGE may be returned by
 4552 the last **C_DecryptMessageNext** call.
 4553 If the decryption mechanism is an AEAD algorithm and the authenticity of the associated data or
 4554 ciphertext cannot be verified, then CKR_AEAD_DECRYPT_FAILED is returned by the last
 4555 **C_DecryptMessageNext** call.
 4556 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
 4557 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
 4558 CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID,
 4559 CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_AEAD_DECRYPT_FAILED,
 4560 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
 4561 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
 4562 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

4563 **5.11.5 C_MessageDecryptFinal**

```

4564 CK_DECLARE_FUNCTION(CK_RV, C_MessageDecryptFinal) (
4565   CK_SESSION_HANDLE hSession
4566 );

```

4567 **C_MessageDecryptFinal** finishes a message-based decryption process. *hSession* is the session's
 4568 handle.
 4569 The message-based decryption process MUST have been initialized with **C_MessageDecryptInit**.
 4570 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
 4571 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
 4572 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
 4573 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
 4574 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

4575 **5.12 Message digesting functions**

4576 Cryptoki provides the following functions for digesting data:

4577 **5.12.1 C_DigestInit**

```
4578 CK_DECLARE_FUNCTION(CK_RV, C_DigestInit) (
4579     CK_SESSION_HANDLE hSession,
4580     CK_MECHANISM_PTR pMechanism
4581 );
```

4582 **C_DigestInit** initializes a message-digesting operation. *hSession* is the session's handle; *pMechanism* points to the digesting mechanism.

4584 After calling **C_DigestInit**, the application can either call **C_Digest** to digest data in a single part; or call
4585 **C_DigestUpdate** zero or more times, followed by **C_DigestFinal**, to digest data in multiple parts. The
4586 message-digesting operation is active until the application uses a call to **C_Digest** or **C_DigestFinal** to
4587 actually obtain the message digest. To process additional data (in single or multiple parts), the
4588 application MUST call **C_DigestInit** again.

4589 **C_DigestInit** can be called with *pMechanism* set to NULL_PTR to terminate an active message-digesting
4590 operation. If an operation has been initialized and it cannot be cancelled,
4591 CKR_OPERATION_CANCEL_FAILED must be returned.

4592 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4593 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4594 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4595 CKR_HOST_MEMORY, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID,
4596 CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
4597 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
4598 CKR_OPERATION_CANCEL_FAILED.

4599 Example: see **C_DigestFinal**.

4600 **5.12.2 C_Digest**

```
4601 CK_DECLARE_FUNCTION(CK_RV, C_Digest) (
4602     CK_SESSION_HANDLE hSession,
4603     CK_BYTE_PTR pData,
4604     CK_ULONG ulDataLen,
4605     CK_BYTE_PTR pDigest,
4606     CK_ULONG_PTR pulDigestLen
4607 );
```

4608 **C_Digest** digests data in a single part. *hSession* is the session's handle, *pData* points to the data;
4609 *ulDataLen* is the length of the data; *pDigest* points to the location that receives the message digest;
4610 *pulDigestLen* points to the location that holds the length of the message digest.

4611 **C_Digest** uses the convention described in Section 5.2 on producing output.

4612 The digest operation MUST have been initialized with **C_DigestInit**. A call to **C_Digest** always
4613 terminates the active digest operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful
4614 call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the message
4615 digest.

4616 **C_Digest** cannot be used to terminate a multi-part operation, and MUST be called after **C_DigestInit**
4617 without intervening **C_DigestUpdate** calls.

4618 The input data and digest output can be in the same place, *i.e.*, it is OK if *pData* and *pDigest* point to the
4619 same location.

4620 **C_Digest** is equivalent to a sequence of **C_DigestUpdate** operations followed by **C_DigestFinal**.

4621 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4622 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4623 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
4624 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
4625 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

4626 Example: see **C_DigestFinal** for an example of similar functions.

4627 5.12.3 C_DigestUpdate

```
4628 CK_DECLARE_FUNCTION(CK_RV, C_DigestUpdate) (
4629     CK_SESSION_HANDLE hSession,
4630     CK_BYTE_PTR pPart,
4631     CK ULONG ulPartLen
4632 );
```

4633 **C_DigestUpdate** continues a multiple-part message-digesting operation, processing another data part.
4634 *hSession* is the session's handle, *pPart* points to the data part; *ulPartLen* is the length of the data part.

4635 The message-digesting operation MUST have been initialized with **C_DigestInit**. Calls to this function
4636 and **C_DigestKey** may be interspersed any number of times in any order. A call to **C_DigestUpdate**
4637 which results in an error terminates the current digest operation.

4638 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4639 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4640 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4641 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
4642 CKR_SESSION_HANDLE_INVALID.

4643 Example: see **C_DigestFinal**.

4644 5.12.4 C_DigestKey

```
4645 CK_DECLARE_FUNCTION(CK_RV, C_DigestKey) (
4646     CK_SESSION_HANDLE hSession,
4647     CK_OBJECT_HANDLE hKey
4648 );
```

4649 **C_DigestKey** continues a multiple-part message-digesting operation by digesting the value of a secret
4650 key. *hSession* is the session's handle; *hKey* is the handle of the secret key to be digested.

4651 The message-digesting operation MUST have been initialized with **C_DigestInit**. Calls to this function
4652 and **C_DigestUpdate** may be interspersed any number of times in any order.

4653 If the value of the supplied key cannot be digested purely for some reason related to its length,
4654 **C_DigestKey** should return the error code CKR_KEY_SIZE_RANGE.

4655 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4656 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
4657 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_HANDLE_INVALID,
4658 CKR_KEY_INDIGESTIBLE, CKR_KEY_SIZE_RANGE, CKR_OK,
4659 CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

4660 Example: see **C_DigestFinal**.

4661 5.12.5 C_DigestFinal

```
4662 CK_DECLARE_FUNCTION(CK_RV, C_DigestFinal) (
4663     CK_SESSION_HANDLE hSession,
4664     CK_BYTE_PTR pDigest,
4665     CK_ULONG_PTR pulDigestLen
4666 );
```

4667 **C_DigestFinal** finishes a multiple-part message-digesting operation, returning the message digest.
4668 *hSession* is the session's handle; *pDigest* points to the location that receives the message digest;
4669 *pulDigestLen* points to the location that holds the length of the message digest.

4670 **C_DigestFinal** uses the convention described in Section 5.2 on producing output.

4671 The digest operation MUST have been initialized with **C_DigestInit**. A call to **C_DigestFinal** always
4672 terminates the active digest operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful

4673 call (i.e., one which returns CKR_OK) to determine the length of the buffer needed to hold the message
4674 digest.

4675 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4676 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4677 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
4678 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
4679 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

4680 Example:

```
4681 CK_SESSION_HANDLE hSession;
4682 CK_OBJECT_HANDLE hKey;
4683 CK_MECHANISM mechanism = {
4684     CKM_MD5, NULL_PTR, 0
4685 };
4686 CK_BYTE data[] = {...};
4687 CK_BYTE digest[16];
4688 CK_ULONG ulDigestLen;
4689 CK_RV rv;
4690
4691 .
4692 .
4693 rv = C_DigestInit(hSession, &mechanism);
4694 if (rv != CKR_OK) {
4695     .
4696     .
4697 }
4698
4699 rv = C_DigestUpdate(hSession, data, sizeof(data));
4700 if (rv != CKR_OK) {
4701     .
4702     .
4703 }
4704
4705 rv = C_DigestKey(hSession, hKey);
4706 if (rv != CKR_OK) {
4707     .
4708     .
4709 }
4710
4711 ulDigestLen = sizeof(digest);
4712 rv = C_DigestFinal(hSession, digest, &ulDigestLen);
4713 .
4714 .
```

4715 **5.13 Signing and MACing functions**

4716 Cryptoki provides the following functions for signing data (for the purposes of Cryptoki, these operations
4717 also encompass message authentication codes).

4718 **5.13.1 C_SignInit**

```
4719 CK_DECLARE_FUNCTION(CK_RV, C_SignInit) (
4720     CK_SESSION_HANDLE hSession,
4721     CK_MECHANISM_PTR pMechanism,
4722     CK_OBJECT_HANDLE hKey
4723 );
```

4724 **C_SignInit** initializes a signature operation, where the signature is an appendix to the data. *hSession* is
4725 the session's handle; *pMechanism* points to the signature mechanism; *hKey* is the handle of the signature
4726 key.

4727 The **CKA_SIGN** attribute of the signature key, which indicates whether the key supports signatures with
4728 appendix, MUST be CK_TRUE.

4729 After calling **C_SignInit**, the application can either call **C_Sign** to sign in a single part; or call
4730 **C_SignUpdate** one or more times, followed by **C_SignFinal**, to sign data in multiple parts. The signature
4731 operation is active until the application uses a call to **C_Sign** or **C_SignFinal** to actually obtain the
4732 signature. To process additional data (in single or multiple parts), the application MUST call **C_SignInit**
4733 again.

4734 **C_SignInit** can be called with *pMechanism* set to NULL_PTR to terminate an active signature operation.
4735 If an operation has been initialized and it cannot be cancelled, CKR_OPERATION_CANCEL_FAILED
4736 must be returned.

4737 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4738 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4739 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4740 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,
4741 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
4742 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
4743 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
4744 CKR_OPERATION_CANCEL_FAILED.

4745 Example: see **C_SignFinal**.

4746 **5.13.2 C_Sign**

```
4747 CK_DECLARE_FUNCTION(CK_RV, C_Sign) (
4748     CK_SESSION_HANDLE hSession,
4749     CK_BYTE_PTR pData,
4750     CK_ULONG ulDataLen,
4751     CK_BYTE_PTR pSignature,
4752     CK_ULONG_PTR pulSignatureLen
4753 );
```

4754 **C_Sign** signs data in a single part, where the signature is an appendix to the data. *hSession* is the
4755 session's handle; *pData* points to the data; *ulDataLen* is the length of the data; *pSignature* points to the
4756 location that receives the signature; *pulSignatureLen* points to the location that holds the length of the
4757 signature.

4758 **C_Sign** uses the convention described in Section 5.2 on producing output.

4759 The signing operation MUST have been initialized with **C_SignInit**. A call to **C_Sign** always terminates
4760 the active signing operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call (i.e.,
4761 one which returns CKR_OK) to determine the length of the buffer needed to hold the signature.

4762 **C_Sign** cannot be used to terminate a multi-part operation, and MUST be called after **C_SignInit** without
4763 intervening **C_SignUpdate** calls.

4764 For most mechanisms, **C_Sign** is equivalent to a sequence of **C_SignUpdate** operations followed by
4765 **C_SignFinal**.

4766 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4767 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE,
4768 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4769 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4770 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
4771 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_FUNCTION_REJECTED,
4772 CKR_TOKEN_RESOURCE_EXCEEDED.

4773 Example: see **C_SignFinal** for an example of similar functions.

4774 5.13.3 C_SignUpdate

```
4775 CK_DECLARE_FUNCTION(CK_RV, C_SignUpdate) (
4776     CK_SESSION_HANDLE hSession,
4777     CK_BYTE_PTR pPart,
4778     CK_ULONG ulPartLen
4779 );
```

4780 **C_SignUpdate** continues a multiple-part signature operation, processing another data part. *hSession* is
4781 the session's handle, *pPart* points to the data part; *ulPartLen* is the length of the data part.

4782 The signature operation MUST have been initialized with **C_SignInit**. This function may be called any
4783 number of times in succession. A call to **C_SignUpdate** which results in an error terminates the current
4784 signature operation.

4785 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4786 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4787 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
4788 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
4789 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
4790 CKR_TOKEN_RESOURCE_EXCEEDED.

4791 Example: see **C_SignFinal**.

4792 5.13.4 C_SignFinal

```
4793 CK_DECLARE_FUNCTION(CK_RV, C_SignFinal) (
4794     CK_SESSION_HANDLE hSession,
4795     CK_BYTE_PTR pSignature,
4796     CK_ULONG_PTR pulSignatureLen
4797 );
```

4798 **C_SignFinal** finishes a multiple-part signature operation, returning the signature. *hSession* is the
4799 session's handle; *pSignature* points to the location that receives the signature; *pulSignatureLen* points to
4800 the location that holds the length of the signature.

4801 **C_SignFinal** uses the convention described in Section 5.2 on producing output.

4802 The signing operation MUST have been initialized with **C_SignInit**. A call to **C_SignFinal** always
4803 terminates the active signing operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful
4804 call (i.e., one which returns CKR_OK) to determine the length of the buffer needed to hold the signature.

4805 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4806 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR,
4807 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED,
4808 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
4809 CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,

```
4810 CKR_USER_NOT_LOGGED_IN, CKR_FUNCTION_REJECTED,  
4811 CKR_TOKEN_RESOURCE_EXCEEDED.
```

```
4812 Example:
```

```
4813 CK_SESSION_HANDLE hSession;  
4814 CK_OBJECT_HANDLE hKey;  
4815 CK_MECHANISM mechanism = {  
4816     CKM_DES_MAC, NULL_PTR, 0  
4817 };  
4818 CK_BYTE data[] = {...};  
4819 CK_BYTE mac[4];  
4820 CK ULONG ulMacLen;  
4821 CK_RV rv;  
4822 .  
4823 .  
4825 rv = C_SignInit(hSession, &mechanism, hKey);  
4826 if (rv == CKR_OK) {  
4827     rv = C_SignUpdate(hSession, data, sizeof(data));  
4828     .  
4829     .  
4830     ulMacLen = sizeof(mac);  
4831     rv = C_SignFinal(hSession, mac, &ulMacLen);  
4832     .  
4833     .  
4834 }
```

4835 5.13.5 C_SignRecoverInit

```
4836 CK_DECLARE_FUNCTION(CK_RV, C_SignRecoverInit) (  
4837     CK_SESSION_HANDLE hSession,  
4838     CK_MECHANISM_PTR pMechanism,  
4839     CK_OBJECT_HANDLE hKey  
4840 );
```

4841 **C_SignRecoverInit** initializes a signature operation, where the data can be recovered from the signature.
4842 *hSession* is the session's handle; *pMechanism* points to the structure that specifies the signature
4843 mechanism; *hKey* is the handle of the signature key.

4844 The **CKA_SIGN_RECOVER** attribute of the signature key, which indicates whether the key supports
4845 signatures where the data can be recovered from the signature, MUST be CK_TRUE.

4846 After calling **C_SignRecoverInit**, the application may call **C_SignRecover** to sign in a single part. The
4847 signature operation is active until the application uses a call to **C_SignRecover** to actually obtain the
4848 signature. To process additional data in a single part, the application MUST call **C_SignRecoverInit**
4849 again.

4850 **C_SignRecoverInit** can be called with *pMechanism* set to NULL_PTR to terminate an active signature
4851 with data recovery operation. If an active operation has been initialized and it cannot be cancelled,
4852 CKR_OPERATION_CANCEL_FAILED must be returned.

4853 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4854 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,

```
4855 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,  
4856 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,  
4857 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,  
4858 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,  
4859 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,  
4860 CKR_OPERATION_CANCEL FAILED.
```

4861 Example: see **C_SignRecover**.

4862 5.13.6 C_SignRecover

```
4863 CK_DECLARE_FUNCTION(CK_RV, C_SignRecover) (  
4864     CK_SESSION_HANDLE hSession,  
4865     CK_BYTE_PTR pData,  
4866     CK_ULONG ulDataLen,  
4867     CK_BYTE_PTR pSignature,  
4868     CK_ULONG_PTR pulSignatureLen  
4869 );
```

4870 **C_SignRecover** signs data in a single operation, where the data can be recovered from the signature.
4871 *hSession* is the session's handle; *pData* points to the data; *ulDataLen* is the length of the data;
4872 *pSignature* points to the location that receives the signature; *pulSignatureLen* points to the location that
4873 holds the length of the signature.

4874 **C_SignRecover** uses the convention described in Section 5.2 on producing output.

4875 The signing operation MUST have been initialized with **C_SignRecoverInit**. A call to **C_SignRecover**
4876 always terminates the active signing operation unless it returns CKR_BUFFER_TOO_SMALL or is a
4877 successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the
4878 signature.

4879 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4880 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE,
4881 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4882 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4883 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
4884 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
4885 CKR_TOKEN_RESOURCE_EXCEEDED.

4886 Example:

```
4887 CK_SESSION_HANDLE hSession;  
4888 CK_OBJECT_HANDLE hKey;  
4889 CK_MECHANISM mechanism = {  
4890     CKM_RSA_9796, NULL_PTR, 0  
4891 };  
4892 CK_BYTE data[] = {...};  
4893 CK_BYTE signature[128];  
4894 CK_ULONG ulSignatureLen;  
4895 CK_RV rv;  
4896  
4897 .  
4898 .  
4899 rv = C_SignRecoverInit(hSession, &mechanism, hKey);  
4900 if (rv == CKR_OK) {  
    ulSignatureLen = sizeof(signature);
```

```

4902     rv = C_SignRecover(
4903         hSession, data, sizeof(data), signature, &ulSignatureLen);
4904     if (rv == CKR_OK) {
4905         .
4906         .
4907     }
4908 }
4909

```

4910 5.14 Message-based signing and MACing functions

4911 Message-based signature refers to the process of signing multiple messages using the same signature
 4912 mechanism and signature key.

4913 Cryptoki provides the following functions for signing messages (for the purposes of Cryptoki, these
 4914 operations also encompass message authentication codes).

4915 5.14.1 C_MessageSignInit

```

4916 CK_DECLARE_FUNCTION(CK_RV, C_MessageSignInit) (
4917     CK_SESSION_HANDLE hSession,
4918     CK_MECHANISM_PTR pMechanism,
4919     CK_OBJECT_HANDLE hKey
4920 );

```

4921 **C_MessageSignInit** initializes a message-based signature process, preparing a session for one or more
 4922 signature operations (where the signature is an appendix to the data) that use the same signature
 4923 mechanism and signature key. *hSession* is the session's handle; *pMechanism* points to the signature
 4924 mechanism; *hKey* is the handle of the signature key.

4925 The **CKA_SIGN** attribute of the signature key, which indicates whether the key supports signatures with
 4926 appendix, MUST be CK_TRUE.

4927 After calling **C_MessageSignInit**, the application can either call **C_SignMessage** to sign a message in a
 4928 single part; or call **C_SignMessageBegin**, followed by **C_SignMessageNext** one or more times, to sign
 4929 a message in multiple parts. This may be repeated several times. The message-based signature process
 4930 is active until the application calls **C_MessageSignFinal** to finish the message-based signature process.

4931 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
 4932 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
 4933 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
 4934 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,
 4935 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
 4936 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
 4937 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

4938 5.14.2 C_SignMessage

```

4939 CK_DECLARE_FUNCTION(CK_RV, C_SignMessage) (
4940     CK_SESSION_HANDLE hSession,
4941     CK_VOID_PTR pParameter,
4942     CK_ULONG ulParameterLen,
4943     CK_BYTE_PTR pData,
4944     CK_ULONG ulDataLen,

```

```
4945     CK_BYTE_PTR pSignature,  
4946     CK ULONG_PTR pulSignatureLen  
4947 );
```

4948 **C_SignMessage** signs a message in a single part, where the signature is an appendix to the message.
4949 **C_MessageSignInit** must previously been called on the session. *hSession* is the session's handle;
4950 *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message signature
4951 operation; *pData* points to the data; *ulDataLen* is the length of the data; *pSignature* points to the location
4952 that receives the signature; *pulSignatureLen* points to the location that holds the length of the signature.
4953 Depending on the mechanism parameter passed to **C_MessageSignInit**, *pParameter* may be either an
4954 input or an output parameter.

4955 **C_SignMessage** uses the convention described in Section 5.2 on producing output.

4956 The message-based signing process MUST have been initialized with **C_MessageSignInit**. A call to
4957 **C_SignMessage** begins and terminates a message signing operation unless it returns
4958 CKR_BUFFER_TOO_SMALL to determine the length of the buffer needed to hold the signature, or is a
4959 successful call (i.e., one which returns CKR_OK).

4960 **C_SignMessage** cannot be called in the middle of a multi-part message signing operation.

4961 **C_SignMessage** does not finish the message-based signing process. Additional **C_SignMessage** or
4962 **C_SignMessageBegin** and **C_SignMessageNext** calls may be made on the session.

4963 For most mechanisms, **C_SignMessage** is equivalent to **C_SignMessageBegin** followed by a sequence
4964 of **C_SignMessageNext** operations.

4965 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4966 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE,
4967 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4968 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4969 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
4970 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_FUNCTION_REJECTED,
4971 CKR_TOKEN_RESOURCE_EXCEEDED.

4972 5.14.3 C_SignMessageBegin

```
4973 CK_DECLARE_FUNCTION(CK_RV, C_SignMessageBegin) (  
4974     CK_SESSION_HANDLE hSession,  
4975     CK_VOID_PTR pParameter,  
4976     CK ULONG ulParameterLen  
4977 );
```

4978 **C_SignMessageBegin** begins a multiple-part message signature operation, where the signature is an
4979 appendix to the message. **C_MessageSignInit** must previously been called on the session. *hSession* is
4980 the session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for
4981 the message signature operation.

4982 Depending on the mechanism parameter passed to **C_MessageSignInit**, *pParameter* may be either an
4983 input or an output parameter.

4984 After calling **C_SignMessageBegin**, the application should call **C_SignMessageNext** one or more times
4985 to sign the message in multiple parts. The message signature operation is active until the application
4986 uses a call to **C_SignMessageNext** with a non-NULL *pulSignatureLen* to actually obtain the signature.
4987 To process additional messages (in single or multiple parts), the application MUST call **C_SignMessage**
4988 or **C_SignMessageBegin** again.

4989 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4990 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4991 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4992 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,

4993 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
4994 CKR_TOKEN_RESOURCE_EXCEEDED.

4995 5.14.4 C_SignMessageNext

```
4996 CK_DECLARE_FUNCTION(CK_RV, C_SignMessageNext) (
4997     CK_SESSION_HANDLE hSession,
4998     CK_VOID_PTR pParameter,
4999     CK ULONG ulParameterLen,
5000     CK_BYTE_PTR pDataPart,
5001     CK ULONG ulDataPartLen,
5002     CK_BYTE_PTR pSignature,
5003     CK ULONG_PTR pulSignatureLen
5004 );
```

5005 **C_SignMessageNext** continues a multiple-part message signature operation, processing another data
5006 part, or finishes a multiple-part message signature operation, returning the signature. *hSession* is the
5007 session's handle, *pDataPart* points to the data part; *pParameter* and *ulParameterLen* specify any
5008 mechanism-specific parameters for the message signature operation; *ulDataPartLen* is the length of the
5009 data part; *pSignature* points to the location that receives the signature; *pulSignatureLen* points to the
5010 location that holds the length of the signature.

5011 The *pulSignatureLen* argument is set to NULL if there is more data part to follow, or set to a non-NUL
5012 value (to receive the signature length) if this is the last data part.

5013 **C_SignMessageNext** uses the convention described in Section 5.2 on producing output.

5014 The message signing operation MUST have been started with **C_SignMessageBegin**. This function may
5015 be called any number of times in succession. A call to **C_SignMessageNext** with a NULL
5016 *pulSignatureLen* which results in an error terminates the current message signature operation. A call to
5017 **C_SignMessageNext** with a non-NULL *pulSignatureLen* always terminates the active message signing
5018 operation unless it returns CKR_BUFFER_TOO_SMALL to determine the length of the buffer needed to
5019 hold the signature, or is a successful call (i.e., one which returns CKR_OK).

5020 Although the last **C_SignMessageNext** call ends the signing of a message, it does not finish the
5021 message-based signing process. Additional **C_SignMessage** or **C_SignMessageBegin** and
5022 **C_SignMessageNext** calls may be made on the session.

5023 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
5024 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR,
5025 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED,
5026 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
5027 CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
5028 CKR_USER_NOT_LOGGED_IN, CKR_FUNCTION_REJECTED,
5029 CKR_TOKEN_RESOURCE_EXCEEDED.

5030 5.14.5 C_MessageSignFinal

```
5031 CK_DECLARE_FUNCTION(CK_RV, C_MessageSignFinal) (
5032     CK_SESSION_HANDLE hSession
5033 );
```

5034 **C_MessageSignFinal** finishes a message-based signing process. *hSession* is the session's handle.

5035 The message-based signing process MUST have been initialized with **C_MessageSignInit**.

5036 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5037 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
5038 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,

5039 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
5040 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_FUNCTION_REJECTED,
5041 CKR_TOKEN_RESOURCE_EXCEEDED.

5042 **5.15 Functions for verifying signatures and MACs**

5043 Cryptoki provides the following functions for verifying signatures on data (for the purposes of Cryptoki,
5044 these operations also encompass message authentication codes):

5045 **5.15.1 C_VerifyInit**

```
5046 CK_DECLARE_FUNCTION(CK_RV, C_VerifyInit) (
5047     CK_SESSION_HANDLE hSession,
5048     CK_MECHANISM_PTR pMechanism,
5049     CK_OBJECT_HANDLE hKey
5050 );
```

5051 **C_VerifyInit** initializes a verification operation, where the signature is an appendix to the data. *hSession*
5052 is the session's handle; *pMechanism* points to the structure that specifies the verification mechanism;
5053 *hKey* is the handle of the verification key.

5054 The **CKA_VERIFY** attribute of the verification key, which indicates whether the key supports verification
5055 where the signature is an appendix to the data, MUST be CK_TRUE.

5056 After calling **C_VerifyInit**, the application can either call **C_Verify** to verify a signature on data in a single
5057 part; or call **C_VerifyUpdate** one or more times, followed by **C_VerifyFinal**, to verify a signature on data
5058 in multiple parts. The verification operation is active until the application calls **C_Verify** or **C_VerifyFinal**.
5059 To process additional data (in single or multiple parts), the application MUST call **C_VerifyInit** again.

5060 **C_VerifyInit** can be called with *pMechanism* set to NULL_PTR to terminate an active verification
5061 operation. If an active operation has been initialized and it cannot be cancelled,
5062 CKR_OPERATION_CANCEL_FAILED must be returned.

5063 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5064 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
5065 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
5066 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,
5067 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
5068 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
5069 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
5070 CKR_OPERATION_CANCEL_FAILED.

5071 Example: see **C_VerifyFinal**.

5072 **5.15.2 C_Verify**

```
5073 CK_DECLARE_FUNCTION(CK_RV, C_Verify) (
5074     CK_SESSION_HANDLE hSession,
5075     CK_BYTE_PTR pData,
5076     CK_ULONG ulDataLen,
5077     CK_BYTE_PTR pSignature,
5078     CK_ULONG ulSignatureLen
5079 );
```

5080 **C_Verify** verifies a signature in a single-part operation, where the signature is an appendix to the data.
5081 *hSession* is the session's handle; *pData* points to the data; *ulDataLen* is the length of the data;
5082 *pSignature* points to the signature; *ulSignatureLen* is the length of the signature.

5083 The verification operation MUST have been initialized with **C_VerifyInit**. A call to **C_Verify** always
5084 terminates the active verification operation.

5085 A successful call to **C_Verify** should return either the value CKR_OK (indicating that the supplied
5086 signature is valid) or CKR_SIGNATURE_INVALID (indicating that the supplied signature is invalid). If the

5087 signature can be seen to be invalid purely on the basis of its length, then
5088 CKR_SIGNATURE_LEN_RANGE should be returned. In any of these cases, the active signing operation
5089 is terminated.

5090 **C_Verify** cannot be used to terminate a multi-part operation, and MUST be called after **C_VerifyInit**
5091 without intervening **C_VerifyUpdate** calls.

5092 For most mechanisms, **C_Verify** is equivalent to a sequence of **C_VerifyUpdate** operations followed by
5093 **C_VerifyFinal**.

5094 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID,
5095 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
5096 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
5097 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
5098 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SIGNATURE_INVALID,
5099 CKR_SIGNATURE_LEN_RANGE, CKR_TOKEN_RESOURCE_EXCEEDED.

5100 Example: see **C_VerifyFinal** for an example of similar functions.

5101 5.15.3 C_VerifyUpdate

```
5102 CK_DECLARE_FUNCTION(CK_RV, C_VerifyUpdate) (
5103     CK_SESSION_HANDLE hSession,
5104     CK_BYTE_PTR pPart,
5105     CK_ULONG ulPartLen
5106 );
```

5107 **C_VerifyUpdate** continues a multiple-part verification operation, processing another data part. *hSession*
5108 is the session's handle, *pPart* points to the data part; *ulPartLen* is the length of the data part.

5109 The verification operation MUST have been initialized with **C_VerifyInit**. This function may be called any
5110 number of times in succession. A call to **C_VerifyUpdate** which results in an error terminates the current
5111 verification operation.

5112 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5113 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
5114 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
5115 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
5116 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
5117 CKR_TOKEN_RESOURCE_EXCEEDED.

5118 Example: see **C_VerifyFinal**.

5119 5.15.4 C_VerifyFinal

```
5120 CK_DECLARE_FUNCTION(CK_RV, C_VerifyFinal) (
5121     CK_SESSION_HANDLE hSession,
5122     CK_BYTE_PTR pSignature,
5123     CK_ULONG ulSignatureLen
5124 );
```

5125 **C_VerifyFinal** finishes a multiple-part verification operation, checking the signature. *hSession* is the
5126 session's handle; *pSignature* points to the signature; *ulSignatureLen* is the length of the signature.

5127 The verification operation MUST have been initialized with **C_VerifyInit**. A call to **C_VerifyFinal** always
5128 terminates the active verification operation.

5129 A successful call to **C_VerifyFinal** should return either the value CKR_OK (indicating that the supplied
5130 signature is valid) or CKR_SIGNATURE_INVALID (indicating that the supplied signature is invalid). If the
5131 signature can be seen to be invalid purely on the basis of its length, then
5132 CKR_SIGNATURE_LEN_RANGE should be returned. In any of these cases, the active verifying
5133 operation is terminated.

5134 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5135 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,

```
5136 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,  
5137 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,  
5138 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SIGNATURE_INVALID,  
5139 CKR_SIGNATURE_LEN_RANGE, CKR_TOKEN_RESOURCE_EXCEEDED.
```

5140 Example:

```
5141 CK_SESSION_HANDLE hSession;  
5142 CK_OBJECT_HANDLE hKey;  
5143 CK_MECHANISM mechanism = {  
5144     CKM_DES_MAC, NULL_PTR, 0  
5145 };  
5146 CK_BYTE data[] = {...};  
5147 CK_BYTE mac[4];  
5148 CK_RV rv;  
5149 .  
5150 .  
5152 rv = C_VerifyInit(hSession, &mechanism, hKey);  
5153 if (rv == CKR_OK) {  
5154     rv = C_VerifyUpdate(hSession, data, sizeof(data));  
5155     .  
5156     .  
5157     rv = C_VerifyFinal(hSession, mac, sizeof(mac));  
5158     .  
5159     .  
5160 }
```

5161 5.15.5 C_VerifyRecoverInit

```
5162 CK_DECLARE_FUNCTION(CK_RV, C_VerifyRecoverInit) (  
5163     CK_SESSION_HANDLE hSession,  
5164     CK_MECHANISM_PTR pMechanism,  
5165     CK_OBJECT_HANDLE hKey  
5166 );
```

5167 **C_VerifyRecoverInit** initializes a signature verification operation, where the data is recovered from the
5168 signature. *hSession* is the session's handle; *pMechanism* points to the structure that specifies the
5169 verification mechanism; *hKey* is the handle of the verification key.

5170 The **CKA_VERIFY_RECOVER** attribute of the verification key, which indicates whether the key supports
5171 verification where the data is recovered from the signature, MUST be CK_TRUE.

5172 After calling **C_VerifyRecoverInit**, the application may call **C_VerifyRecover** to verify a signature on
5173 data in a single part. The verification operation is active until the application uses a call to
5174 **C_VerifyRecover** to actually obtain the recovered message.

5175 **C_VerifyRecoverInit** can be called with *pMechanism* set to NULL_PTR to terminate an active verification
5176 with data recovery operation. If an active operations has been initialized and it cannot be cancelled,
5177 CKR_OPERATION_CANCEL_FAILED must be returned.

5178 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5179 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
5180 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
5181 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,

5182 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
5183 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
5184 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
5185 CKR_OPERATION_CANCEL_FAILED.

5186 Example: see **C_VerifyRecover**.

5187 5.15.6 C_VerifyRecover

```
5188 CK_DECLARE_FUNCTION(CK_RV, C_VerifyRecover) {
5189     CK_SESSION_HANDLE hSession,
5190     CK_BYTE_PTR pSignature,
5191     CK_ULONG ulSignatureLen,
5192     CK_BYTE_PTR pData,
5193     CK_ULONG_PTR pulDataLen
5194 };
```

5195 **C_VerifyRecover** verifies a signature in a single-part operation, where the data is recovered from the
5196 *signature*. *hSession* is the session's handle; *pSignature* points to the signature; *ulSignatureLen* is the
5197 length of the signature; *pData* points to the location that receives the recovered data; and *pulDataLen*
5198 points to the location that holds the length of the recovered data.

5199 **C_VerifyRecover** uses the convention described in Section 5.2 on producing output.

5200 The verification operation MUST have been initialized with **C_VerifyRecoverInit**. A call to
5201 **C_VerifyRecover** always terminates the active verification operation unless it returns
5202 CKR_BUFFER_TOO_SMALL or is a successful call (*i.e.*, one which returns CKR_OK) to determine the
5203 length of the buffer needed to hold the recovered data.

5204 A successful call to **C_VerifyRecover** should return either the value CKR_OK (indicating that the
5205 supplied signature is valid) or CKR_SIGNATURE_INVALID (indicating that the supplied signature is
5206 invalid). If the signature can be seen to be invalid purely on the basis of its length, then
5207 CKR_SIGNATURE_LEN_RANGE should be returned. The return codes CKR_SIGNATURE_INVALID
5208 and CKR_SIGNATURE_LEN_RANGE have a higher priority than the return code
5209 CKR_BUFFER_TOO_SMALL, *i.e.*, if **C_VerifyRecover** is supplied with an invalid signature, it will never
5210 return CKR_BUFFER_TOO_SMALL.

5211 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
5212 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE,
5213 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
5214 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
5215 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
5216 CKR_SESSION_HANDLE_INVALID, CKR_SIGNATURE_LEN_RANGE, CKR_SIGNATURE_INVALID,
5217 CKR_TOKEN_RESOURCE_EXCEEDED.

5218 Example:

```
5219 CK_SESSION_HANDLE hSession;
5220 CK_OBJECT_HANDLE hKey;
5221 CK_MECHANISM mechanism = {
5222     CKM_RSA_9796, NULL_PTR, 0
5223 };
5224 CK_BYTE data[] = {...};
5225 CK_ULONG ulDataLen;
5226 CK_BYTE signature[128];
5227 CK_RV rv;
5228
5229 .
```

```

5230 .
5231     rv = C_VerifyRecoverInit(hSession, &mechanism, hKey);
5232     if (rv == CKR_OK) {
5233         ulDataLen = sizeof(data);
5234         rv = C_VerifyRecover(
5235             hSession, signature, sizeof(signature), data, &ulDataLen);
5236         .
5237         .
5238     }

```

5.16 Message-based functions for verifying signatures and MACs

5240 Message-based verification refers to the process of verifying signatures on multiple messages using the
5241 same verification mechanism and verification key.

5242 Cryptoki provides the following functions for verifying signatures on messages (for the purposes of
5243 Cryptoki, these operations also encompass message authentication codes).

5.16.1 C_MessageVerifyInit

```

5245 CK_DECLARE_FUNCTION(CK_RV, C_MessageVerifyInit) (
5246     CK_SESSION_HANDLE hSession,
5247     CK_MECHANISM_PTR pMechanism,
5248     CK_OBJECT_HANDLE hKey
5249 );

```

5250 **C_MessageVerifyInit** initializes a message-based verification process, preparing a session for one or
5251 more verification operations (where the signature is an appendix to the data) that use the same
5252 verification mechanism and verification key. *hSession* is the session's handle; *pMechanism* points to the
5253 structure that specifies the verification mechanism; *hKey* is the handle of the verification key.

5254 The **CKA_VERIFY** attribute of the verification key, which indicates whether the key supports verification
5255 where the signature is an appendix to the data, MUST be CK_TRUE.

5256 After calling **C_MessageVerifyInit**, the application can either call **C_VerifyMessage** to verify a signature
5257 on a message in a single part; or call **C_VerifyMessageBegin**, followed by **C_VerifyMessageNext** one
5258 or more times, to verify a signature on a message in multiple parts. This may be repeated several times.
5259 The message-based verification process is active until the application calls **C_MessageVerifyFinal** to
5260 finish the message-based verification process.

5261 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5262 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
5263 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
5264 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,
5265 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
5266 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
5267 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

5.16.2 C_VerifyMessage

```

5269 CK_DECLARE_FUNCTION(CK_RV, C_VerifyMessage) (
5270     CK_SESSION_HANDLE hSession,
5271     CK_VOID_PTR pParameter,
5272     CK_ULONG ulParameterLen,
5273     CK_BYTE_PTR pData,

```

```

5274     CK ULONG ulDataLen,
5275     CK_BYTE_PTR pSignature,
5276     CK ULONG ulSignatureLen
5277 ) ;

```

5278 **C_VerifyMessage** verifies a signature on a message in a single part operation, where the signature is an appendix to the data. **C_MessageVerifyInit** must previously been called on the session. *hSession* is the session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message verification operation; *pData* points to the data; *ulDataLen* is the length of the data; *pSignature* points to the signature; *ulSignatureLen* is the length of the signature.

5283 Unlike the *pParameter* parameter of **C_SignMessage**, *pParameter* is always an input parameter.

5284 The message-based verification process MUST have been initialized with **C_MessageVerifyInit**. A call to **C_VerifyMessage** starts and terminates a message verification operation.

5286 A successful call to **C_VerifyMessage** should return either the value CKR_OK (indicating that the supplied signature is valid) or CKR_SIGNATURE_INVALID (indicating that the supplied signature is invalid). If the signature can be seen to be invalid purely on the basis of its length, then CKR_SIGNATURE_LEN_RANGE should be returned.

5290 **C_VerifyMessage** does not finish the message-based verification process. Additional **C_VerifyMessage** or **C_VerifyMessageBegin** and **C_VerifyMessageNext** calls may be made on the session.

5292 For most mechanisms, **C_VerifyMessage** is equivalent to **C_VerifyMessageBegin** followed by a sequence of **C_VerifyMessageNext** operations.

5294 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SIGNATURE_INVALID, CKR_SIGNATURE_LEN_RANGE, CKR_TOKEN_RESOURCE_EXCEEDED.

5.16.3 C_VerifyMessageBegin

```

5301 CK_DECLARE_FUNCTION(CK_RV, C_VerifyMessageBegin) (
5302     CK_SESSION_HANDLE hSession,
5303     CK_VOID_PTR pParameter,
5304     CK ULONG ulParameterLen
5305 ) ;

```

5306 **C_VerifyMessageBegin** begins a multiple-part message verification operation, where the signature is an appendix to the message. **C_MessageVerifyInit** must previously been called on the session. *hSession* is the session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message verification operation.

5310 Unlike the *pParameter* parameter of **C_SignMessageBegin**, *pParameter* is always an input parameter.

5311 After calling **C_VerifyMessageBegin**, the application should call **C_VerifyMessageNext** one or more times to verify a signature on a message in multiple parts. The message verification operation is active until the application calls **C_VerifyMessageNext** with a non-NULL *pSignature*. To process additional messages (in single or multiple parts), the application MUST call **C_VerifyMessage** or **C_VerifyMessageBegin** again.

5316 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_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

5321 **5.16.4 C_VerifyMessageNext**

```
5322 CK_DECLARE_FUNCTION(CK_RV, C_VerifyMessageNext) (
5323     CK_SESSION_HANDLE hSession,
5324     CK_VOID_PTR pParameter,
5325     CK ULONG ulParameterLen,
5326     CK_BYTE_PTR pDataPart,
5327     CK ULONG ulDataPartLen,
5328     CK_BYTE_PTR pSignature,
5329     CK ULONG ulSignatureLen
5330 );
```

5331 **C_VerifyMessageNext** continues a multiple-part message verification operation, processing another data
5332 part, or finishes a multiple-part message verification operation, checking the signature. *hSession* is the
5333 session's handle, *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the
5334 message verification operation, *pDataPart* points to the data part; *ulPartLen* is the length of the data part;
5335 *pSignature* points to the signature; *ulSignatureLen* is the length of the signature.

5336 The *pSignature* argument is set to NULL if there is more data part to follow, or set to a non-NULL value
5337 (pointing to the signature to verify) if this is the last data part.

5338 The message verification operation MUST have been started with **C_VerifyMessageBegin**. This function
5339 may be called any number of times in succession. A call to **C_VerifyMessageNext** with a NULL
5340 *pSignature* which results in an error terminates the current message verification operation. A call to
5341 **C_VerifyMessageNext** with a non-NULL *pSignature* always terminates the active message verification
5342 operation.

5343 A successful call to **C_VerifyMessageNext** with a non-NULL *pSignature* should return either the value
5344 CKR_OK (indicating that the supplied signature is valid) or CKR_SIGNATURE_INVALID (indicating that
5345 the supplied signature is invalid). If the signature can be seen to be invalid purely on the basis of its
5346 length, then CKR_SIGNATURE_LEN_RANGE should be returned. In any of these cases, the active
5347 message verifying operation is terminated.

5348 Although the last **C_VerifyMessageNext** call ends the verification of a message, it does not finish the
5349 message-based verification process. Additional **C_VerifyMessage** or **C_VerifyMessageBegin** and
5350 **C_VerifyMessageNext** calls may be made on the session.

5351 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5352 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
5353 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
5354 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
5355 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SIGNATURE_INVALID,
5356 CKR_SIGNATURE_LEN_RANGE, CKR_TOKEN_RESOURCE_EXCEEDED.

5357 **5.16.5 C_MessageVerifyFinal**

```
5358 CK_DECLARE_FUNCTION(CK_RV, C_MessageVerifyFinal) (
5359     CK_SESSION_HANDLE hSession
5360 );
```

5361 **C_MessageVerifyFinal** finishes a message-based verification process. *hSession* is the session's handle.

5362 The message-based verification process MUST have been initialized with **C_MessageVerifyInit**.

5363 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5364 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
5365 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
5366 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,

5367 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
5368 CKR_TOKEN_RESOURCE_EXCEEDED.

5369 **5.17 Dual-function cryptographic functions**

5370 Cryptoki provides the following functions to perform two cryptographic operations “simultaneously” within
5371 a session. These functions are provided so as to avoid unnecessarily passing data back and forth to and
5372 from a token.

5373 **5.17.1 C_DigestEncryptUpdate**

```
5374 CK_DECLARE_FUNCTION(CK_RV, C_DigestEncryptUpdate) (
5375     CK_SESSION_HANDLE hSession,
5376     CK_BYTE_PTR pPart,
5377     CK ULONG ulPartLen,
5378     CK_BYTE_PTR pEncryptedPart,
5379     CK ULONG_PTR pulEncryptedPartLen
5380 );
```

5381 **C_DigestEncryptUpdate** continues multiple-part digest and encryption operations, processing another
5382 data part. *hSession* is the session’s handle; *pPart* points to the data part; *ulPartLen* is the length of the
5383 data part; *pEncryptedPart* points to the location that receives the digested and encrypted data part;
5384 *pulEncryptedPartLen* points to the location that holds the length of the encrypted data part.

5385 **C_DigestEncryptUpdate** uses the convention described in Section 5.2 on producing output. If a
5386 **C_DigestEncryptUpdate** call does not produce encrypted output (because an error occurs, or because
5387 *pEncryptedPart* has the value `NULL_PTR`, or because *pulEncryptedPartLen* is too small to hold the entire
5388 encrypted part output), then no plaintext is passed to the active digest operation.

5389 Digest and encryption operations MUST both be active (they MUST have been initialized with
5390 **C_DigestInit** and **C_EncryptInit**, respectively). This function may be called any number of times in
5391 succession, and may be interspersed with **C_DigestUpdate**, **C_DigestKey**, and **C_EncryptUpdate** calls
5392 (it would be somewhat unusual to intersperse calls to **C_DigestEncryptUpdate** with calls to
5393 **C_DigestKey**, however).

5394 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
5395 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR,
5396 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED,
5397 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
5398 CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.
5399 Example:

```
5400 #define BUF_SZ 512
5401
5402 CK_SESSION_HANDLE hSession;
5403 CK_OBJECT_HANDLE hKey;
5404 CK_BYTE iv[8];
5405 CK_MECHANISM digestMechanism = {
5406     CKM_MD5, NULL_PTR, 0
5407 };
5408 CK_MECHANISM encryptionMechanism = {
5409     CKM_DES_ECB, iv, sizeof(iv)
5410 };
5411 CK_BYTE encryptedData[BUF_SZ];
5412 CK ULONG ulEncryptedDataLen;
```

```

5413 CK_BYTE digest[16];
5414 CK_ULONG ulDigestLen;
5415 CK_BYTE data[(2*BUF_SZ)+8];
5416 CK_RV rv;
5417 int i;
5418 .
5419 .
5420 .
5421 memset(iv, 0, sizeof(iv));
5422 memset(data, 'A', ((2*BUF_SZ)+5));
5423 rv = C_EncryptInit(hSession, &encryptionMechanism, hKey);
5424 if (rv != CKR_OK) {
5425 .
5426 .
5427 }
5428 rv = C_DigestInit(hSession, &digestMechanism);
5429 if (rv != CKR_OK) {
5430 .
5431 .
5432 }
5433
5434 ulEncryptedDataLen = sizeof(encryptedData);
5435 rv = C_DigestEncryptUpdate(
5436     hSession,
5437     &data[0], BUF_SZ,
5438     encryptedData, &ulEncryptedDataLen);
5439 .
5440 .
5441 ulEncryptedDataLen = sizeof(encryptedData);
5442 rv = C_DigestEncryptUpdate(
5443     hSession,
5444     &data[BUF_SZ], BUF_SZ,
5445     encryptedData, &ulEncryptedDataLen);
5446 .
5447 .
5448 /*
5449 * The last portion of the buffer needs to be
5450 * handled with separate calls to deal with
5451 * padding issues in ECB mode
5452 */
5453
5454
5455 /* First, complete the digest on the buffer */

```

```

5456 rv = C_DigestUpdate(hSession, &data[BUF_SZ*2], 5);
5457 .
5458 .
5459 ulDigestLen = sizeof(digest);
5460 rv = C_DigestFinal(hSession, digest, &ulDigestLen);
5461 .
5462 .
5463
5464 /* Then, pad last part with 3 0x00 bytes, and complete encryption */
5465 for(i=0;i<3;i++)
5466     data[((BUF_SZ*2)+5)+i] = 0x00;
5467
5468 /* Now, get second-to-last piece of ciphertext */
5469 ulEncryptedDataLen = sizeof(encryptedData);
5470 rv = C_EncryptUpdate(
5471     hSession,
5472     &data[BUF_SZ*2], 8,
5473     encryptedData, &ulEncryptedDataLen);
5474 .
5475 .
5476
5477 /* Get last piece of ciphertext (should have length 0, here) */
5478 ulEncryptedDataLen = sizeof(encryptedData);
5479 rv = C_EncryptFinal(hSession, encryptedData, &ulEncryptedDataLen);
5480 .
5481 .

```

5.17.2 C_DecryptDigestUpdate

```

5483 CK_DECLARE_FUNCTION(CK_RV, C_DecryptDigestUpdate) (
5484     CK_SESSION_HANDLE hSession,
5485     CK_BYTE_PTR pEncryptedPart,
5486     CK_ULONG ulEncryptedPartLen,
5487     CK_BYTE_PTR pPart,
5488     CK ULONG_PTR pulPartLen
5489 );

```

5490 **C_DecryptDigestUpdate** continues a multiple-part combined decryption and digest operation,
 5491 processing another data part. *hSession* is the session's handle; *pEncryptedPart* points to the encrypted
 5492 data part; *ulEncryptedPartLen* is the length of the encrypted data part; *pPart* points to the location that
 5493 receives the recovered data part; *pulPartLen* points to the location that holds the length of the recovered
 5494 data part.

5495 **C_DecryptDigestUpdate** uses the convention described in Section 5.2 on producing output. If a
 5496 **C_DecryptDigestUpdate** call does not produce decrypted output (because an error occurs, or because
 5497 *pPart* has the value `NULL_PTR`, or because *pulPartLen* is too small to hold the entire decrypted part
 5498 output), then no plaintext is passed to the active digest operation.

5499 Decryption and digesting operations MUST both be active (they MUST have been initialized with
 5500 **C_DecryptInit** and **C_DigestInit**, respectively). This function may be called any number of times in

5501 succession, and may be interspersed with **C_DecryptUpdate**, **C_DigestUpdate**, and **C_DigestKey** calls
 5502 (it would be somewhat unusual to intersperse calls to **C_DigestEncryptUpdate** with calls to
 5503 **C_DigestKey**, however).

5504 Use of **C_DecryptDigestUpdate** involves a pipelining issue that does not arise when using
 5505 **C_DigestEncryptUpdate**, the “inverse function” of **C_DecryptDigestUpdate**. This is because when
 5506 **C_DigestEncryptUpdate** is called, precisely the same input is passed to both the active digesting
 5507 operation and the active encryption operation; however, when **C_DecryptDigestUpdate** is called, the
 5508 input passed to the active digesting operation is the *output* of the active decryption operation. This issue
 5509 comes up only when the mechanism used for decryption performs padding.

5510 In particular, envision a 24-byte ciphertext which was obtained by encrypting an 18-byte plaintext with
 5511 DES in CBC mode with PKCS padding. Consider an application which will simultaneously decrypt this
 5512 ciphertext and digest the original plaintext thereby obtained.

5513 After initializing decryption and digesting operations, the application passes the 24-byte ciphertext (3 DES
 5514 blocks) into **C_DecryptDigestUpdate**. **C_DecryptDigestUpdate** returns exactly 16 bytes of plaintext,
 5515 since at this point, Cryptoki doesn’t know if there’s more ciphertext coming, or if the last block of
 5516 ciphertext held any padding. These 16 bytes of plaintext are passed into the active digesting operation.

5517 Since there is no more ciphertext, the application calls **C_DecryptFinal**. This tells Cryptoki that there’s
 5518 no more ciphertext coming, and the call returns the last 2 bytes of plaintext. However, since the active
 5519 decryption and digesting operations are linked *only* through the **C_DecryptDigestUpdate** call, these 2
 5520 bytes of plaintext are *not* passed on to be digested.

5521 A call to **C_DigestFinal**, therefore, would compute the message digest of *the first 16 bytes of the*
 5522 *plaintext*, not the message digest of the entire plaintext. It is crucial that, before **C_DigestFinal** is called,
 5523 the last 2 bytes of plaintext get passed into the active digesting operation via a **C_DigestUpdate** call.

5524 Because of this, it is critical that when an application uses a padded decryption mechanism with
 5525 **C_DecryptDigestUpdate**, it knows exactly how much plaintext has been passed into the active digesting
 5526 operation. *Extreme caution is warranted when using a padded decryption mechanism with*
 5527 **C_DecryptDigestUpdate**.

5528 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
 5529 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
 5530 CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID,
 5531 CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
 5532 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
 5533 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

5534 Example:

```

5535 #define BUF_SZ 512
5536
5537 CK_SESSION_HANDLE hSession;
5538 CK_OBJECT_HANDLE hKey;
5539 CK_BYTE iv[8];
5540 CK_MECHANISM decryptionMechanism = {
5541   CKM_DES_ECB, iv, sizeof(iv)
5542 };
5543 CK_MECHANISM digestMechanism = {
5544   CKM_MD5, NULL_PTR, 0
5545 };
5546 CK_BYTE encryptedData[(2*BUF_SZ)+8];
5547 CK_BYTE digest[16];
5548 CK_ULONG ulDigestLen;
5549 CK_BYTE data[BUF_SZ];
  
```

```

5550 CK ULONG ulDataLen, ulLastUpdateSize;
5551 CK_RV rv;
5552 .
5553 .
5554 .
5555 memset(iv, 0, sizeof(iv));
5556 memset(encryptedData, 'A', ((2*BUF_SZ)+8));
5557 rv = C_DecryptInit(hSession, &decryptionMechanism, hKey);
5558 if (rv != CKR_OK) {
5559 .
5560 .
5561 }
5562 rv = C_DigestInit(hSession, &digestMechanism);
5563 if (rv != CKR_OK) {
5564 .
5565 .
5566 }
5567 .
5568 ulDataLen = sizeof(data);
5569 rv = C_DecryptDigestUpdate(
5570     hSession,
5571     &encryptedData[0], BUF_SZ,
5572     data, &ulDataLen);
5573 .
5574 .
5575 ulDataLen = sizeof(data);
5576 rv = C_DecryptDigestUpdate(
5577     hSession,
5578     &encryptedData[BUF_SZ], BUF_SZ,
5579     data, &ulDataLen);
5580 .
5581 .
5582 /*
5583 * The last portion of the buffer needs to be handled with
5584 * separate calls to deal with padding issues in ECB mode
5585 */
5586 .
5587 /*
5588 * First, complete the decryption of the buffer */
5589 ulLastUpdateSize = sizeof(data);
5590 rv = C_DecryptUpdate(
5591     hSession,
5592     &encryptedData[BUF_SZ*2], 8,

```

```

5593     data, &ulLastUpdateSize);
5594 .
5595 .
5596 /* Get last piece of plaintext (should have length 0, here) */
5597 ulDataLen = sizeof(data)-ulLastUpdateSize;
5598 rv = C_DecryptFinal(hSession, &data[ulLastUpdateSize], &ulDataLen);
5599 if (rv != CKR_OK) {
5600 .
5601 .
5602 }
5603 .
5604 /* Digest last bit of plaintext */
5605 rv = C_DigestUpdate(hSession, data, 5);
5606 if (rv != CKR_OK) {
5607 .
5608 .
5609 }
5610 ulDigestLen = sizeof(digest);
5611 rv = C_DigestFinal(hSession, digest, &ulDigestLen);
5612 if (rv != CKR_OK) {
5613 .
5614 .
5615 }

```

5616 5.17.3 C_SignEncryptUpdate

```

5617 CK_DECLARE_FUNCTION(CK_RV, C_SignEncryptUpdate) (
5618     CK_SESSION_HANDLE hSession,
5619     CK_BYTE_PTR pPart,
5620     CK ULONG ulPartLen,
5621     CK_BYTE_PTR pEncryptedPart,
5622     CK ULONG_PTR pulEncryptedPartLen
5623 );

```

5624 **C_SignEncryptUpdate** continues a multiple-part combined signature and encryption operation,
 5625 processing another data part. *hSession* is the session's handle; *pPart* points to the data part; *ulPartLen* is
 5626 the length of the data part; *pEncryptedPart* points to the location that receives the digested and encrypted
 5627 data part; and *pulEncryptedPartLen* points to the location that holds the length of the encrypted data part.

5628 **C_SignEncryptUpdate** uses the convention described in Section 5.2 on producing output. If a
 5629 **C_SignEncryptUpdate** call does not produce encrypted output (because an error occurs, or because
 5630 *pEncryptedPart* has the value `NULL_PTR`, or because *pulEncryptedPartLen* is too small to hold the entire
 5631 encrypted part output), then no plaintext is passed to the active signing operation.

5632 Signature and encryption operations MUST both be active (they MUST have been initialized with
 5633 **C_SignInit** and **C_EncryptInit**, respectively). This function may be called any number of times in
 5634 succession, and may be interspersed with **C_SignUpdate** and **C_EncryptUpdate** calls.

5635 Return values: `CKR_ARGUMENTS_BAD`, `CKR_BUFFER_TOO_SMALL`,
 5636 `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DATA_LEN_RANGE`, `CKR_DEVICE_ERROR`,
 5637 `CKR_DEVICE_MEMORY`, `CKR_DEVICE_REMOVED`, `CKR_FUNCTION_CANCELED`,
 5638 `CKR_FUNCTION_FAILED`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`,

5639 CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
5640 CKR_USER_NOT_LOGGED_IN.

5641 Example:

```
5642 #define BUF_SZ 512
5643
5644 CK_SESSION_HANDLE hSession;
5645 CK_OBJECT_HANDLE hEncryptionKey, hMacKey;
5646 CK_BYTE iv[8];
5647 CK_MECHANISM signMechanism = {
5648     CKM_DES_MAC, NULL_PTR, 0
5649 };
5650 CK_MECHANISM encryptionMechanism = {
5651     CKM_DES_ECB, iv, sizeof(iv)
5652 };
5653 CK_BYTE encryptedData[BUF_SZ];
5654 CK_ULONG ulEncryptedDataLen;
5655 CK_BYTE MAC[4];
5656 CK_ULONG ulMacLen;
5657 CK_BYTE data[(2*BUF_SZ)+8];
5658 CK_RV rv;
5659 int i;
5660
5661 .
5662 .
5663 memset(iv, 0, sizeof(iv));
5664 memset(data, 'A', ((2*BUF_SZ)+5));
5665 rv = C_EncryptInit(hSession, &encryptionMechanism, hEncryptionKey);
5666 if (rv != CKR_OK) {
5667 .
5668 .
5669 }
5670 rv = C_SignInit(hSession, &signMechanism, hMacKey);
5671 if (rv != CKR_OK) {
5672 .
5673 .
5674 }
5675
5676 ulEncryptedDataLen = sizeof(encryptedData);
5677 rv = C_SignEncryptUpdate(
5678     hSession,
5679     &data[0], BUF_SZ,
5680     encryptedData, &ulEncryptedDataLen);
5681 .
```

```

5682 .
5683 ulEncryptedDataLen = sizeof(encryptedData);
5684 rv = C_SignEncryptUpdate(
5685     hSession,
5686     &data[BUF_SZ], BUF_SZ,
5687     encryptedData, &ulEncryptedDataLen);
5688 .
5689 .
5690 .
5691 /*
5692 * The last portion of the buffer needs to be handled with
5693 * separate calls to deal with padding issues in ECB mode
5694 */
5695 .
5696 /* First, complete the signature on the buffer */
5697 rv = C_SignUpdate(hSession, &data[BUF_SZ*2], 5);
5698 .
5699 .
5700 ulMacLen = sizeof(MAC);
5701 rv = C_SignFinal(hSession, MAC, &ulMacLen);
5702 .
5703 .
5704 .
5705 /* Then pad last part with 3 0x00 bytes, and complete encryption */
5706 for(i=0;i<3;i++)
5707     data[((BUF_SZ*2)+5)+i] = 0x00;
5708 .
5709 /* Now, get second-to-last piece of ciphertext */
5710 ulEncryptedDataLen = sizeof(encryptedData);
5711 rv = C_EncryptUpdate(
5712     hSession,
5713     &data[BUF_SZ*2], 8,
5714     encryptedData, &ulEncryptedDataLen);
5715 .
5716 .
5717 .
5718 /* Get last piece of ciphertext (should have length 0, here) */
5719 ulEncryptedDataLen = sizeof(encryptedData);
5720 rv = C_EncryptFinal(hSession, encryptedData, &ulEncryptedDataLen);
5721 .
5722 .

```

5723 **5.17.4 C_DecryptVerifyUpdate**

```
5724 CK_DECLARE_FUNCTION(CK_RV, C_DecryptVerifyUpdate) (
5725     CK_SESSION_HANDLE hSession,
5726     CK_BYTE_PTR pEncryptedPart,
5727     CK ULONG ulEncryptedPartLen,
5728     CK_BYTE_PTR pPart,
5729     CK ULONG_PTR pulPartLen
5730 );
```

5731 **C_DecryptVerifyUpdate** continues a multiple-part combined decryption and verification operation,
5732 processing another data part. *hSession* is the session's handle; *pEncryptedPart* points to the encrypted
5733 data; *ulEncryptedPartLen* is the length of the encrypted data; *pPart* points to the location that receives the
5734 recovered data; and *pulPartLen* points to the location that holds the length of the recovered data.

5735 **C_DecryptVerifyUpdate** uses the convention described in Section 5.2 on producing output. If a
5736 **C_DecryptVerifyUpdate** call does not produce decrypted output (because an error occurs, or because
5737 *pPart* has the value `NULL_PTR`, or because *pulPartLen* is too small to hold the entire encrypted part
5738 output), then no plaintext is passed to the active verification operation.

5739 Decryption and signature operations MUST both be active (they MUST have been initialized with
5740 **C_DecryptInit** and **C_VerifyInit**, respectively). This function may be called any number of times in
5741 succession, and may be interspersed with **C_DecryptUpdate** and **C_VerifyUpdate** calls.

5742 Use of **C_DecryptVerifyUpdate** involves a pipelining issue that does not arise when using
5743 **C_SignEncryptUpdate**, the "inverse function" of **C_DecryptVerifyUpdate**. This is because when
5744 **C_SignEncryptUpdate** is called, precisely the same input is passed to both the active signing operation
5745 and the active encryption operation; however, when **C_DecryptVerifyUpdate** is called, the input passed
5746 to the active verifying operation is the *output* of the active decryption operation. This issue comes up only
5747 when the mechanism used for decryption performs padding.

5748 In particular, envision a 24-byte ciphertext which was obtained by encrypting an 18-byte plaintext with
5749 DES in CBC mode with PKCS padding. Consider an application which will simultaneously decrypt this
5750 ciphertext and verify a signature on the original plaintext thereby obtained.

5751 After initializing decryption and verification operations, the application passes the 24-byte ciphertext (3
5752 DES blocks) into **C_DecryptVerifyUpdate**. **C_DecryptVerifyUpdate** returns exactly 16 bytes of
5753 plaintext, since at this point, Cryptoki doesn't know if there's more ciphertext coming, or if the last block of
5754 ciphertext held any padding. These 16 bytes of plaintext are passed into the active verification operation.

5755 Since there is no more ciphertext, the application calls **C_DecryptFinal**. This tells Cryptoki that there's
5756 no more ciphertext coming, and the call returns the last 2 bytes of plaintext. However, since the active
5757 decryption and verification operations are linked *only* through the **C_DecryptVerifyUpdate** call, these 2
5758 bytes of plaintext are *not* passed on to the verification mechanism.

5759 A call to **C_VerifyFinal**, therefore, would verify whether or not the signature supplied is a valid signature
5760 on the *first 16 bytes of the plaintext*, not on the entire plaintext. It is crucial that, before **C_VerifyFinal** is
5761 called, the last 2 bytes of plaintext get passed into the active verification operation via a **C_VerifyUpdate**
5762 call.

5763 Because of this, it is critical that when an application uses a padded decryption mechanism with
5764 **C_DecryptVerifyUpdate**, it knows exactly how much plaintext has been passed into the active
5765 verification operation. *Extreme caution is warranted when using a padded decryption mechanism with*
5766 **C_DecryptVerifyUpdate**.

5767 Return values: `CKR_ARGUMENTS_BAD`, `CKR_BUFFER_TOO_SMALL`,
5768 `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DATA_LEN_RANGE`, `CKR_DEVICE_ERROR`,
5769 `CKR_DEVICE_MEMORY`, `CKR_DEVICE_REMOVED`, `CKR_ENCRYPTED_DATA_INVALID`,
5770 `CKR_ENCRYPTED_DATA_LEN_RANGE`, `CKR_FUNCTION_CANCELED`, `CKR_FUNCTION_FAILED`,
5771 `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`, `CKR_OPERATION_NOT_INITIALIZED`,
5772 `CKR_SESSION_CLOSED`, `CKR_SESSION_HANDLE_INVALID`.

5773 Example:

```
5774 #define BUF_SZ 512
```

```

5775
5776     CK_SESSION_HANDLE hSession;
5777     CK_OBJECT_HANDLE hDecryptionKey, hMacKey;
5778     CK_BYTE iv[8];
5779     CK_MECHANISM decryptionMechanism = {
5780         CKM_DES_ECB, iv, sizeof(iv)
5781     };
5782     CK_MECHANISM verifyMechanism = {
5783         CKM_DES_MAC, NULL_PTR, 0
5784     };
5785     CK_BYTE encryptedData[(2*BUF_SZ)+8];
5786     CK_BYTE MAC[4];
5787     CK ULONG ulMacLen;
5788     CK_BYTE data[BUF_SZ];
5789     CK ULONG ulDataLen, ulLastUpdateSize;
5790     CK_RV rv;
5791
5792 .
5793 .
5794     memset(iv, 0, sizeof(iv));
5795     memset(encryptedData, 'A', ((2*BUF_SZ)+8));
5796     rv = C_DecryptInit(hSession, &decryptionMechanism, hDecryptionKey);
5797     if (rv != CKR_OK) {
5798 .
5799 .
5800 }
5801     rv = C_VerifyInit(hSession, &verifyMechanism, hMacKey);
5802     if (rv != CKR_OK) {
5803 .
5804 .
5805 }
5806
5807     ulDataLen = sizeof(data);
5808     rv = C_DecryptVerifyUpdate(
5809         hSession,
5810         &encryptedData[0], BUF_SZ,
5811         data, &ulDataLen);
5812 .
5813 .
5814     ulDataLen = sizeof(data);
5815     rv = C_DecryptVerifyUpdate(
5816         hSession,
5817         &encryptedData[BUF_SZ], BUF_SZ,

```

```

5818     data, &ulDataLen);
5819 .
5820 .
5821 /*
5822 * The last portion of the buffer needs to be handled with
5823 * separate calls to deal with padding issues in ECB mode
5824 */
5825
5826
5827 /* First, complete the decryption of the buffer */
5828 ulLastUpdateSize = sizeof(data);
5829 rv = C_DecryptUpdate(
5830     hSession,
5831     &encryptedData[BUF_SZ*2], 8,
5832     data, &ulLastUpdateSize);
5833 .
5834 .
5835 /* Get last little piece of plaintext. Should have length 0 */
5836 ulDataLen = sizeof(data)-ulLastUpdateSize;
5837 rv = C_DecryptFinal(hSession, &data[ulLastUpdateSize], &ulDataLen);
5838 if (rv != CKR_OK) {
5839     .
5840     .
5841 }
5842
5843 /* Send last bit of plaintext to verification operation */
5844 rv = C_VerifyUpdate(hSession, data, 5);
5845 if (rv != CKR_OK) {
5846     .
5847     .
5848 }
5849 rv = C_VerifyFinal(hSession, MAC, ulMacLen);
5850 if (rv == CKR_SIGNATURE_INVALID) {
5851     .
5852     .
5853 }

```

5854 **5.18 Key management functions**

5855 Cryptoki provides the following functions for key management:

5856 **5.18.1 C_GenerateKey**

```

5857 CK_DECLARE_FUNCTION(CK_RV, C_GenerateKey) (
5858     CK_SESSION_HANDLE hSession

```

```
5859     CK_MECHANISM_PTR pMechanism,
5860     CK_ATTRIBUTE_PTR pTemplate,
5861     CK_ULONG ulCount,
5862     CK_OBJECT_HANDLE_PTR phKey
5863 );
```

5864 **C_GenerateKey** generates a secret key or set of domain parameters, creating a new object. *hSession* is
5865 the session's handle; *pMechanism* points to the generation mechanism; *pTemplate* points to the template
5866 for the new key or set of domain parameters; *ulCount* is the number of attributes in the template; *phKey*
5867 points to the location that receives the handle of the new key or set of domain parameters.

5868 If the generation mechanism is for domain parameter generation, the **CKA_CLASS** attribute will have the
5869 value **CKO_DOMAIN_PARAMETERS**; otherwise, it will have the value **CKO_SECRET_KEY**.

5870 Since the type of key or domain parameters to be generated is implicit in the generation mechanism, the
5871 template does not need to supply a key type. If it does supply a key type which is inconsistent with the
5872 generation mechanism, **C_GenerateKey** fails and returns the error code
5873 **CKR_TEMPLATE_INCONSISTENT**. The **CKA_CLASS** attribute is treated similarly.

5874 If a call to **C_GenerateKey** cannot support the precise template supplied to it, it will fail and return without
5875 creating an object.

5876 The object created by a successful call to **C_GenerateKey** will have its **CKA_LOCAL** attribute set to
5877 **CK_TRUE**. In addition, the object created will have a value for **CKA_UNIQUE_ID** generated and
5878 assigned (See Section 4.4.1).

5879 Return values: **CKR_ARGUMENTS_BAD**, **CKR_ATTRIBUTE_READ_ONLY**,
5880 **CKR_ATTRIBUTE_TYPE_INVALID**, **CKR_ATTRIBUTE_VALUE_INVALID**,
5881 **CKR_CRYPTOKI_NOT_INITIALIZED**, **CKR_CURVE_NOT_SUPPORTED**, **CKR_DEVICE_ERROR**,
5882 **CKR_DEVICE_MEMORY**, **CKR_DEVICE_REMOVED**, **CKR_FUNCTION_CANCELED**,
5883 **CKR_FUNCTION_FAILED**, **CKR_GENERAL_ERROR**, **CKR_HOST_MEMORY**,
5884 **CKR_MECHANISM_INVALID**, **CKR_MECHANISM_PARAM_INVALID**, **CKR_OK**,
5885 **CKR_OPERATION_ACTIVE**, **CKR_PIN_EXPIRED**, **CKR_SESSION_CLOSED**,
5886 **CKR_SESSION_HANDLE_INVALID**, **CKR_SESSION_READ_ONLY**, **CKR_TEMPLATE_INCOMPLETE**,
5887 **CKR_TEMPLATE_INCONSISTENT**, **CKR_TOKEN_WRITE_PROTECTED**,
5888 **CKR_USER_NOT_LOGGED_IN**.

5889 Example:

```
5890 CK_SESSION_HANDLE hSession;
5891 CK_OBJECT_HANDLE hKey;
5892 CK_MECHANISM mechanism = {
5893     CKM_DES_KEY_GEN, NULL_PTR, 0
5894 };
5895 CK_RV rv;
5896 .
5897 .
5898 .
5899 rv = C_GenerateKey(hSession, &mechanism, NULL_PTR, 0, &hKey);
5900 if (rv == CKR_OK) {
5901     .
5902     .
5903 }
```

5904 **5.18.2 C_GenerateKeyPair**

```
5905 CK_DECLARE_FUNCTION(CK_RV, C_GenerateKeyPair) (
5906     CK_SESSION_HANDLE hSession,
5907     CK_MECHANISM_PTR pMechanism,
5908     CK_ATTRIBUTE_PTR pPublicKeyTemplate,
5909     CK ULONG ulPublicKeyAttributeCount,
5910     CK_ATTRIBUTE_PTR pPrivateKeyTemplate,
5911     CK ULONG ulPrivateKeyAttributeCount,
5912     CK_OBJECT_HANDLE_PTR phPublicKey,
5913     CK_OBJECT_HANDLE_PTR phPrivateKey
5914 );
```

5915 **C_GenerateKeyPair** generates a public/private key pair, creating new key objects. *hSession* is the
5916 session's handle; *pMechanism* points to the key generation mechanism; *pPublicKeyTemplate* points to
5917 the template for the public key; *ulPublicKeyAttributeCount* is the number of attributes in the public-key
5918 template; *pPrivateKeyTemplate* points to the template for the private key; *ulPrivateKeyAttributeCount* is
5919 the number of attributes in the private-key template; *phPublicKey* points to the location that receives the
5920 handle of the new public key; *phPrivateKey* points to the location that receives the handle of the new
5921 private key.

5922 Since the types of keys to be generated are implicit in the key pair generation mechanism, the templates
5923 do not need to supply key types. If one of the templates does supply a key type which is inconsistent with
5924 the key generation mechanism, **C_GenerateKeyPair** fails and returns the error code
5925 **CKR_TEMPLATE_INCONSISTENT**. The **CKA_CLASS** attribute is treated similarly.

5926 If a call to **C_GenerateKeyPair** cannot support the precise templates supplied to it, it will fail and return
5927 without creating any key objects.

5928 A call to **C_GenerateKeyPair** will never create just one key and return. A call can fail, and create no
5929 keys; or it can succeed, and create a matching public/private key pair.

5930 The key objects created by a successful call to **C_GenerateKeyPair** will have their **CKA_LOCAL**
5931 attributes set to **CK_TRUE**. In addition, the key objects created will both have values for
5932 **CKA_UNIQUE_ID** generated and assigned (See Section 4.4.1).

5933 Note carefully the order of the arguments to **C_GenerateKeyPair**. The last two arguments do not have
5934 the same order as they did in the original Cryptoki Version 1.0 document. The order of these two
5935 arguments has caused some unfortunate confusion.

5936 Return values: **CKR_ARGUMENTS_BAD**, **CKR_ATTRIBUTE_READ_ONLY**,
5937 **CKR_ATTRIBUTE_TYPE_INVALID**, **CKR_ATTRIBUTE_VALUE_INVALID**,
5938 **CKR_CRYPTOKI_NOT_INITIALIZED**, **CKR_CURVE_NOT_SUPPORTED**, **CKR_DEVICE_ERROR**,
5939 **CKR_DEVICE_MEMORY**, **CKR_DEVICE_REMOVED**, **CKR_DOMAIN_PARAMS_INVALID**,
5940 **CKR_FUNCTION_CANCELED**, **CKR_FUNCTION_FAILED**, **CKR_GENERAL_ERROR**,
5941 **CKR_HOST_MEMORY**, **CKR_MECHANISM_INVALID**, **CKR_MECHANISM_PARAM_INVALID**,
5942 **CKR_OK**, **CKR_OPERATION_ACTIVE**, **CKR_PIN_EXPIRED**, **CKR_SESSION_CLOSED**,
5943 **CKR_SESSION_HANDLE_INVALID**, **CKR_SESSION_READ_ONLY**, **CKR_TEMPLATE_INCOMPLETE**,
5944 **CKR_TEMPLATE_INCONSISTENT**, **CKR_TOKEN_WRITE_PROTECTED**,
5945 **CKR_USER_NOT_LOGGED_IN**.

5946 Example:

```
5947 CK_SESSION_HANDLE hSession;
5948 CK_OBJECT_HANDLE hPublicKey, hPrivateKey;
5949 CK_MECHANISM mechanism = {
5950     CKM_RSA_PKCS_KEY_PAIR_GEN, NULL_PTR, 0
5951 };
5952 CK ULONG modulusBits = 3072;
5953 CK_BYTE publicExponent[] = { 3 };
```

```

5954 CK_BYTE subject[] = {...};
5955 CK_BYTE id[] = {123};
5956 CK_BBOOL true = CK_TRUE;
5957 CK_ATTRIBUTE publicKeyTemplate[] = {
5958     {CKA_ENCRYPT, &true, sizeof(true)},
5959     {CKA_VERIFY, &true, sizeof(true)},
5960     {CKA_WRAP, &true, sizeof(true)},
5961     {CKA_MODULUS_BITS, &modulusBits, sizeof(modulusBits)},
5962     {CKA_PUBLIC_EXPONENT, publicExponent, sizeof(publicExponent)}}
5963 };
5964 CK_ATTRIBUTE privateKeyTemplate[] = {
5965     {CKA_TOKEN, &true, sizeof(true)},
5966     {CKA_PRIVATE, &true, sizeof(true)},
5967     {CKA SUBJECT, subject, sizeof(subject)},
5968     {CKA_ID, id, sizeof(id)},
5969     {CKA_SENSITIVE, &true, sizeof(true)},
5970     {CKA_DECRYPT, &true, sizeof(true)},
5971     {CKA_SIGN, &true, sizeof(true)},
5972     {CKA_UNWRAP, &true, sizeof(true)}}
5973 };
5974 CK_RV rv;
5975
5976 rv = C_GenerateKeyPair(
5977     hSession, &mechanism,
5978     publicKeyTemplate, 5,
5979     privateKeyTemplate, 8,
5980     &hPublicKey, &hPrivateKey);
5981 if (rv == CKR_OK) {
5982 .
5983 .
5984 }

```

5985 5.18.3 C_WrapKey

```

5986 CK_DECLARE_FUNCTION(CK_RV, C_WrapKey) (
5987     CK_SESSION_HANDLE hSession,
5988     CK_MECHANISM_PTR pMechanism,
5989     CK_OBJECT_HANDLE hWrappingKey,
5990     CK_OBJECT_HANDLE hKey,
5991     CK_BYTE_PTR pWrappedKey,
5992     CK ULONG_PTR pulWrappedKeyLen
5993 );

```

5994 **C_WrapKey** wraps (*i.e.*, encrypts) a private or secret key. *hSession* is the session's handle; *pMechanism* points to the wrapping mechanism; *hWrappingKey* is the handle of the wrapping key; *hKey* is the handle of the key to be wrapped; *pWrappedKey* points to the location that receives the wrapped key; and *pulWrappedKeyLen* points to the location that receives the length of the wrapped key.

5998 **C_WrapKey** uses the convention described in Section 5.2 on producing output.
 5999 The **CKA_WRAP** attribute of the wrapping key, which indicates whether the key supports wrapping,
 6000 MUST be CK_TRUE. The **CKA_EXTRACTABLE** attribute of the key to be wrapped MUST also be
 6001 CK_TRUE.
 6002 If the key to be wrapped cannot be wrapped for some token-specific reason, despite its having its
 6003 **CKA_EXTRACTABLE** attribute set to CK_TRUE, then **C_WrapKey** fails with error code
 6004 CKR_KEY_NOT_WRAPPABLE. If it cannot be wrapped with the specified wrapping key and mechanism
 6005 solely because of its length, then **C_WrapKey** fails with error code CKR_KEY_SIZE_RANGE.
 6006 **C_WrapKey** can be used in the following situations:
 6007

- To wrap any secret key with a public key that supports encryption and decryption.
- To wrap any secret key with any other secret key. Consideration MUST be given to key size and mechanism strength or the token may not allow the operation.
- To wrap a private key with any secret key.

 6008 Of course, tokens vary in which types of keys can actually be wrapped with which mechanisms.
 6009 To partition the wrapping keys so they can only wrap a subset of extractable keys the attribute
 6010 **CKA_WRAP_TEMPLATE** can be used on the wrapping key to specify an attribute set that will be
 6011 compared against the attributes of the key to be wrapped. If all attributes match according to the
 6012 **C_FindObject** rules of attribute matching then the wrap will proceed. The value of this attribute is an
 6013 attribute template and the size is the number of items in the template times the size of **CK_ATTRIBUTE**. If
 6014 this attribute is not supplied then any template is acceptable. If an attribute is not present, it will not be
 6015 checked. If any attribute mismatch occurs on an attempt to wrap a key then the function SHALL return
 6016 CKR_KEY_HANDLE_INVALID.
 6017 Return Values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
 6018 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
 6019 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
 6020 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_HANDLE_INVALID,
 6021 CKR_KEY_NOT_WRAPPABLE, CKR_KEY_SIZE_RANGE, CKR_KEY_UNEXTRACTABLE,
 6022 CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK,
 6023 CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
 6024 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
 6025 CKR_WRAPPER_KEY_HANDLE_INVALID, CKR_WRAPPER_KEY_SIZE_RANGE,
 6026 CKR_WRAPPER_KEY_TYPE_INCONSISTENT.
 6027 Example:
 6028


```

    6029 CK_SESSION_HANDLE hSession;
    6030 CK_OBJECT_HANDLE hWrappingKey, hKey;
    6031 CK_MECHANISM mechanism = {
    6032     CKM_DES3_ECB, NULL_PTR, 0
    6033 };
    6034 CK_BYTE wrappedKey[8];
    6035 CK ULONG ulWrappedKeyLen;
    6036 CK_RV rv;
    6037 .
    6038 .
    6039 ulWrappedKeyLen = sizeof(wrappedKey);
    6040 rv = C_WrapKey(
    6041     hSession, &mechanism,
    6042     hWrappingKey, hKey,
```

```
6046     wrappedKey, &ulWrappedKeyLen);  
6047     if (rv == CKR_OK) {  
6048         .  
6049         .  
6050     }
```

6051 5.18.4 C_UnwrapKey

```
6052 CK_DECLARE_FUNCTION(CK_RV, C_UnwrapKey) (  
6053     CK_SESSION_HANDLE hSession,  
6054     CK_MECHANISM_PTR pMechanism,  
6055     CK_OBJECT_HANDLE hUnwrappingKey,  
6056     CK_BYTE_PTR pWrappedKey,  
6057     CK ULONG ulWrappedKeyLen,  
6058     CK_ATTRIBUTE_PTR pTemplate,  
6059     CK ULONG ulAttributeCount,  
6060     CK_OBJECT_HANDLE_PTR phKey  
6061 );
```

6062 **C_UnwrapKey** unwraps (*i.e.* decrypts) a wrapped key, creating a new private key or secret key object.
6063 *hSession* is the session's handle; *pMechanism* points to the unwrapping mechanism; *hUnwrappingKey* is
6064 the handle of the unwrapping key; *pWrappedKey* points to the wrapped key; *ulWrappedKeyLen* is the
6065 length of the wrapped key; *pTemplate* points to the template for the new key; *ulAttributeCount* is the
6066 number of attributes in the template; *phKey* points to the location that receives the handle of the
6067 recovered key.

6068 The **CKA_UNWRAP** attribute of the unwrapping key, which indicates whether the key supports
6069 unwrapping, MUST be CK_TRUE.

6070 The new key will have the **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, and the
6071 **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE. The **CKA_EXTRACTABLE** attribute is by
6072 default set to CK_TRUE.

6073 Some mechanisms may modify, or attempt to modify, the contents of the *pMechanism* structure at the
6074 same time that the key is unwrapped.

6075 If a call to **C_UnwrapKey** cannot support the precise template supplied to it, it will fail and return without
6076 creating any key object.

6077 The key object created by a successful call to **C_UnwrapKey** will have its **CKA_LOCAL** attribute set to
6078 CK_FALSE. In addition, the object created will have a value for **CKA_UNIQUE_ID** generated and
6079 assigned (See Section 4.4.1).

6080 To partition the unwrapping keys so they can only unwrap a subset of keys the attribute
6081 **CKA_UNWRAP_TEMPLATE** can be used on the unwrapping key to specify an attribute set that will be
6082 added to attributes of the key to be unwrapped. If the attributes do not conflict with the user supplied
6083 attribute template, in 'pTemplate', then the unwrap will proceed. The value of this attribute is an attribute
6084 template and the size is the number of items in the template times the size of CK_ATTRIBUTE. If this
6085 attribute is not present on the unwrapping key then no additional attributes will be added. If any attribute
6086 conflict occurs on an attempt to unwrap a key then the function SHALL return
6087 CKR_TEMPLATE_INCONSISTENT.

6088 Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY,
6089 CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID,
6090 CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED,
6091 CKR_CURVE_NOT_SUPPORTED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
6092 CKR_DEVICE_REMOVED, CKR_DOMAIN_PARAMS_INVALID, CKR_FUNCTION_CANCELED,
6093 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
6094 CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK,
6095 CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
6096 CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCOMPLETE,

```
6097 CKR_TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED,  
6098 CKR_UNWRAPPING_KEY_HANDLE_INVALID, CKR_UNWRAPPING_KEY_SIZE_RANGE,  
6099 CKR_UNWRAPPING_KEY_TYPE_INCONSISTENT, CKR_USER_NOT_LOGGED_IN,  
6100 CKR_WWRAPPED_KEY_INVALID, CKR_WWRAPPED_KEY_LEN_RANGE.
```

6101 Example:

```
6102 CK_SESSION_HANDLE hSession;  
6103 CK_OBJECT_HANDLE hUnwrappingKey, hKey;  
6104 CK_MECHANISM mechanism = {  
6105     CKM_DES3_ECB, NULL_PTR, 0  
6106 };  
6107 CK_BYTE wrappedKey[8] = {...};  
6108 CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;  
6109 CK_KEY_TYPE keyType = CKK_DES;  
6110 CK_BBOOL true = CK_TRUE;  
6111 CK_ATTRIBUTE template[] = {  
6112     {CKA_CLASS, &keyClass, sizeof(keyClass)},  
6113     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},  
6114     {CKA_ENCRYPT, &true, sizeof(true)},  
6115     {CKA_DECRYPT, &true, sizeof(true)}  
6116 };  
6117 CK_RV rv;  
6118 .  
6119 .  
6120 .  
6121 rv = C_UnwrapKey(  
6122     hSession, &mechanism, hUnwrappingKey,  
6123     wrappedKey, sizeof(wrappedKey), template, 4, &hKey);  
6124 if (rv == CKR_OK) {  
6125     .  
6126     .  
6127 }
```

6128 5.18.5 C_DeriveKey

```
6129 CK_DECLARE_FUNCTION(CK_RV, C_DeriveKey) (  
6130     CK_SESSION_HANDLE hSession,  
6131     CK_MECHANISM_PTR pMechanism,  
6132     CK_OBJECT_HANDLE hBaseKey,  
6133     CK_ATTRIBUTE_PTR pTemplate,  
6134     CK_ULONG ulAttributeCount,  
6135     CK_OBJECT_HANDLE_PTR phKey  
6136 );
```

6137 **C_DeriveKey** derives a key from a base key, creating a new key object. *hSession* is the session's
6138 handle; *pMechanism* points to a structure that specifies the key derivation mechanism; *hBaseKey* is the
6139 handle of the base key; *pTemplate* points to the template for the new key; *ulAttributeCount* is the number
6140 of attributes in the template; and *phKey* points to the location that receives the handle of the derived key.

6141 The values of the **CKA_SENSITIVE**, **CKA_ALWAYS_SENSITIVE**, **CKA_EXTRACTABLE**, and
 6142 **CKA_NEVER_EXTRACTABLE** attributes for the base key affect the values that these attributes can hold
 6143 for the newly-derived key. See the description of each particular key-derivation mechanism in Section
 6144 5.21.2 for any constraints of this type.
 6145 If a call to **C_DeriveKey** cannot support the precise template supplied to it, it will fail and return without
 6146 creating any key object.
 6147 The key object created by a successful call to **C_DeriveKey** will have its **CKA_LOCAL** attribute set to
 6148 **CK_FALSE**. In addition, the object created will have a value for **CKA_UNIQUE_ID** generated and
 6149 assigned (See Section 4.4.1).
 6150 To partition the derivation keys so they can only derive a subset of keys the attribute
 6151 **CKA_DERIVE_TEMPLATE** can be used on the derivation keys to specify an attribute set that will be
 6152 added to attributes of the key to be derived. If the attributes do not conflict with the user supplied attribute
 6153 template, in 'pTemplate', then the derivation will proceed. The value of this attribute is an attribute
 6154 template and the size is the number of items in the template times the size of **CK_ATTRIBUTE**. If this
 6155 attribute is not present on the base derivation keys then no additional attributes will be added. If any
 6156 attribute conflict occurs on an attempt to derive a key then the function SHALL return
 6157 **CKR_TEMPLATE_INCONSISTENT**.
 6158 Return values: **CKR_ARGUMENTS_BAD**, **CKR_ATTRIBUTE_READ_ONLY**,
 6159 **CKR_ATTRIBUTE_TYPE_INVALID**, **CKR_ATTRIBUTE_VALUE_INVALID**,
 6160 **CKR_CRYPTOKI_NOT_INITIALIZED**, **CKR_CURVE_NOT_SUPPORTED**, **CKR_DEVICE_ERROR**,
 6161 **CKR_DEVICE_MEMORY**, **CKR_DEVICE_REMOVED**, **CKR_DOMAIN_PARAMS_INVALID**,
 6162 **CKR_FUNCTION_CANCELED**, **CKR_FUNCTION_FAILED**, **CKR_GENERAL_ERROR**,
 6163 **CKR_HOST_MEMORY**, **CKR_KEY_HANDLE_INVALID**, **CKR_KEY_SIZE_RANGE**,
 6164 **CKR_KEY_TYPE_INCONSISTENT**, **CKR_MECHANISM_INVALID**,
 6165 **CKR_MECHANISM_PARAM_INVALID**, **CKR_OK**, **CKR_OPERATION_ACTIVE**, **CKR_PIN_EXPIRED**,
 6166 **CKR_SESSION_CLOSED**, **CKR_SESSION_HANDLE_INVALID**, **CKR_SESSION_READ_ONLY**,
 6167 **CKR_TEMPLATE_INCOMPLETE**, **CKR_TEMPLATE_INCONSISTENT**,
 6168 **CKR_TOKEN_WRITE_PROTECTED**, **CKR_USER_NOT_LOGGED_IN**.

6169 Example:

```

6170 CK_SESSION_HANDLE hSession;
6171 CK_OBJECT_HANDLE hPublicKey, hPrivateKey, hKey;
6172 CK_MECHANISM keyPairMechanism = {
6173   CKM_DH_PKCS_KEY_PAIR_GEN, NULL_PTR, 0
6174 };
6175 CK_BYTE prime[] = {...};
6176 CK_BYTE base[] = {...};
6177 CK_BYTE publicValue[128];
6178 CK_BYTE otherPublicValue[128];
6179 CK_MECHANISM mechanism = {
6180   CKM_DH_PKCS_DERIVE, otherPublicValue, sizeof(otherPublicValue)
6181 };
6182 CK_ATTRIBUTE template[] = {
6183   {CKA_VALUE, &publicValue, sizeof(publicValue) }
6184 };
6185 CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;
6186 CK_KEY_TYPE keyType = CKK_DES;
6187 CK_BBOOL true = CK_TRUE;
6188 CK_ATTRIBUTE publicKeyTemplate[] = {

```

```

6189 { CKA_PRIME, prime, sizeof(prime) },
6190 { CKA_BASE, base, sizeof(base) }
6191 };
6192 CK_ATTRIBUTE privateKeyTemplate[] = {
6193 { CKA_DERIVE, &true, sizeof(true) }
6194 };
6195 CK_ATTRIBUTE derivedKeyTemplate[] = {
6196 { CKA_CLASS, &keyClass, sizeof(keyClass) },
6197 { CKA_KEY_TYPE, &keyType, sizeof(keyType) },
6198 { CKA_ENCRYPT, &true, sizeof(true) },
6199 { CKA_DECRYPT, &true, sizeof(true) }
6200 };
6201 CK_RV rv;
6202
6203 .
6204 .
6205 rv = C_GenerateKeyPair(
6206     hSession, &keyPairMechanism,
6207     publicKeyTemplate, 2,
6208     privateKeyTemplate, 1,
6209     &hPublicKey, &hPrivateKey);
6210 if (rv == CKR_OK) {
6211     rv = C_GetAttributeValue(hSession, hPublicKey, template, 1);
6212     if (rv == CKR_OK) {
6213         /* Put other guy's public value in otherPublicKey */
6214         .
6215         .
6216         rv = C_DeriveKey(
6217             hSession, &mechanism,
6218             hPrivateKey, derivedKeyTemplate, 4, &hKey);
6219         if (rv == CKR_OK) {
6220             .
6221             .
6222         }
6223     }
6224 }

```

6225 **5.19 Random number generation functions**

6226 Cryptoki provides the following functions for generating random numbers:

6227 **5.19.1 C_SeedRandom**

```

6228 CK_DECLARE_FUNCTION(CK_RV, C_SeedRandom) (
6229     CK_SESSION_HANDLE hSession,

```

```

6230     CK_BYTE_PTR pSeed,
6231     CK_ULONG ulSeedLen
6232 ) ;

```

6233 **C_SeedRandom** mixes additional seed material into the token's random number generator. *hSession* is
6234 the session's handle; *pSeed* points to the seed material; and *ulSeedLen* is the length in bytes of the seed
6235 material.

6236 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
6237 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
6238 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
6239 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE,
6240 CKR_RANDOM_SEED_NOT_SUPPORTED, CKR_RANDOM_NO_RNG, CKR_SESSION_CLOSED,
6241 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

6242 Example: see **C_GenerateRandom**.

6243 5.19.2 C_GenerateRandom

```

6244 CK_DECLARE_FUNCTION(CK_RV, C_GenerateRandom) (
6245     CK_SESSION_HANDLE hSession,
6246     CK_BYTE_PTR pRandomData,
6247     CK_ULONG ulRandomLen
6248 );

```

6249 **C_GenerateRandom** generates random or pseudo-random data. *hSession* is the session's handle;
6250 *pRandomData* points to the location that receives the random data; and *ulRandomLen* is the length in
6251 bytes of the random or pseudo-random data to be generated.

6252 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
6253 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
6254 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
6255 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_RANDOM_NO_RNG,
6256 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

6257 Example:

```

6258 CK_SESSION_HANDLE hSession;
6259 CK_BYTE seed[] = {...};
6260 CK_BYTE randomData[] = {...};
6261 CK_RV rv;
6262
6263 .
6264 .
6265 rv = C_SeedRandom(hSession, seed, sizeof(seed));
6266 if (rv != CKR_OK) {
6267     .
6268     .
6269 }
6270 rv = C_GenerateRandom(hSession, randomData, sizeof(randomData));
6271 if (rv == CKR_OK) {
6272     .
6273     .
6274 }

```

6275 5.20 Parallel function management functions

6276 Cryptoki provides the following functions for managing parallel execution of cryptographic functions.
6277 These functions exist only for backwards compatibility.

6278 5.20.1 C_GetFunctionStatus

```
6279 CK_DECLARE_FUNCTION(CK_RV, C_GetFunctionStatus) (
6280     CK_SESSION_HANDLE hSession
6281 );
```

6282 In previous versions of Cryptoki, **C_GetFunctionStatus** obtained the status of a function running in
6283 parallel with an application. Now, however, **C_GetFunctionStatus** is a legacy function which should
6284 simply return the value CKR_FUNCTION_NOT_PARALLEL.

6285 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_FUNCTION_FAILED,
6286 CKR_FUNCTION_NOT_PARALLEL, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
6287 CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED.

6288 5.20.2 C_CancelFunction

```
6289 CK_DECLARE_FUNCTION(CK_RV, C_CancelFunction) (
6290     CK_SESSION_HANDLE hSession
6291 );
```

6292 In previous versions of Cryptoki, **C_CancelFunction** cancelled a function running in parallel with an
6293 application. Now, however, **C_CancelFunction** is a legacy function which should simply return the value
6294 CKR_FUNCTION_NOT_PARALLEL.

6295 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_FUNCTION_FAILED,
6296 CKR_FUNCTION_NOT_PARALLEL, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
6297 CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED.

6298 5.21 Callback functions

6299 Cryptoki sessions can use function pointers of type **CK_NOTIFY** to notify the application of certain
6300 events.

6301 5.21.1 Surrender callbacks

6302 Cryptographic functions (*i.e.*, any functions falling under one of these categories: encryption functions;
6303 decryption functions; message digesting functions; signing and MACing functions; functions for verifying
6304 signatures and MACs; dual-purpose cryptographic functions; key management functions; random number
6305 generation functions) executing in Cryptoki sessions can periodically surrender control to the application
6306 who called them if the session they are executing in had a notification callback function associated with it
6307 when it was opened. They do this by calling the session's callback with the arguments (hSession,
6308 CKN_SURRENDER, pApplication), where hSession is the session's handle and pApplication was
6309 supplied to **C_OpenSession** when the session was opened. Surrender callbacks should return either the
6310 value CKR_OK (to indicate that Cryptoki should continue executing the function) or the value
6311 CKR_CANCEL (to indicate that Cryptoki should abort execution of the function). Of course, before
6312 returning one of these values, the callback function can perform some computation, if desired.

6313 A typical use of a surrender callback might be to give an application user feedback during a lengthy key
6314 pair generation operation. Each time the application receives a callback, it could display an additional “.”
6315 to the user. It might also examine the keyboard's activity since the last surrender callback, and abort the
6316 key pair generation operation (probably by returning the value CKR_CANCEL) if the user hit <ESCAPE>.

6317 A Cryptoki library is not *required* to make any surrender callbacks.

6318 **5.21.2 Vendor-defined callbacks**

6319 Library vendors can also define additional types of callbacks. Because of this extension capability,
6320 application-supplied notification callback routines should examine each callback they receive, and if they
6321 are unfamiliar with the type of that callback, they should immediately give control back to the library by
6322 returning with the value CKR_OK.

6 Mechanisms

6.1 RSA

Table 32, Mechanisms vs. Functions

| Mechanism | Functions | | | | | | |
|----------------------------|-------------------|----------------|----------------------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_RSA_PKCS_KEY_PAIR_GEN | | | | | ✓ | | |
| CKM_RSA_X9_31_KEY_PAIR_GEN | | | | | ✓ | | |
| CKM_RSA_PKCS | ✓ ² | ✓ ² | ✓ | | | ✓ | |
| CKM_RSA_PKCS_OAEP | ✓ ² | | | | | ✓ | |
| CKM_RSA_PKCS_PSS | | ✓ ² | | | | | |
| CKM_RSA_9796 | | ✓ ² | ✓ | | | | |
| CKM_RSA_X_509 | ✓ ² | ✓ ² | ✓ | | | ✓ | |
| CKM_RSA_X9_31 | | ✓ ² | | | | | |
| CKM_SHA1_RSA_PKCS | | ✓ | | | | | |
| CKM_SHA224_RSA_PKCS | | ✓ | | | | | |
| CKM_SHA256_RSA_PKCS | | ✓ | | | | | |
| CKM_SHA384_RSA_PKCS | | ✓ | | | | | |
| CKM_SHA512_RSA_PKCS | | ✓ | | | | | |
| CKM_SHA1_RSA_PKCS_PSS | | ✓ | | | | | |
| CKM_SHA224_RSA_PKCS_PSS | | ✓ | | | | | |
| CKM_SHA256_RSA_PKCS_PSS | | ✓ | | | | | |
| CKM_SHA384_RSA_PKCS_PSS | | ✓ | | | | | |
| CKM_SHA512_RSA_PKCS_PSS | | ✓ | | | | | |
| CKM_SHA1_RSA_X9_31 | | ✓ | | | | | |
| CKM_RSA_PKCS TPM_1_1 | ✓ ² | | | | | ✓ | |
| CKM_RSA_PKCS_OAEP TPM_1_1 | ✓ ² | | | | | ✓ | |
| CKM_SHA3_224_RSA_PKCS | | ✓ | | | | | |
| CKM_SHA3_256_RSA_PKCS | | ✓ | | | | | |
| CKM_SHA3_384_RSA_PKCS | | ✓ | | | | | |
| CKM_SHA3_512_RSA_PKCS | | ✓ | | | | | |
| CKM_SHA3_224_RSA_PKCS_PSS | | ✓ | | | | | |
| CKM_SHA3_256_RSA_PKCS_PSS | | ✓ | | | | | |
| CKM_SHA3_384_RSA_PKCS_PSS | | ✓ | | | | | |
| CKM_SHA3_512_RSA_PKCS_PSS | | ✓ | | | | | |

6.1.1 Definitions

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

Mechanisms:

CKM_RSA_PKCS_KEY_PAIR_GEN

CKM_RSA_PKCS

CKM_RSA_9796

6333 CKM_RSA_X_509
6334 CKM_MD2_RSA_PKCS
6335 CKM_MD5_RSA_PKCS
6336 CKM_SHA1_RSA_PKCS
6337 CKM_SHA224_RSA_PKCS
6338 CKM_SHA256_RSA_PKCS
6339 CKM_SHA384_RSA_PKCS
6340 CKM_SHA512_RSA_PKCS
6341 CKM_RIPEMD128_RSA_PKCS
6342 CKM_RIPEMD160_RSA_PKCS
6343 CKM_RSA_PKCS_OAEP
6344 CKM_RSA_X9_31_KEY_PAIR_GEN
6345 CKM_RSA_X9_31
6346 CKM_SHA1_RSA_X9_31
6347 CKM_RSA_PKCS_PSS
6348 CKM_SHA1_RSA_PKCS_PSS
6349 CKM_SHA224_RSA_PKCS_PSS
6350 CKM_SHA256_RSA_PKCS_PSS
6351 CKM_SHA512_RSA_PKCS_PSS
6352 CKM_SHA384_RSA_PKCS_PSS
6353 CKM_RSA_PKCS TPM_1_1
6354 CKM_RSA_PKCS_OAEP TPM_1_1
6355 CKM_RSA_AES_KEY_WRAP
6356 CKM_SHA3_224_RSA_PKCS
6357 CKM_SHA3_256_RSA_PKCS
6358 CKM_SHA3_384_RSA_PKCS
6359 CKM_SHA3_512_RSA_PKCS
6360 CKM_SHA3_224_RSA_PKCS_PSS
6361 CKM_SHA3_256_RSA_PKCS_PSS
6362 CKM_SHA3_384_RSA_PKCS_PSS
6363 CKM_SHA3_512_RSA_PKCS_PSS
6364

6365 6.1.2 RSA public key objects

6366 RSA public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_RSA**) hold RSA public keys.
6367 The following table defines the RSA public key object attributes, in addition to the common attributes
6368 defined for this object class:

6369 *Table 33, RSA Public Key Object Attributes*

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

6370 - Refer to Table 11 for footnotes

6371 Depending on the token, there may be limits on the length of key components. See PKCS #1 for more
6372 information on RSA keys.

6373 The following is a sample template for creating an RSA public key object:

```
6374     CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;  
6375     CK_KEY_TYPE keyType = CKK_RSA;  
6376     CK_UTF8CHAR label[] = "An RSA public key object";  
6377     CK_BYTE modulus[] = {...};  
6378     CK_BYTE exponent[] = {...};  
6379     CK_BBOOL true = CK_TRUE;  
6380     CK_ATTRIBUTE template[] = {  
6381         {CKA_CLASS, &class, sizeof(class)},  
6382         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},  
6383         {CKA_TOKEN, &true, sizeof(true)},  
6384         {CKA_LABEL, label, sizeof(label)-1},  
6385         {CKA_WRAP, &true, sizeof(true)},  
6386         {CKA_ENCRYPT, &true, sizeof(true)},  
6387         {CKA_MODULUS, modulus, sizeof(modulus)},  
6388         {CKA_PUBLIC_EXPONENT, exponent, sizeof(exponent)}  
6389     };
```

6390 6.1.3 RSA private key objects

6391 RSA private key objects (object class **CKO_PRIVATE_KEY**, key type **CKK_RSA**) hold RSA private keys.
6392 The following table defines the RSA private key object attributes, in addition to the common attributes
6393 defined for this object class:

6394 *Table 34, RSA Private Key Object Attributes*

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

6395 ¹Refer to Table 11 for footnotes

6396 Depending on the token, there may be limits on the length of the key components. See PKCS #1 for
6397 more information on RSA keys.

6398 Tokens vary in what they actually store for RSA private keys. Some tokens store all of the above
6399 attributes, which can assist in performing rapid RSA computations. Other tokens might store only the
6400 **CKA_MODULUS** and **CKA_PRIVATE_EXPONENT** values. Effective with version 2.40, tokens MUST
6401 also store **CKA_PUBLIC_EXPONENT**. This permits the retrieval of sufficient data to reconstitute the
6402 associated public key.

6403 Because of this, Cryptoki is flexible in dealing with RSA private key objects. When a token generates an
6404 RSA private key, it stores whichever of the fields in Table 34 it keeps track of. Later, if an application
6405 asks for the values of the key's various attributes, Cryptoki supplies values only for attributes whose
6406 values it can obtain (*i.e.*, if Cryptoki is asked for the value of an attribute it cannot obtain, the request
6407 fails). Note that a Cryptoki implementation may or may not be able and/or willing to supply various
6408 attributes of RSA private keys which are not actually stored on the token. *E.g.*, if a particular token stores

6409 values only for the **CKA_PRIVATE_EXPONENT**, **CKA_PRIME_1**, and **CKA_PRIME_2** attributes, then
6410 Cryptoki is certainly *able* to report values for all the attributes above (since they can all be computed
6411 efficiently from these three values). However, a Cryptoki implementation may or may not actually do this
6412 extra computation. The only attributes from Table 34 for which a Cryptoki implementation is *required* to
6413 be able to return values are **CKA_MODULUS**, **CKA_PUBLIC_EXPONENT** and
6414 **CKA_PRIVATE_EXPONENT**. A token SHOULD also be able to return **CKA_PUBLIC_KEY_INFO** for an
6415 RSA private key.

6416 If an RSA private key object is created on a token, and more attributes from Table 34 are supplied to the
6417 object creation call than are supported by the token, the extra attributes are likely to be thrown away. If
6418 an attempt is made to create an RSA private key object on a token with insufficient attributes for that
6419 particular token, then the object creation call fails and returns CKR_TEMPLATE_INCOMPLETE.

6420 Note that when generating an RSA private key, there is no **CKA_MODULUS_BITS** attribute specified.
6421 This is because RSA private keys are only generated as part of an RSA key pair, and the
6422 **CKA_MODULUS_BITS** attribute for the pair is specified in the template for the RSA public key.

6423 The following is a sample template for creating an RSA private key object:

```
6424     CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
6425     CK_KEY_TYPE keyType = CKK_RSA;
6426     CK_UTF8CHAR label[] = "An RSA private key object";
6427     CK_BYTE subject[] = {...};
6428     CK_BYTE id[] = {123};
6429     CK_BYTE modulus[] = {...};
6430     CK_BYTE publicExponent[] = {...};
6431     CK_BYTE privateExponent[] = {...};
6432     CK_BYTE prime1[] = {...};
6433     CK_BYTE prime2[] = {...};
6434     CK_BYTE exponent1[] = {...};
6435     CK_BYTE exponent2[] = {...};
6436     CK_BYTE coefficient[] = {...};
6437     CK_BBOOL true = CK_TRUE;
6438     CK_ATTRIBUTE template[] = {
6439         {CKA_CLASS, &class, sizeof(class)},
6440         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
6441         {CKA_TOKEN, &true, sizeof(true)},
6442         {CKA_LABEL, label, sizeof(label)-1},
6443         {CKA_SUBJECT, subject, sizeof(subject)},
6444         {CKA_ID, id, sizeof(id)},
6445         {CKA_SENSITIVE, &true, sizeof(true)},
6446         {CKA_DECRYPT, &true, sizeof(true)},
6447         {CKA_SIGN, &true, sizeof(true)},
6448         {CKA_MODULUS, modulus, sizeof(modulus)},
6449         {CKA_PUBLIC_EXPONENT, publicExponent,
6450             sizeof(publicExponent)},
6451         {CKA_PRIVATE_EXPONENT, privateExponent,
6452             sizeof(privateExponent)},
6453         {CKA_PRIME_1, prime1, sizeof(prime1)},
6454         {CKA_PRIME_2, prime2, sizeof(prime2)},
6455         {CKA_EXPONENT_1, exponent1, sizeof(exponent1)},
6456         {CKA_EXPONENT_2, exponent2, sizeof(exponent2)},
6457         {CKA_COEFFICIENT, coefficient, sizeof(coefficient)} }
```

6458 } ;

6459 6.1.4 PKCS #1 RSA key pair generation

6460 The PKCS #1 RSA key pair generation mechanism, denoted **CKM_RSA_PKCS_KEY_PAIR_GEN**, is a
6461 key pair generation mechanism based on the RSA public-key cryptosystem, as defined in PKCS #1.

6462 It does not have a parameter.

6463 The mechanism generates RSA public/private key pairs with a particular modulus length in bits and public
6464 exponent, as specified in the **CKA_MODULUS_BITS** and **CKA_PUBLIC_EXPONENT** attributes of the
6465 template for the public key. The **CKA_PUBLIC_EXPONENT** may be omitted in which case the
6466 mechanism shall supply the public exponent attribute using the default value of 0x10001 (65537).
6467 Specific implementations may use a random value or an alternative default if 0x10001 cannot be used by
6468 the token.

6469 Note: Implementations strictly compliant with version 2.11 or prior versions may generate an error
6470 if this attribute is omitted from the template. Experience has shown that many implementations of 2.11
6471 and prior did allow the **CKA_PUBLIC_EXPONENT** attribute to be omitted from the template, and
6472 behaved as described above. The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**,
6473 **CKA_MODULUS**, and **CKA_PUBLIC_EXPONENT** attributes to the new public key.
6474 **CKA_PUBLIC_EXPONENT** will be copied from the template if supplied.

6475 **CKR_TEMPLATE_INCONSISTENT** shall be returned if the implementation cannot use the supplied
6476 exponent value. It contributes the **CKA_CLASS** and **CKA_KEY_TYPE** attributes to the new private key; it
6477 may also contribute some of the following attributes to the new private key: **CKA_MODULUS**,
6478 **CKA_PUBLIC_EXPONENT**, **CKA_PRIVATE_EXPONENT**, **CKA_PRIME_1**, **CKA_PRIME_2**,
6479 **CKA_EXPONENT_1**, **CKA_EXPONENT_2**, **CKA_COEFFICIENT**. Other attributes supported by the
6480 RSA public and private key types (specifically, the flags indicating which functions the keys support) may
6481 also be specified in the templates for the keys, or else are assigned default initial values.

6482 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
6483 specify the supported range of RSA modulus sizes, in bits.

6484 6.1.5 X9.31 RSA key pair generation

6485 The X9.31 RSA key pair generation mechanism, denoted **CKM_RSA_X9_31_KEY_PAIR_GEN**, is a key
6486 pair generation mechanism based on the RSA public-key cryptosystem, as defined in X9.31.

6487 It does not have a parameter.

6488 The mechanism generates RSA public/private key pairs with a particular modulus length in bits and public
6489 exponent, as specified in the **CKA_MODULUS_BITS** and **CKA_PUBLIC_EXPONENT** attributes of the
6490 template for the public key.

6491 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_MODULUS**, and
6492 **CKA_PUBLIC_EXPONENT** attributes to the new public key. It contributes the **CKA_CLASS** and
6493 **CKA_KEY_TYPE** attributes to the new private key; it may also contribute some of the following attributes
6494 to the new private key: **CKA_MODULUS**, **CKA_PUBLIC_EXPONENT**, **CKA_PRIVATE_EXPONENT**,
6495 **CKA_PRIME_1**, **CKA_PRIME_2**, **CKA_EXPONENT_1**, **CKA_EXPONENT_2**, **CKA_COEFFICIENT**.

6496 Other attributes supported by the RSA public and private key types (specifically, the flags indicating which
6497 functions the keys support) may also be specified in the templates for the keys, or else are assigned
6498 default initial values. Unlike the **CKM_RSA_PKCS_KEY_PAIR_GEN** mechanism, this mechanism is
6499 guaranteed to generate *p* and *q* values, **CKA_PRIME_1** and **CKA_PRIME_2** respectively, that meet the
6500 strong primes requirement of X9.31.

6501 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
6502 specify the supported range of RSA modulus sizes, in bits.

6503 6.1.6 PKCS #1 v1.5 RSA

6504 The PKCS #1 v1.5 RSA mechanism, denoted **CKM_RSA_PKCS**, is a multi-purpose mechanism based
6505 on the RSA public-key cryptosystem and the block formats initially defined in PKCS #1 v1.5. It supports

6506 single-part encryption and decryption; single-part signatures and verification with and without message
6507 recovery; key wrapping; and key unwrapping. This mechanism corresponds only to the part of PKCS #1
6508 v1.5 that involves RSA; it does not compute a message digest or a DigestInfo encoding as specified for
6509 the md2withRSAEncryption and md5withRSAEncryption algorithms in PKCS #1 v1.5 .

6510 This mechanism does not have a parameter.

6511 This mechanism can wrap and unwrap any secret key of appropriate length. Of course, a particular token
6512 may not be able to wrap/unwrap every appropriate-length secret key that it supports. For wrapping, the
6513 "input" to the encryption operation is the value of the **CKA_VALUE** attribute of the key that is wrapped;
6514 similarly for unwrapping. The mechanism does not wrap the key type or any other information about the
6515 key, except the key length; the application must convey these separately. In particular, the mechanism
6516 contributes only the **CKA_CLASS** and **CKA_VALUE** (and **CKA_VALUE_LEN**, if the key has it) attributes
6517 to the recovered key during unwrapping; other attributes must be specified in the template.

6518 Constraints on key types and the length of the data are summarized in the following table. For
6519 encryption, decryption, signatures and signature verification, the input and output data may begin at the
6520 same location in memory. In the table, k is the length in bytes of the RSA modulus.

6521 *Table 35, PKCS #1 v1.5 RSA: Key And Data Length*

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

6522 ¹ Single-part operations only.

6523 ² Data length, signature length.

6524 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
6525 specify the supported range of RSA modulus sizes, in bits.

6526 **6.1.7 PKCS #1 RSA OAEP mechanism parameters**

6527 ♦ **CK_RSA_PKCS_MGF_TYPE; CK_RSA_PKCS_MGF_TYPE_PTR**

6528 **CK_RSA_PKCS_MGF_TYPE** is used to indicate the Mask Generation Function (MGF) applied to a
6529 message block when formatting a message block for the PKCS #1 OAEP encryption scheme or the
6530 PKCS #1 PSS signature scheme. It is defined as follows:

6531 **typedef CK ULONG CK_RSA_PKCS_MGF_TYPE;**

6533 The following MGFs are defined in PKCS #1. The following table lists the defined functions.

6534 *Table 36, PKCS #1 Mask Generation Functions*

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

6535 **CK_RSA_PKCS_MGF_TYPE_PTR** is a pointer to a **CK_RSA_PKCS_MGF_TYPE**.

6536 ◆ **CK_RSA_PKCS_OAEP_SOURCE_TYPE;**
 6537 **CK_RSA_PKCS_OAEP_SOURCE_TYPE_PTR**

6538 **CK_RSA_PKCS_OAEP_SOURCE_TYPE** is used to indicate the source of the encoding parameter
 6539 when formatting a message block for the PKCS #1 OAEP encryption scheme. It is defined as follows:

6540 **typedef CK ULONG CK_RSA_PKCS_OAEP_SOURCE_TYPE;**

6541

6542 The following encoding parameter sources are defined in PKCS #1. The following table lists the defined
 6543 sources along with the corresponding data type for the *pSourceData* field in the
 6544 **CK_RSA_PKCS_OAEP_PARAMS** structure defined below.

6545 *Table 37, PKCS #1 RSA OAEP: Encoding parameter sources*

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

6546 **CK_RSA_PKCS_OAEP_SOURCE_TYPE_PTR** is a pointer to a

6547 **CK_RSA_PKCS_OAEP_SOURCE_TYPE**.

6548 ◆ **CK_RSA_PKCS_OAEP_PARAMS; CK_RSA_PKCS_OAEP_PARAMS_PTR**

6549 **CK_RSA_PKCS_OAEP_PARAMS** is a structure that provides the parameters to the
 6550 **CKM_RSA_PKCS_OAEP** mechanism. The structure is defined as follows:

```
6551   typedef struct CK_RSA_PKCS_OAEP_PARAMS {
 6552     CK_MECHANISM_TYPE                           hashAlg;
 6553     CK_RSA_PKCS_MGF_TYPE                       mgf;
 6554     CK_RSA_PKCS_OAEP_SOURCE_TYPE           source;
 6555     CK_VOID_PTR                               pSourceData;
 6556     CK ULONG                                  ulSourceDataLen;
 6557 } CK_RSA_PKCS_OAEP_PARAMS;
```

6558

6559 The fields of the structure have the following meanings:

6560 hashAlg mechanism ID of the message digest algorithm used to calculate
 6561 the digest of the encoding parameter

6562 mgf mask generation function to use on the encoded block

6563 source source of the encoding parameter
 6564 pSourceData data used as the input for the encoding parameter source
 6565 ulSourceDataLen length of the encoding parameter source input
 6566 **CK_RSA_PKCS_OAEP_PARAMS_PTR** is a pointer to a **CK_RSA_PKCS_OAEP_PARAMS**.
 6567

6568 6.1.8 PKCS #1 RSA OAEP

6569 The PKCS #1 RSA OAEP mechanism, denoted **CKM_RSA_PKCS_OAEP**, is a multi-purpose
 6570 mechanism based on the RSA public-key cryptosystem and the OAEP block format defined in PKCS #1.
 6571 It supports single-part encryption and decryption; key wrapping; and key unwrapping.
 6572 It has a parameter, a **CK_RSA_PKCS_OAEP_PARAMS** structure.
 6573 This mechanism can wrap and unwrap any secret key of appropriate length. Of course, a particular token
 6574 may not be able to wrap/unwrap every appropriate-length secret key that it supports. For wrapping, the
 6575 “input” to the encryption operation is the value of the **CKA_VALUE** attribute of the key that is wrapped;
 6576 similarly for unwrapping. The mechanism does not wrap the key type or any other information about the
 6577 key, except the key length; the application must convey these separately. In particular, the mechanism
 6578 contributes only the **CKA_CLASS** and **CKA_VALUE** (and **CKA_VALUE_LEN**, if the key has it) attributes
 6579 to the recovered key during unwrapping; other attributes must be specified in the template.
 6580 Constraints on key types and the length of the data are summarized in the following table. For encryption
 6581 and decryption, the input and output data may begin at the same location in memory. In the table, k is the
 6582 length in bytes of the RSA modulus, and $hLen$ is the output length of the message digest algorithm
 6583 specified by the *hashAlg* field of the **CK_RSA_PKCS_OAEP_PARAMS** structure.
 6584

Table 38, PKCS #1 RSA OAEP: Key And Data Length

| Function | Key type | Input length | Output length |
|------------------------|-----------------|------------------|------------------|
| C_Encrypt ¹ | RSA public key | $\leq k-2-2hLen$ | 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 | $\leq k-2-2hLen$ |

6585 ¹ Single-part operations only.

6586 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 6587 specify the supported range of RSA modulus sizes, in bits.

6588 6.1.9 PKCS #1 RSA PSS mechanism parameters

6589 ◆ **CK_RSA_PKCS_PSS_PARAMS; CK_RSA_PKCS_PSS_PARAMS_PTR**

6590 **CK_RSA_PKCS_PSS_PARAMS** is a structure that provides the parameters to the
 6591 **CKM_RSA_PKCS_PSS** mechanism. The structure is defined as follows:

```

    6592       typedef struct CK_RSA_PKCS_PSS_PARAMS {
    6593           CK_MECHANISM_TYPE     hashAlg;
    6594           CK_RSA_PKCS_MGF_TYPE mgf;
    6595           CK ULONG            sLen;
    6596       } CK_RSA_PKCS_PSS_PARAMS;
  
```

6597 The fields of the structure have the following meanings:

6599 hashAlg hash algorithm used in the PSS encoding; if the signature
 6600 mechanism does not include message hashing, then this value must
 6601 be the mechanism used by the application to generate the message
 6602 hash; if the signature mechanism includes hashing, then this value
 6603 must match the hash algorithm indicated by the signature
 6604 mechanism
 6605 mgf mask generation function to use on the encoded block
 6606 sLen length, in bytes, of the salt value used in the PSS encoding; typical
 6607 values are the length of the message hash and zero

6608 **CK_RSA_PKCS_PSS_PARAMS_PTR** is a pointer to a **CK_RSA_PKCS_PSS_PARAMS**.

6.1.10 PKCS #1 RSA PSS

6610 The PKCS #1 RSA PSS mechanism, denoted **CKM_RSA_PKCS_PSS**, is a mechanism based on the
 6611 RSA public-key cryptosystem and the PSS block format defined in PKCS #1. It supports single-part
 6612 signature generation and verification without message recovery. This mechanism corresponds only to the
 6613 part of PKCS #1 that involves block formatting and RSA, given a hash value; it does not compute a hash
 6614 value on the message to be signed.

6615 It has a parameter, a **CK_RSA_PKCS_PSS_PARAMS** structure. The *sLen* field must be less than or
 6616 equal to $k^* - 2 \cdot hLen$ and *hLen* is the length of the input to the C_Sign or C_Verify function. k^* is the length
 6617 in bytes of the RSA modulus, except if the length in bits of the RSA modulus is one more than a multiple
 6618 of 8, in which case k^* is one less than the length in bytes of the RSA modulus.

6619 Constraints on key types and the length of the data are summarized in the following table. In the table, *k*
 6620 is the length in bytes of the RSA.

6621 *Table 39, PKCS #1 RSA PSS: Key And Data Length*

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

6622 ¹ Single-part operations only.

6623 ² Data length, signature length.

6624 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 6625 specify the supported range of RSA modulus sizes, in bits.

6.1.11 ISO/IEC 9796 RSA

6627 The ISO/IEC 9796 RSA mechanism, denoted **CKM_RSA_9796**, is a mechanism for single-part
 6628 signatures and verification with and without message recovery based on the RSA public-key
 6629 cryptosystem and the block formats defined in ISO/IEC 9796 and its annex A.

6630 This mechanism processes only byte strings, whereas ISO/IEC 9796 operates on bit strings. Accordingly,
 6631 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.

6640 This mechanism does not have a parameter.

6641 Constraints on key types and the length of input and output data are summarized in the following table.
6642 In the table, k is the length in bytes of the RSA modulus.

6643 *Table 40, ISO/IEC 9796 RSA: Key And Data Length*

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

6644 ¹ Single-part operations only.

6645 ² Data length, signature length.

6646 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
6647 specify the supported range of RSA modulus sizes, in bits.

6648 **6.1.12 X.509 (raw) RSA**

6649 The X.509 (raw) RSA mechanism, denoted **CKM_RSA_X_509**, is a multi-purpose mechanism based on
6650 the RSA public-key cryptosystem. It supports single-part encryption and decryption; single-part signatures
6651 and verification with and without message recovery; key wrapping; and key unwrapping. All these
6652 operations are based on so-called “raw” RSA, as assumed in X.509.

6653 “Raw” RSA as defined here encrypts a byte string by converting it to an integer, most-significant byte first,
6654 applying “raw” RSA exponentiation, and converting the result to a byte string, most-significant byte first.
6655 The input string, considered as an integer, must be less than the modulus; the output string is also less
6656 than the modulus.

6657 This mechanism does not have a parameter.

6658 This mechanism can wrap and unwrap any secret key of appropriate length. Of course, a particular token
6659 may not be able to wrap/unwrap every appropriate-length secret key that it supports. For wrapping, the
6660 “input” to the encryption operation is the value of the **CKA_VALUE** attribute of the key that is wrapped;
6661 similarly for unwrapping. The mechanism does not wrap the key type, key length, or any other
6662 information about the key; the application must convey these separately, and supply them when
6663 unwrapping the key.

6664 Unfortunately, X.509 does not specify how to perform padding for RSA encryption. For this mechanism,
6665 padding should be performed by prepending plaintext data with 0-valued bytes. In effect, to encrypt the
6666 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
6667 be less than the RSA modulus. The k -byte ciphertext (k is the length in bytes of the RSA modulus) is
6668 produced by raising P to the RSA public exponent modulo the RSA modulus. Decryption of a k -byte
6669 ciphertext C is accomplished by raising C to the RSA private exponent modulo the RSA modulus, and
6670 returning the resulting value as a sequence of exactly k bytes. If the resulting plaintext is to be used to
6671 produce an unwrapped key, then however many bytes are specified in the template for the length of the
6672 key are taken *from the end* of this sequence of bytes.

6673 Technically, the above procedures may differ very slightly from certain details of what is specified in
6674 X.509.

6675 Executing cryptographic operations using this mechanism can result in the error returns
6676 **CKR_DATA_INVALID** (if plaintext is supplied which has the same length as the RSA modulus and is
6677 numerically at least as large as the modulus) and **CKR_ENCRYPTED_DATA_INVALID** (if ciphertext is
6678 supplied which has the same length as the RSA modulus and is numerically at least as large as the
6679 modulus).

6680 Constraints on key types and the length of input and output data are summarized in the following table.
6681 In the table, k is the length in bytes of the RSA modulus.

6682 *Table 41, X.509 (Raw) RSA: Key And Data Length*

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

6683 1 Single-part operations only.

6684 2 Data length, signature length.

6685 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
6686 specify the supported range of RSA modulus sizes, in bits.

6687 This mechanism is intended for compatibility with applications that do not follow the PKCS #1 or ISO/IEC
6688 9796 block formats.

6689 6.1.13 ANSI X9.31 RSA

6690 The ANSI X9.31 RSA mechanism, denoted **CKM_RSA_X9_31**, is a mechanism for single-part signatures
6691 and verification without message recovery based on the RSA public-key cryptosystem and the block
6692 formats defined in ANSI X9.31.

6693 This mechanism applies the header and padding fields of the hash encapsulation. The trailer field must
6694 be applied by the application.

6695 This mechanism processes only byte strings, whereas ANSI X9.31 operates on bit strings. Accordingly,
6696 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.

6705 This mechanism does not have a parameter.

6706 Constraints on key types and the length of input and output data are summarized in the following table.
6707 In the table, k is the length in bytes of the RSA modulus. For all operations, the k value must be at least
6708 128 and a multiple of 32 as specified in ANSI X9.31.

6709 Table 42, ANSI X9.31 RSA: Key And Data Length

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

6710 1 Single-part operations only.

6711 2 Data length, signature length.

6712 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
6713 specify the supported range of RSA modulus sizes, in bits.

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

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

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

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

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

None of these mechanisms has a parameter.

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

Table 43, PKCS #1 v1.5 RSA Signatures with Various Hash Functions: Key And Data Length

| Function | Key type | Input length | Output length | Comments |
|----------|-----------------|--------------|---------------|---------------|
| C_Sign | RSA private key | any | k | block type 01 |
| C_Verify | RSA public key | any, k^2 | N/A | block type 01 |

2 Data length, signature length.

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

6.1.15 PKCS #1 v1.5 RSA signature with SHA-224

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

6.1.16 PKCS #1 RSA PSS signature with SHA-224

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

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

The PKCS #1 RSA PSS signature with SHA-1 mechanism, denoted **CKM_SHA1_RSA_PKCS_PSS**, performs single- and multiple-part digital signatures and verification operations without message

6756 recovery. The operations performed are as described in PKCS #1 with the object identifier id-RSASSA-
6757 PSS, i.e., as in the scheme RSASSA-PSS in PKCS #1 where the underlying hash function is SHA-1.

6758 The PKCS #1 RSA PSS signature with SHA-256, SHA-384, and SHA-512 mechanisms, denoted
6759 **CKM_SHA256_RSA_PKCS_PSS**, **CKM_SHA384_RSA_PKCS_PSS**, and
6760 **CKM_SHA512_RSA_PKCS_PSS** respectively, perform the same operations using the SHA-256, SHA-
6761 384 and SHA-512 hash functions.

6762 The mechanisms have a parameter, a **CK_RSA_PKCS_PSS_PARAMS** structure. The *sLen* field must
6763 be less than or equal to $k^* - hLen$ where *hLen* is the length in bytes of the hash value. k^* is the length in
6764 bytes of the RSA modulus, except if the length in bits of the RSA modulus is one more than a multiple of
6765 8, in which case k^* is one less than the length in bytes of the RSA modulus.

6766 Constraints on key types and the length of the data are summarized in the following table. In the table, *k*
6767 is the length in bytes of the RSA modulus.

6768 *Table 44, PKCS #1 RSA PSS Signatures with Various Hash Functions: Key And Data Length*

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

6769 2 Data length, signature length.

6770 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
6771 specify the supported range of RSA modulus sizes, in bits.

6772 **6.1.18 PKCS #1 v1.5 RSA signature with SHA3**

6773 The PKCS #1 v1.5 RSA signature with SHA3-224, SHA3-256, SHA3-384, SHA3-512 mechanisms,
6774 denoted **CKM_SHA3_224_RSA_PKCS**, **CKM_SHA3_256_RSA_PKCS**, **CKM_SHA3_384_RSA_PKCS**,
6775 and **CKM_SHA3_512_RSA_PKCS** respectively, performs similarly as the other
6776 **CKM_SHAX_RSA_PKCS** mechanisms but uses the corresponding SHA3 hash functions.

6777 **6.1.19 PKCS #1 RSA PSS signature with SHA3**

6778 The PKCS #1 RSA PSS signature with SHA3-224, SHA3-256, SHA3-384, SHA3-512 mechanisms,
6779 denoted **CKM_SHA3_224_RSA_PKCS_PSS**, **CKM_SHA3_256_RSA_PKCS_PSS**,
6780 **CKM_SHA3_384_RSA_PKCS_PSS**, and **CKM_SHA3_512_RSA_PKCS_PSS** respectively, performs
6781 similarly as the other **CKM_SHAX_RSA_PKCS_PSS** mechanisms but uses the corresponding SHA-3
6782 hash functions.

6783 **6.1.20 ANSI X9.31 RSA signature with SHA-1**

6784 The ANSI X9.31 RSA signature with SHA-1 mechanism, denoted **CKM_SHA1_RSA_X9_31**, performs
6785 single- and multiple-part digital signatures and verification operations without message recovery. The
6786 operations performed are as described in ANSI X9.31.

6787 This mechanism does not have a parameter.

6788 Constraints on key types and the length of the data for these mechanisms are summarized in the
6789 following table. In the table, *k* is the length in bytes of the RSA modulus. For all operations, the *k* value
6790 must be at least 128 and a multiple of 32 as specified in ANSI X9.31.

6791 *Table 45, ANSI X9.31 RSA Signatures with SHA-1: Key And Data Length*

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

6792 2 Data length, signature length.

6793 For these mechanisms, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO**
6794 structure specify the supported range of RSA modulus sizes, in bits.

6795 **6.1.21 TPM 1.1b and TPM 1.2 PKCS #1 v1.5 RSA**

6796 The TPM 1.1b and TPM 1.2 PKCS #1 v1.5 RSA mechanism, denoted **CKM_RSA_PKCS TPM_1_1**, is a
6797 multi-use mechanism based on the RSA public-key cryptosystem and the block formats initially defined in
6798 PKCS #1 v1.5, with additional formatting rules defined in TCPA TPM Specification Version 1.1b.
6799 Additional formatting rules remained the same in TCG TPM Specification 1.2. The mechanism supports
6800 single-part encryption and decryption; key wrapping; and key unwrapping.

6801 This mechanism does not have a parameter. It differs from the standard PKCS#1 v1.5 RSA encryption
6802 mechanism in that the plaintext is wrapped in a **TCPA_BOUND_DATA** (**TPM_BOUND_DATA** for TPM
6803 1.2) structure before being submitted to the PKCS#1 v1.5 encryption process. On encryption, the version
6804 field of the **TCPA_BOUND_DATA** (**TPM_BOUND_DATA** for TPM 1.2) structure must contain 0x01, 0x01,
6805 0x00, 0x00. On decryption, any structure of the form 0x01, 0x01, 0xXX, 0xYY may be accepted.

6806 This mechanism can wrap and unwrap any secret key of appropriate length. Of course, a particular token
6807 may not be able to wrap/unwrap every appropriate-length secret key that it supports. For wrapping, the
6808 “input” to the encryption operation is the value of the **CKA_VALUE** attribute of the key that is wrapped;
6809 similarly for unwrapping. The mechanism does not wrap the key type or any other information about the
6810 key, except the key length; the application must convey these separately. In particular, the mechanism
6811 contributes only the **CKA_CLASS** and **CKA_VALUE** (and **CKA_VALUE_LEN**, if the key has it) attributes
6812 to the recovered key during unwrapping; other attributes must be specified in the template.

6813 Constraints on key types and the length of the data are summarized in the following table. For encryption
6814 and decryption, the input and output data may begin at the same location in memory. In the table, *k* is the
6815 length in bytes of the RSA modulus.

6816 *Table 46, TPM 1.1b and TPM 1.2 PKCS #1 v1.5 RSA: Key And Data Length*

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

6817 ¹ Single-part operations only.

6818

6819 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
6820 specify the supported range of RSA modulus sizes, in bits.

6821 **6.1.22 TPM 1.1b and TPM 1.2 PKCS #1 RSA OAEP**

6822 The TPM 1.1b and TPM 1.2 PKCS #1 RSA OAEP mechanism, denoted
6823 **CKM_RSA_PKCS_OAEP TPM_1_1**, is a multi-purpose mechanism based on the RSA public-key
6824 cryptosystem and the OAEP block format defined in PKCS #1, with additional formatting defined in TCPA
6825 TPM Specification Version 1.1b. Additional formatting rules remained the same in TCG TPM
6826 Specification 1.2. The mechanism supports single-part encryption and decryption; key wrapping; and key
6827 unwrapping.

6828 This mechanism does not have a parameter. It differs from the standard PKCS#1 OAEP RSA encryption
6829 mechanism in that the plaintext is wrapped in a **TCPA_BOUND_DATA** (**TPM_BOUND_DATA** for TPM
6830 1.2) structure before being submitted to the encryption process and that all of the values of the
6831 parameters that are passed to a standard **CKM_RSA_PKCS_OAEP** operation are fixed. On encryption,
6832 the version field of the **TCPA_BOUND_DATA** (**TPM_BOUND_DATA** for TPM 1.2) structure must contain
6833 0x01, 0x01, 0x00, 0x00. On decryption, any structure of the form 0x01, 0x01, 0xXX, 0xYY may be
6834 accepted.

6835 This mechanism can wrap and unwrap any secret key of appropriate length. Of course, a particular token
6836 may not be able to wrap/unwrap every appropriate-length secret key that it supports. For wrapping, the
6837 "input" to the encryption operation is the value of the **CKA_VALUE** attribute of the key that is wrapped;
6838 similarly for unwrapping. The mechanism does not wrap the key type or any other information about the
6839 key, except the key length; the application must convey these separately. In particular, the mechanism
6840 contributes only the **CKA_CLASS** and **CKA_VALUE** (and **CKA_VALUE_LEN**, if the key has it) attributes
6841 to the recovered key during unwrapping; other attributes must be specified in the template.

6842 Constraints on key types and the length of the data are summarized in the following table. For encryption
6843 and decryption, the input and output data may begin at the same location in memory. In the table, k is the
6844 length in bytes of the RSA modulus.

6845 *Table 47, TPM 1.1b and TPM 1.2 PKCS #1 RSA OAEP: Key And Data Length*

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

6846 ¹ Single-part operations only.

6847 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
6848 specify the supported range of RSA modulus sizes, in bits.

6849 **6.1.23 RSA AES KEY WRAP**

6850 The RSA AES key wrap mechanism, denoted **CKM_RSA_AES_KEY_WRAP**, is a mechanism based on
6851 the RSA public-key cryptosystem and the AES key wrap mechanism. It supports single-part key
6852 wrapping; and key unwrapping.

6853 It has a parameter, a **CK_RSA_AES_KEY_WRAP_PARAMS** structure.

6854 The mechanism can wrap and unwrap a target asymmetric key of any length and type using an RSA
6855 key.

- 6856 - A temporary AES key is used for wrapping the target key using
6857 **CKM_AES_KEY_WRAP_KWP** mechanism.
- 6858 - The temporary AES key is wrapped with the wrapping RSA key using
6859 **CKM_RSA_PKCS_OAEP** mechanism.

6860 For wrapping, the mechanism -

- 6862 • Generates a temporary random AES key of *ulAESKeyBits* length. This key is not accessible to the
6863 user - no handle is returned.
- 6864 • Wraps the AES key with the wrapping RSA key using **CKM_RSA_PKCS_OAEP** with parameters
6865 of *OAEPPParams*.
- 6866 • Wraps the target key with the temporary AES key using **CKM_AES_KEY_WRAP_KWP**.
- 6867 • Zeroizes the temporary AES key
- 6868 • Concatenates two wrapped keys and outputs the concatenated blob. The first is the wrapped AES
6869 key, and the second is the wrapped target key.

6870 The private target key will be encoded as defined in section 6.7.

6871 The use of Attributes in the PrivateKeyInfo structure is OPTIONAL. In case of conflicts between the
6872 object attribute template, and Attributes in the PrivateKeyInfo structure, an error should be thrown
6873

- 6876 For unwrapping, the mechanism -
- 6877 • 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.
- 6879 • Un-wraps the temporary AES key from the first part with the private RSA key using **CKM_RSA_PKCS_OAEP** with parameters of *OAEPPParams*.
- 6881 • Un-wraps the target key from the second part with the temporary AES key using **CKM_AES_KEY_WRAP_KWP**.
- 6883 • Zeroizes the temporary AES key.
- 6884 • Returns the handle to the newly unwrapped target key.

6885 *Table 48, CKM_RSA_AES_KEY_WRAP Mechanisms vs. Functions*

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

¹SR = SignRecover, VR = VerifyRecover

6886 6.1.24 RSA AES KEY WRAP mechanism parameters

- 6887 ♦ **CK_RSA_AES_KEY_WRAP_PARAMS; CK_RSA_AES_KEY_WRAP_PARAMS_PTR**

6888 **CK_RSA_AES_KEY_WRAP_PARAMS** is a structure that provides the parameters to the
6889 **CKM_RSA_AES_KEY_WRAP** mechanism. It is defined as follows:

```
6890     typedef struct CK_RSA_AES_KEY_WRAP_PARAMS {
6891         CK ULONG                     ulAESKeyBits;
6892         CK_RSA_PKCS_OAEP_PARAMS_PTR pOAEPPParams;
6893     } CK_RSA_AES_KEY_WRAP_PARAMS;
```

6894

6895 The fields of the structure have the following meanings:

| | |
|--------------|---|
| ulAESKeyBits | length of the temporary AES key in bits. Can be only 128, 192 or 256. |
|--------------|---|

| | |
|--------------|---|
| pOAEPPParams | pointer to the parameters of the temporary AES key wrapping. See also the description of PKCS #1 RSA OAEP mechanism parameters. |
|--------------|---|

6901 **CK_RSA_AES_KEY_WRAP_PARAMS_PTR** is a pointer to a **CK_RSA_AES_KEY_WRAP_PARAMS**.

6902 6.1.25 FIPS 186-4

6903 When **CKM_RSA_PKCS** is operated in FIPS mode, the length of the modulus SHALL only be 1024,
6904 2048, or 3072 bits.

6905 6.2 DSA

6906 *Table 49, DSA Mechanisms vs. Functions*

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

6907 6.2.1 Definitions

6908 This section defines the key type “CKK_DSA” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE
 6909 attribute of DSA key objects.

6910 Mechanisms:

6911 CKM_DSA_KEY_PAIR_GEN
 6912 CKM_DSA
 6913 CKM_DSA_SHA1
 6914 CKM_DSA_SHA224
 6915 CKM_DSA_SHA256
 6916 CKM_DSA_SHA384
 6917 CKM_DSA_SHA512
 6918 CKM_DSA_SHA3_224
 6919 CKM_DSA_SHA3_256
 6920 CKM_DSA_SHA3_384
 6921 CKM_DSA_SHA3_512
 6922 CKM_DSA_PARAMETER_GEN
 6923 CKM_DSA_PROBABILISTIC_PARAMETER_GEN
 6924 CKM_DSA_SHAWE_TAYLOR_PARAMETER_GEN
 6925 CKM_DSA_FIPS_G_GEN
 6926

6927 ◆ **CK_DSA_PARAMETER_GEN_PARAM**

6928 CK_DSA_PARAMETER_GEN_PARAM is a structure which provides and returns parameters for the
6929 NIST FIPS 186-4 parameter generating algorithms.

6930 CK_DSA_PARAMETER_GEN_PARAM_PTR is a pointer to a CK_DSA_PARAMETER_GEN_PARAM.

6931

```
6932     typedef struct CK_DSA_PARAMETER_GEN_PARAM {  
6933         CK_MECHANISM_TYPE    hash;  
6934         CK_BYTE_PTR          pSeed;  
6935         CK ULONG              ulSeedLen;  
6936         CK ULONG              ulIndex;  
6937     } CK_DSA_PARAMETER_GEN_PARAM;
```

6938

6939 The fields of the structure have the following meanings:

6940 hash Mechanism value for the base hash used in PQG generation, Valid
6941 values are CKM_SHA_1, CKM_SHA224, CKM_SHA256,
6942 CKM_SHA384, CKM_SHA512.

6943 pSeed Seed value used to generate PQ and G. This value is returned by
6944 CKM_DSA_PROBABILISTIC_PARAMETER_GEN,
6945 CKM_DSA_SHAWE_TAYLOR_PARAMETER_GEN, and passed
6946 into CKM_DSA_FIPS_G_GEN.

6947 ulSeedLen Length of seed value.

6948 ullIndex Index value for generating G. Input for CKM_DSA_FIPS_G_GEN.
6949 Ignored by CKM_DSA_PROBABILISTIC_PARAMETER_GEN and
6950 CKM_DSA_SHAWE_TAYLOR_PARAMETER_GEN.

6951 **6.2.2 DSA public key objects**

6952 DSA public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_DSA**) hold DSA public keys.
6953 The following table defines the DSA public key object attributes, in addition to the common attributes
6954 defined for this object class:

6955 *Table 50, DSA Public Key Object Attributes*

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

6956 ^ Refer to Table 11 for footnotes

6957 The **CKA_PRIME**, **CKA_SUBPRIME** and **CKA_BASE** attribute values are collectively the “DSA domain
6958 parameters”. See FIPS PUB 186-4 for more information on DSA keys.

6959 The following is a sample template for creating a DSA public key object:

```
6960     CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;  
6961     CK_KEY_TYPE keyType = CKK_DSA;  
6962     CK_UTF8CHAR label[] = "A DSA public key object";  
6963     CK_BYTE prime[] = {...};  
6964     CK_BYTE subprime[] = {...};  
6965     CK_BYTE base[] = {...};
```

```

6966    CK_BYTE value[] = { ... };
6967    CK_BBOOL true = CK_TRUE;
6968    CK_ATTRIBUTE template[] = {
6969        {CKA_CLASS, &class, sizeof(class)},
6970        {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
6971        {CKA_TOKEN, &true, sizeof(true)},
6972        {CKA_LABEL, label, sizeof(label)-1},
6973        {CKA_PRIME, prime, sizeof(prime)},
6974        {CKA_SUBPRIME, subprime, sizeof(subprime)},
6975        {CKA_BASE, base, sizeof(base)},
6976        {CKA_VALUE, value, sizeof(value)}}
6977    };
6978

```

6.2.3 DSA Key Restrictions

6980 FIPS PUB 186-4 specifies permitted combinations of prime and sub-prime lengths. They are:

- Prime: 1024 bits, Subprime: 160
- Prime: 2048 bits, Subprime: 224
- Prime: 2048 bits, Subprime: 256
- Prime: 3072 bits, Subprime: 256

6985 Earlier versions of FIPS 186 permitted smaller prime lengths, and those are included here for backwards
6986 compatibility. An implementation that is compliant to FIPS 186-4 does not permit the use of primes of
6987 any length less than 1024 bits.

6.2.4 DSA private key objects

6989 DSA private key objects (object class **CKO_PRIVATE_KEY**, key type **CKK_DSA**) hold DSA private keys.
6990 The following table defines the DSA private key object attributes, in addition to the common attributes
6991 defined for this object class:

6992 *Table 51, DSA Private Key Object Attributes*

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

6993 ¹Refer to Table 11 for footnotes

6994 The **CKA_PRIME**, **CKA_SUBPRIME** and **CKA_BASE** attribute values are collectively the “DSA domain
6995 parameters”. See FIPS PUB 186-4 for more information on DSA keys.

6996 Note that when generating a DSA private key, the DSA domain parameters are *not* specified in the key’s
6997 template. This is because DSA private keys are only generated as part of a DSA key pair, and the DSA
6998 domain parameters for the pair are specified in the template for the DSA public key.

6999 The following is a sample template for creating a DSA private key object:

```

7000    CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
7001    CK_KEY_TYPE keyType = CKK_DSA;
7002    CK_UTF8CHAR label[] = "A DSA private key object";
7003    CK_BYTE subject[] = { ... };
7004    CK_BYTE id[] = {123};

```

```

7005 CK_BYTE prime[] = {...};
7006 CK_BYTE subprime[] = {...};
7007 CK_BYTE base[] = {...};
7008 CK_BYTE value[] = {...};
7009 CK_BBOOL true = CK_TRUE;
7010 CK_ATTRIBUTE template[] = {
7011     {CKA_CLASS, &class, sizeof(class)},
7012     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
7013     {CKA_TOKEN, &true, sizeof(true)},
7014     {CKA_LABEL, label, sizeof(label)-1},
7015     {CKA SUBJECT, subject, sizeof(subject)},
7016     {CKA_ID, id, sizeof(id)},
7017     {CKA_SENSITIVE, &true, sizeof(true)},
7018     {CKA_SIGN, &true, sizeof(true)},
7019     {CKA_PRIME, prime, sizeof(prime)},
7020     {CKA_SUBPRIME, subprime, sizeof(subprime)},
7021     {CKA_BASE, base, sizeof(base)},
7022     {CKA_VALUE, value, sizeof(value)}
7023 };

```

6.2.5 DSA domain parameter objects

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

Table 52, DSA Domain Parameter Object Attributes

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

Refer to Table 11 for footnotes

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

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

- If **CKA_PRIME_BITS** is less than or equal to 1024 then **CKA_SUBPRIME_BITS** shall be 160 bits
- If **CKA_PRIME_BITS** equals 2048 then **CKA_SUBPRIME_BITS** shall be 224 bits
- If **CKA_PRIME_BITS** equals 3072 then **CKA_SUBPRIME_BITS** shall be 256 bits

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

```

7039     CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
7040     CK_KEY_TYPE keyType = CKK_DSA;
7041     CK_UTF8CHAR label[] = "A DSA domain parameter object";
7042     CK_BYTE prime[] = {...};
7043     CK_BYTE subprime[] = {...};
7044     CK_BYTE base[] = {...};

```

```

7045     CK_BBOOL true = CK_TRUE;
7046     CK_ATTRIBUTE template[] = {
7047         { CKA_CLASS, &class, sizeof(class) },
7048         { CKA_KEY_TYPE, &keyType, sizeof(keyType) },
7049         { CKA_TOKEN, &true, sizeof(true) },
7050         { CKA_LABEL, label, sizeof(label)-1 },
7051         { CKA_PRIME, prime, sizeof(prime) },
7052         { CKA_SUBPRIME, subprime, sizeof(subprime) },
7053         { CKA_BASE, base, sizeof(base) },
7054     };
```

7055 6.2.6 DSA key pair generation

7056 The DSA key pair generation mechanism, denoted **CKM_DSA_KEY_PAIR_GEN**, is a key pair generation
 7057 mechanism based on the Digital Signature Algorithm defined in FIPS PUB 186-2.

7058 This mechanism does not have a parameter.

7059 The mechanism generates DSA public/private key pairs with a particular prime, subprime and base, as
 7060 specified in the **CKA_PRIME**, **CKA_SUBPRIME**, and **CKA_BASE** attributes of the template for the public
 7061 key.

7062 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
 7063 public key and the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_SUBPRIME**, **CKA_BASE**, and
 7064 **CKA_VALUE** attributes to the new private key. Other attributes supported by the DSA public and private
 7065 key types (specifically, the flags indicating which functions the keys support) may also be specified in the
 7066 templates for the keys, or else are assigned default initial values.

7067 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 7068 specify the supported range of DSA prime sizes, in bits.

7069 6.2.7 DSA domain parameter generation

7070 The DSA domain parameter generation mechanism, denoted **CKM_DSA_PARAMETER_GEN**, is a
 7071 domain parameter generation mechanism based on the Digital Signature Algorithm defined in FIPS PUB
 7072 186-2.

7073 This mechanism does not have a parameter.

7074 The mechanism generates DSA domain parameters with a particular prime length in bits, as specified in
 7075 the **CKA_PRIME_BITS** attribute of the template.

7076 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_SUBPRIME**,
 7077 **CKA_BASE** and **CKA_PRIME_BITS** attributes to the new object. Other attributes supported by the DSA
 7078 domain parameter types may also be specified in the template, or else are assigned default initial values.

7079 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 7080 specify the supported range of DSA prime sizes, in bits.

7081 6.2.8 DSA probabilistic domain parameter generation

7082 The DSA probabilistic domain parameter generation mechanism, denoted
 7083 **CKM_DSA_PROBABILISTIC_PARAMETER_GEN**, is a domain parameter generation mechanism based
 7084 on the Digital Signature Algorithm defined in FIPS PUB 186-4, section Appendix A.1.1 Generation and
 7085 Validation of Probable Primes..

7086 This mechanism takes a **CK_DSA_PARAMETER_GEN_PARAM** which supplies the base hash and
 7087 returns the seed (pSeed) and the length (ulSeedLen).

7088 The mechanism generates DSA the prime and subprime domain parameters with a particular prime
 7089 length in bits, as specified in the **CKA_PRIME_BITS** attribute of the template and the subprime length as
 7090 specified in the **CKA_SUBPRIME_BITS** attribute of the template.

7091 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_SUBPRIME**,
7092 **CKA_PRIME_BITS**, and **CKA_SUBPRIME_BITS** attributes to the new object. **CKA_BASE** is not set by
7093 this call. Other attributes supported by the DSA domain parameter types may also be specified in the
7094 template, or else are assigned default initial values.
7095 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7096 specify the supported range of DSA prime sizes, in bits.

7097 **6.2.9 DSA Shawe-Taylor domain parameter generation**

7098 The DSA Shawe-Taylor domain parameter generation mechanism, denoted
7099 **CKM_DSA_SHAWE_TAYLOR_PARAMETER_GEN**, is a domain parameter generation mechanism
7100 based on the Digital Signature Algorithm defined in FIPS PUB 186-4, section Appendix A.1.2
7101 Construction and Validation of Provable Primes p and q.
7102 This mechanism takes a **CK_DSA_PARAMETER_GEN_PARAM** which supplies the base hash and
7103 returns the seed (pSeed) and the length (ulSeedLen).
7104 The mechanism generates DSA the prime and subprime domain parameters with a particular prime
7105 length in bits, as specified in the **CKA_PRIME_BITS** attribute of the template and the subprime length as
7106 specified in the **CKA_SUBPRIME_BITS** attribute of the template.
7107 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_SUBPRIME**,
7108 **CKA_PRIME_BITS**, and **CKA_SUBPRIME_BITS** attributes to the new object. **CKA_BASE** is not set by
7109 this call. Other attributes supported by the DSA domain parameter types may also be specified in the
7110 template, or else are assigned default initial values.
7111 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7112 specify the supported range of DSA prime sizes, in bits.

7113 **6.2.10 DSA base domain parameter generation**

7114 The DSA base domain parameter generation mechanism, denoted **CKM_DSA_FIPS_G_GEN**, is a base
7115 parameter generation mechanism based on the Digital Signature Algorithm defined in FIPS PUB 186-4,
7116 section Appendix A.2 Generation of Generator G.
7117 This mechanism takes a **CK_DSA_PARAMETER_GEN_PARAM** which supplies the base hash the seed
7118 (pSeed) and the length (ulSeedLen) and the index value.
7119 The mechanism generates the DSA base with the domain parameter specified in the **CKA_PRIME** and
7120 **CKA_SUBPRIME** attributes of the template.
7121 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_BASE** attributes to the new
7122 object. Other attributes supported by the DSA domain parameter types may also be specified in the
7123 template, or else are assigned default initial values.
7124 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7125 specify the supported range of DSA prime sizes, in bits.

7126 **6.2.11 DSA without hashing**

7127 The DSA without hashing mechanism, denoted **CKM_DSA**, is a mechanism for single-part signatures and
7128 verification based on the Digital Signature Algorithm defined in FIPS PUB 186-2. (This mechanism
7129 corresponds only to the part of DSA that processes the 20-byte hash value; it does not compute the hash
7130 value.)
7131 For the purposes of this mechanism, a DSA signature is a 40-byte string, corresponding to the
7132 concatenation of the DSA values *r* and *s*, each represented most-significant byte first.
7133 It does not have a parameter.
7134 Constraints on key types and the length of data are summarized in the following table:
7135 *Table 53, DSA: Key And Data Length*

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

7136 ¹ Single-part operations only.

7137 ² Data length, signature length.

7138 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7139 specify the supported range of DSA prime sizes, in bits.

7140 6.2.12 DSA with SHA-1

7141 The DSA with SHA-1 mechanism, denoted **CKM_DSA_SHA1**, is a mechanism for single- and multiple-
7142 part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-2.
7143 This mechanism computes the entire DSA specification, including the hashing with SHA-1.

7144 For the purposes of this mechanism, a DSA signature is a 40-byte string, corresponding to the
7145 concatenation of the DSA values *r* and *s*, each represented most-significant byte first.

7146 This mechanism does not have a parameter.

7147 Constraints on key types and the length of data are summarized in the following table:

7148 *Table 54, DSA with SHA-1: Key And Data Length*

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

7149 ² Data length, signature length.

7150 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7151 specify the supported range of DSA prime sizes, in bits.

7152 6.2.13 FIPS 186-4

7153 When CKM_DSA is operated in FIPS mode, only the following bit lengths of *p* and *q*, represented by *L*
7154 and *N*, SHALL be used:

7155 *L* = 1024, *N* = 160

7156 *L* = 2048, *N* = 224

7157 *L* = 2048, *N* = 256

7158 *L* = 3072, *N* = 256

7159

7160 6.2.14 DSA with SHA-224

7161 The DSA with SHA-224 mechanism, denoted **CKM_DSA_SHA224**, is a mechanism for single- and
7162 multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB
7163 186-4. This mechanism computes the entire DSA specification, including the hashing with SHA-224.

7164 For the purposes of this mechanism, a DSA signature is a string of length 2^{*}subprime , corresponding to
 7165 the concatenation of the DSA values r and s , each represented most-significant byte first.
 7166 This mechanism does not have a parameter.
 7167 Constraints on key types and the length of data are summarized in the following table:
 7168 *Table 55, DSA with SHA-244: Key And Data Length*

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

7169 ² Data length, signature length.
 7170 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 7171 specify the supported range of DSA prime sizes, in bits.

6.2.15 DSA with SHA-256

7173 The DSA with SHA-256 mechanism, denoted **CKM_DSA_SHA256**, is a mechanism for single- and
 7174 multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB
 7175 186-4. This mechanism computes the entire DSA specification, including the hashing with SHA-256.
 7176 For the purposes of this mechanism, a DSA signature is a string of length 2^{*}subprime , corresponding to
 7177 the concatenation of the DSA values r and s , each represented most-significant byte first.
 7178 This mechanism does not have a parameter.
 7179 Constraints on key types and the length of data are summarized in the following table:
 7180 *Table 56, DSA with SHA-256: Key And Data Length*

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

7181 ² Data length, signature length.
 7182

6.2.16 DSA with SHA-384

 7183 The DSA with SHA-384 mechanism, denoted **CKM_DSA_SHA384**, is a mechanism for single- and
 7184 multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB
 7185 186-4. This mechanism computes the entire DSA specification, including the hashing with SHA-384.
 7186 For the purposes of this mechanism, a DSA signature is a string of length 2^{*}subprime , corresponding to
 7187 the concatenation of the DSA values r and s , each represented most-significant byte first.
 7188 This mechanism does not have a parameter.
 7189 Constraints on key types and the length of data are summarized in the following table:
 7190 *Table 57, DSA with SHA-384: Key And Data Length*

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

7191 ² Data length, signature length.

6.2.17 DSA with SHA-512

7193 The DSA with SHA-512 mechanism, denoted **CKM_DSA_SHA512**, is a mechanism for single- and
 7194 multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB
 7195 186-4. This mechanism computes the entire DSA specification, including the hashing with SHA-512.

7196 For the purposes of this mechanism, a DSA signature is a string of length 2*subprime, corresponding to
 7197 the concatenation of the DSA values r and s , each represented most-significant byte first.

7198 This mechanism does not have a parameter.

7199 Constraints on key types and the length of data are summarized in the following table:

7200 *Table 58, DSA with SHA-512: Key And Data Length*

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

7201 ² Data length, signature length.

6.2.18 DSA with SHA3-224

7203 The DSA with SHA3-224 mechanism, denoted **CKM_DSA_SHA3_224**, is a mechanism for single- and
 7204 multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB
 7205 186-4. This mechanism computes the entire DSA specification, including the hashing with SHA3-224.

7206 For the purposes of this mechanism, a DSA signature is a string of length 2*subprime, corresponding to
 7207 the concatenation of the DSA values r and s , each represented most-significant byte first.

7208 This mechanism does not have a parameter.

7209 Constraints on key types and the length of data are summarized in the following table:

7210 *Table 59, DSA with SHA3-224: Key And Data Length*

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

7211 ² Data length, signature length.

7212 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 7213 specify the supported range of DSA prime sizes, in bits.

6.2.19 DSA with SHA3-256

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

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

This mechanism does not have a parameter.

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

Table 60, DSA with SHA3-256: Key And Data Length

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

² Data length, signature length.

6.2.20 DSA with SHA3-384

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

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

This mechanism does not have a parameter.

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

Table 61, DSA with SHA3-384: Key And Data Length

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

² Data length, signature length.

6.2.21 DSA with SHA3-512

The DSA with SHA3-512 mechanism, denoted **CKM_DSA_SHA3_512**, is a mechanism for single- and multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-4. This mechanism computes the entire DSA specification, including the hashing with SH3A-512.

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

This mechanism does not have a parameter.

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

Table 62, DSA with SHA3-512: Key And Data Length

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

7243 ² Data length, signature length.

7244

7245 6.3 Elliptic Curve

7246 The Elliptic Curve (EC) cryptosystem in this document was originally based on the one described in the
 7247 ANSI X9.62 and X9.63 standards developed by the ANSI X9F1 working group.

7248 The EC cryptosystem developed by the ANSI X9F1 working group was created at a time when EC curves
 7249 were always represented in their Weierstrass form. Since that time, new curves represented in Edwards
 7250 form (RFC 8032) and Montgomery form (RFC 7748) have become more common. To support these new
 7251 curves, the EC cryptosystem in this document has been extended from the original. Additional key
 7252 generation mechanisms have been added as well as an additional signature generation mechanism.

7253

7254 *Table 63, Elliptic Curve Mechanisms vs. Functions*

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

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

7255

7256

7257 Table 64, Mechanism Information Flags

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

7258 Note: CKF_EC_NAMEDCURVE is deprecated with PKCS#11 3.00. It is replaced by CKF_EC_OID.

7259 In these standards, there are two different varieties of EC defined:

7260 1. EC using a field with an odd prime number of elements (i.e. the finite field F_p).7261 2. EC using a field of characteristic two (i.e. the finite field F_{2^m}).7262 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.7268 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.7275 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

7279 supporting the compressed form. A Cryptoki library that can perform EC mechanisms must set either or
7280 both of these flags for each EC mechanism.
7281 Note that an implementation of a Cryptoki library supporting EC with only one variety, one representation
7282 of domain parameters or one form may encounter difficulties achieving interoperability with other
7283 implementations.
7284 If an attempt to create, generate, derive or unwrap an EC key of an unsupported curve is made, the
7285 attempt should fail with the error code CKR_CURVE_NOT_SUPPORTED. If an attempt to create,
7286 generate, derive, or unwrap an EC key with invalid or of an unsupported representation of domain
7287 parameters is made, that attempt should fail with the error code CKR_DOMAIN_PARAMS_INVALID. If
7288 an attempt to create, generate, derive, or unwrap an EC key of an unsupported form is made, that
7289 attempt should fail with the error code CKR_TEMPLATE_INCONSISTENT.

7290 **6.3.1 EC Signatures**

7291 For the purposes of these mechanisms, an ECDSA signature is an octet string of even length which is at
7292 most two times $nLen$ octets, where $nLen$ is the length in octets of the base point order n . The signature
7293 octets correspond to the concatenation of the ECDSA values r and s , both represented as an octet string
7294 of equal length of at most $nLen$ with the most significant byte first. If r and s have different octet length,
7295 the shorter of both must be padded with leading zero octets such that both have the same octet length.
7296 Loosely spoken, the first half of the signature is r and the second half is s . For signatures created by a
7297 token, the resulting signature is always of length $2nLen$. For signatures passed to a token for verification,
7298 the signature may have a shorter length but must be composed as specified before.

7299 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
7300 length of n will be used. Any truncation is done by the token.

7301 Note: For applications, it is recommended to encode the signature as an octet string of length two times
7302 $nLen$ if possible. This ensures that the application works with PKCS#11 modules which have been
7303 implemented based on an older version of this document. Older versions required all signatures to have
7304 length two times $nLen$. It may be impossible to encode the signature with the maximum length of two
7305 times $nLen$ if the application just gets the integer values of r and s (i.e. without leading zeros), but does
7306 not know the base point order n , because r and s can have any value between zero and the base point
7307 order n .

7308 An EdDSA signature is an octet string of even length which is two times $nLen$ octets, where $nLen$ is
7309 calculated as EdDSA parameter b divided by 8. The signature octets correspond to the concatenation of
7310 the EdDSA values R and S as defined in [RFC 8032], both represented as an octet string of equal length
7311 of $nLen$ bytes in little endian order.

7312 **6.3.2 Definitions**

7313 This section defines the key types "CKK_EC", "CKK_EC_EDWARDS" and "CKK_EC_MONTGOMERY"
7314 for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

7315 Note: CKK_ECDSA is deprecated. It is replaced by CKK_EC.

7316 Mechanisms:

7317 CKM_EC_KEY_PAIR_GEN
7318 CKM_EC_EDWARDS_KEY_PAIR_GEN
7319 CKM_EC_MONTGOMERY_KEY_PAIR_GEN
7320 CKM_ECDSA
7321 CKM_ECDSA_SHA1
7322 CKM_ECDSA_SHA224
7323 CKM_ECDSA_SHA256
7324 CKM_ECDSA_SHA384

7326 CKM_ECDSA_SHA512
7327 CKM_ECDSA_SHA3_224
7328 CKM_ECDSA_SHA3_256
7329 CKM_ECDSA_SHA3_384
7330 CKM_ECDSA_SHA3_512
7331 CKM_EDDSA
7332 CKM_XEDDSA
7333 CKM_ECDH1_DERIVE
7334 CKM_ECDH1_COFACTOR_DERIVE
7335 CKM_ECMQV_DERIVE
7336 CKM_ECDH_AES_KEY_WRAP
7337
7338 CKD_NULL
7339 CKD_SHA1_KDF
7340 CKD_SHA224_KDF
7341 CKD_SHA256_KDF
7342 CKD_SHA384_KDF
7343 CKD_SHA512_KDF
7344 CKD_SHA3_224_KDF
7345 CKD_SHA3_256_KDF
7346 CKD_SHA3_384_KDF
7347 CKD_SHA3_512_KDF
7348 CKD_SHA1_KDF_SP800
7349 CKD_SHA224_KDF_SP800
7350 CKD_SHA256_KDF_SP800
7351 CKD_SHA384_KDF_SP800
7352 CKD_SHA512_KDF_SP800
7353 CKD_SHA3_224_KDF_SP800
7354 CKD_SHA3_256_KDF_SP800
7355 CKD_SHA3_384_KDF_SP800
7356 CKD_SHA3_512_KDF_SP800
7357 CKD_BLAKE2B_160_KDF
7358 CKD_BLAKE2B_256_KDF
7359 CKD_BLAKE2B_384_KDF
7360 CKD_BLAKE2B_512_KDF

7361 6.3.3 Short Weierstrass Elliptic Curve public key objects

7362 Short Weierstrass EC public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_EC**) hold EC
7363 public keys. The following table defines the EC public key object attributes, in addition to the common
7364 attributes defined for this object class:

7365 *Table 65, Elliptic Curve Public Key Object Attributes*

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

7366 - Refer to Table 11 for footnotes

7367 Note: CKA_ECDSA_PARAMS is deprecated. It is replaced by CKA_EC_PARAMS.

7368 The **CKA_EC_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI
7369 X9.62 as a choice of three parameter representation methods with the following syntax:

```
7370     Parameters ::= CHOICE {
7371         ecParameters    ECParameters,
7372         oId             CURVES.&id({CurveNames}),
7373         implicitlyCA   NULL,
7374         curveName      PrintableString
7375     }
```

7376

7377 This allows detailed specification of all required values using choice **ecParameters**, the use of **oId** as an
7378 object identifier substitute for a particular set of Elliptic Curve domain parameters, or **implicitlyCA** to
7379 indicate that the domain parameters are explicitly defined elsewhere, or **curveName** to specify a curve
7380 name as e.g. define in [ANSI X9.62], [BRAINPOOL], [SEC 2], [LEGIFRANCE]. The use of **oId** or
7381 **curveName** is recommended over the choice **ecParameters**. The choice **implicitlyCA** must not be used
7382 in Cryptoki.

7383 The following is a sample template for creating an short Weierstrass EC public key object:

```
7384     CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
7385     CK_KEY_TYPE keyType = CKK_EC;
7386     CK_UTF8CHAR label[] = "An EC public key object";
7387     CK_BYTE ecParams[] = {...};
7388     CK_BYTE ecPoint[] = {...};
7389     CK_BBOOL true = CK_TRUE;
7390     CK_ATTRIBUTE template[] = {
7391         {CKA_CLASS, &class, sizeof(class)},
7392         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
7393         {CKA_TOKEN, &true, sizeof(true)},
7394         {CKA_LABEL, label, sizeof(label)-1},
7395         {CKA_EC_PARAMS, ecParams, sizeof(ecParams)},
7396         {CKA_EC_POINT, ecPoint, sizeof(ecPoint)}
7397     };
```

7398 6.3.4 Short Weierstrass Elliptic Curve private key objects

7399 Short Weierstrass EC private key objects (object class **CKO_PRIVATE_KEY**, key type **CKK_EC**) hold
7400 EC private keys. See Section 6.3 for more information about EC. The following table defines the EC
7401 private key object attributes, in addition to the common attributes defined for this object class:

7402 *Table 66, Elliptic Curve Private Key Object Attributes*

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

7403 - Refer to Table 11 for footnotes

7404 The **CKA_EC_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI
7405 X9.62 as a choice of three parameter representation methods with the following syntax:

```
7406     Parameters ::= CHOICE {
7407         ecParameters    ECParameters,
7408         oId             CURVES.&id({CurveNames}),
7409         implicitlyCA   NULL,
7410         curveName      PrintableString
7411     }
```

7412 This allows detailed specification of all required values using choice **ecParameters**, the use of **oId** as an
7413 object identifier substitute for a particular set of Elliptic Curve domain parameters, or **implicitlyCA** to
7414 indicate that the domain parameters are explicitly defined elsewhere, or **curveName** to specify a curve
7415 name as e.g. define in [ANSI X9.62], [BRAINPOOL], [SEC 2], [LEGIFRANCE]. The use of **oId** or
7416 **curveName** is recommended over the choice **ecParameters**. The choice **implicitlyCA** must not be used
7417 in Cryptoki. Note that when generating an EC private key, the EC domain parameters are *not* specified in
7418 the key’s template. This is because EC private keys are only generated as part of an EC key pair, and
7419 the EC domain parameters for the pair are specified in the template for the EC public key.

7420
7421 The following is a sample template for creating an short Weierstrass EC private key object:

```
7422     CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
7423     CK_KEY_TYPE keyType = CKK_EC;
7424     CK_UTF8CHAR label[] = "An EC private key object";
7425     CK_BYTE subject[] = {...};
7426     CK_BYTE id[] = {123};
7427     CK_BYTE ecParams[] = {...};
7428     CK_BYTE value[] = {...};
7429     CK_BBOOL true = CK_TRUE;
7430     CK_ATTRIBUTE template[] = {
7431         {CKA_CLASS, &class, sizeof(class)},
7432         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
7433         {CKA_TOKEN, &true, sizeof(true)},
7434         {CKA_LABEL, label, sizeof(label)-1},
7435         {CKA_SUBJECT, subject, sizeof(subject)},
7436         {CKA_ID, id, sizeof(id)},
7437         {CKA_SENSITIVE, &true, sizeof(true)},
7438         {CKA_DERIVE, &true, sizeof(true)},
7439         {CKA_EC_PARAMS, ecParams, sizeof(ecParams)},
7440         {CKA_VALUE, value, sizeof(value)}
7441     };
```

7442 6.3.5 Edwards Elliptic Curve public key objects

7443 Edwards EC public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_EC_EDWARDS**) hold
7444 Edwards EC public keys. The following table defines the Edwards EC public key object attributes, in
7445 addition to the common attributes defined for this object class:

7446 *Table 67, Edwards Elliptic Curve Public Key Object Attributes*

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

7447 Refer to Table 11 for footnotes

7448 The **CKA_EC_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI
 7449 X9.62 as a choice of three parameter representation methods. A 4th choice is added to support Edwards
 7450 and Montgomery Elliptic Curves. The CKA_EC_PARAMS attribute has the following syntax:

```
7451   Parameters ::= CHOICE {
7452     ecParameters  ECParameters,
7453     oId          CURVES.&id({CurveNames}),
7454     implicitlyCA NULL,
7455     curveName    PrintableString
7456 }
```

7457 Edwards EC public keys only support the use of the **curveName** selection to specify a curve name as
 7458 defined in [RFC 8032] and the use of the **oId** selection to specify a curve through an EdDSA algorithm as
 7459 defined in [RFC 8410]. Note that keys defined by RFC 8032 and RFC 8410 are incompatible.

7460 The following is a sample template for creating an Edwards EC public key object with Edwards25519
 7461 being specified as curveName:

```
7462   CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
7463   CK_KEY_TYPE keyType = CKK_EC_EDWARDS;
7464   CK_UTF8CHAR label[] = "An Edwards EC public key object";
7465   CK_BYTE ecParams[] = {0x13, 0x0c, 0x65, 0x64, 0x77, 0x61,
7466                         0x72, 0x64, 0x73, 0x32, 0x35, 0x35, 0x31, 0x39};
7467   CK_BYTE ecPoint[] = {...};
7468   CK_BBOOL true = CK_TRUE;
7469   CK_ATTRIBUTE template[] = {
7470     {CKA_CLASS, &class, sizeof(class)},
7471     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
7472     {CKA_TOKEN, &true, sizeof(true)},
7473     {CKA_LABEL, label, sizeof(label)-1},
7474     {CKA_EC_PARAMS, ecParams, sizeof(ecParams)},
7475     {CKA_EC_POINT, ecPoint, sizeof(ecPoint)}}
7476 };
```

7477 6.3.6 Edwards Elliptic Curve private key objects

7478 Edwards EC private key objects (object class **CKO_PRIVATE_KEY**, key type **CKK_EC_EDWARDS**)
 7479 hold Edwards EC private keys. See Section 6.3 for more information about EC. The following table
 7480 defines the Edwards EC private key object attributes, in addition to the common attributes defined for this
 7481 object class:

7482 *Table 68, Edwards Elliptic Curve Private Key Object Attributes*

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

Refer to Table 11 for footnotes

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

```
Parameters ::= CHOICE {
    ecParameters  ECParameters,
    oId           CURVES.&id({CurveNames}),
    implicitlyCA NULL,
    curveName     PrintableString
}
```

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

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

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

```
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_EC_EDWARDS;
CK_UTF8CHAR label[] = "An Edwards EC private key object";
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE ecParams[] = {...};
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_SUBJECT, subject, sizeof(subject)},
    {CKA_ID, id, sizeof(id)},
    {CKA_SENSITIVE, &true, sizeof(true)},
    {CKA_DERIVE, &true, sizeof(true)},
    {CKA_VALUE, value, sizeof(value)}};
};
```

6.3.7 Montgomery Elliptic Curve public key objects

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

7525 Table 69, Montgomery Elliptic Curve Public Key Object Attributes

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

7526 Refer to Table 11 for footnotes

7527 The **CKA_EC_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI
7528 X9.62 as a choice of three parameter representation methods. A 4th choice is added to support Edwards
7529 and Montgomery Elliptic Curves. The CKA_EC_PARAMS attribute has the following syntax:

```
7530     Parameters ::= CHOICE {  
7531         ecParameters    ECParameters,  
7532         oId             CURVES.&id({CurveNames}),  
7533         implicitlyCA   NULL,  
7534         curveName      PrintableString  
7535     }
```

7536 Montgomery EC public keys only support the use of the **curveName** selection to specify a curve name as
7537 defined in [RFC7748] and the use of the **oId** selection to specify a curve through an ECDH algorithm as
7538 defined in [RFC 8410]. Note that keys defined by RFC 7748 and RFC 8410 are incompatible.

7539 The following is a sample template for creating a Montgomery EC public key object:

```
7540     CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;  
7541     CK_KEY_TYPE keyType = CKK_EC_MONTGOMERY;  
7542     CK_UTF8CHAR label[] = "A Montgomery EC public key object";  
7543     CK_BYTE ecParams[] = {...};  
7544     CK_BYTE ecPoint[] = {...};  
7545     CK_BBOOL true = CK_TRUE;  
7546     CK_ATTRIBUTE template[] = {  
7547         {CKA_CLASS, &class, sizeof(class)},  
7548         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},  
7549         {CKA_TOKEN, &true, sizeof(true)},  
7550         {CKA_LABEL, label, sizeof(label)-1},  
7551         {CKA_EC_PARAMS, ecParams, sizeof(ecParams)},  
7552         {CKA_EC_POINT, ecPoint, sizeof(ecPoint)}  
7553     };
```

7554 6.3.8 Montgomery Elliptic Curve private key objects

7555 Montgomery EC private key objects (object class **CKO_PRIVATE_KEY**, key type
7556 **CKK_EC_MONTGOMERY**) hold Montgomery EC private keys. See Section 6.3 for more information
7557 about EC. The following table defines the Montgomery EC private key object attributes, in addition to the
7558 common attributes defined for this object class:

7559 Table 70, Montgomery Elliptic Curve Private Key Object Attributes

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

7560 Refer to Table 11 for footnotes

7561 The **CKA_EC_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI
7562 X9.62 as a choice of three parameter representation methods. A 4th choice is added to support Edwards
7563 and Montgomery Elliptic Curves. The **CKA_EC_PARAMS** attribute has the following syntax:

```
7564     Parameters ::= CHOICE {
7565         ecParameters    ECParameters,
7566         oId             CURVES.&id({CurveNames}),
7567         implicitlyCA   NULL,
7568         curveName       PrintableString
7569     }
```

7570 Montgomery EC private keys only support the use of the **curveName** selection to specify a curve name
7571 as defined in [RFC7748] and the use of the **oId** selection to specify a curve through an ECDH algorithm
7572 as defined in [RFC 8410]. Note that keys defined by RFC 7748 and RFC 8410 are incompatible.

7573 Note that when generating a Montgomery EC private key, the EC domain parameters are *not* specified in
7574 the key's template. This is because Montgomery EC private keys are only generated as part of a
7575 Montgomery EC key *pair*, and the EC domain parameters for the pair are specified in the template for the
7576 Montgomery EC public key.

7577 The following is a sample template for creating a Montgomery EC private key object:

```
7578     CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
7579     CK_KEY_TYPE keyType = CKK_EC_MONTGOMERY;
7580     CK_UTF8CHAR label[] = "A Montgomery EC private key object";
7581     CK_BYTE subject[] = {...};
7582     CK_BYTE id[] = {123};
7583     CK_BYTE ecParams[] = {...};
7584     CK_BYTE value[] = {...};
7585     CK_BBOOL true = CK_TRUE;
7586     CK_ATTRIBUTE template[] = {
7587         {CKA_CLASS, &class, sizeof(class)},
7588         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
7589         {CKA_TOKEN, &true, sizeof(true)},
7590         {CKA_LABEL, label, sizeof(label)-1},
7591         {CKA_SUBJECT, subject, sizeof(subject)},
7592         {CKA_ID, id, sizeof(id)},
7593         {CKA_SENSITIVE, &true, sizeof(true)},
7594         {CKA_DERIVE, &true, sizeof(true)},
7595         {CKA_VALUE, value, sizeof(value)}
7596     };
```

7597 6.3.9 Elliptic Curve key pair generation

7598 The short Weierstrass ECkey pair generation mechanism, denoted CKM_EC_KEY_PAIR_GEN, is a key
7599 pair generation mechanism that uses the method defined by the ANSI X9.62 and X9.63 standards.

7600 The short Weierstrass EC key pair generation mechanism, denoted
7601 CKM_EC_KEY_PAIR_GEN_W_EXTRA_BITS, is a key pair generation mechanism that uses the method
7602 defined by FIPS 186-4 Appendix B.4.1.

7603 These mechanisms do not have a parameter.

7604 These mechanisms generate EC public/private key pairs with particular EC domain parameters, as
7605 specified in the **CKA_EC_PARAMS** attribute of the template for the public key. Note that this version of
7606 Cryptoki does not include a mechanism for generating these EC domain parameters.

7607 These mechanism contribute the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_EC_POINT** attributes to the
7608 new public key and the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_EC_PARAMS** and **CKA_VALUE**
7609 attributes to the new private key. Other attributes supported by the EC public and private key types
7610 (specifically, the flags indicating which functions the keys support) may also be specified in the templates
7611 for the keys, or else are assigned default initial values.

7612 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7613 specify the minimum and maximum supported number of bits in the field sizes, respectively. For
7614 example, if a Cryptoki library supports only ECDSA using a field of characteristic 2 which has between
7615 2^{200} and 2^{300} elements, then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in binary
7616 notation, the number 2^{200} consists of a 1 bit followed by 200 0 bits. It is therefore a 201-bit number.
7617 Similarly, 2^{300} is a 301-bit number).

7618 **6.3.10 Edwards Elliptic Curve key pair generation**

7619 The Edwards EC key pair generation mechanism, denoted **CKM_EC_EDWARDS_KEY_PAIR_GEN**, is a
7620 key pair generation mechanism for EC keys over curves represented in Edwards form.

7621 This mechanism does not have a parameter.

7622 The mechanism can only generate EC public/private key pairs over the curves edwards25519 and
7623 edwards448 as defined in RFC 8032 or the curves id-Ed25519 and id-Ed448 as defined in RFC 8410.
7624 These curves can only be specified in the **CKA_EC_PARAMS** attribute of the template for the public key
7625 using the **curveName** or the **oid** methods. Attempts to generate keys over these curves using any other
7626 EC key pair generation mechanism will fail with **CKR_CURVE_NOT_SUPPORTED**.

7627 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_EC_POINT** attributes to the
7628 new public key and the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_EC_PARAMS** and **CKA_VALUE**
7629 attributes to the new private key. Other attributes supported by the Edwards EC public and private key
7630 types (specifically, the flags indicating which functions the keys support) may also be specified in the
7631 templates for the keys, or else are assigned default initial values.

7632 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7633 specify the minimum and maximum supported number of bits in the field sizes, respectively. For this
7634 mechanism, the only allowed values are 255 and 448 as RFC 8032 only defines curves of these two
7635 sizes. A Cryptoki implementation may support one or both of these curves and should set the
7636 *ulMinKeySize* and *ulMaxKeySize* fields accordingly.

7637 **6.3.11 Montgomery Elliptic Curve key pair generation**

7638 The Montgomery EC key pair generation mechanism, denoted
7639 **CKM_EC_MONTGOMERY_KEY_PAIR_GEN**, is a key pair generation mechanism for EC keys over
7640 curves represented in Montgomery form.

7641 This mechanism does not have a parameter.

7642 The mechanism can only generate Montgomery EC public/private key pairs over the curves curve25519
7643 and curve448 as defined in RFC 7748 or the curves id-X25519 and id-X448 as defined in RFC 8410.
7644 These curves can only be specified in the **CKA_EC_PARAMS** attribute of the template for the public key
7645 using the **curveName** or **old** methods. Attempts to generate keys over these curves using any other EC
7646 key pair generation mechanism will fail with **CKR_CURVE_NOT_SUPPORTED**.

7647 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_EC_POINT** attributes to the
7648 new public key and the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_EC_PARAMS** and **CKA_VALUE**
7649 attributes to the new private key. Other attributes supported by the EC public and private key types
7650 (specifically, the flags indicating which functions the keys support) may also be specified in the templates
7651 for the keys, or else are assigned default initial values.

7652 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7653 specify the minimum and maximum supported number of bits in the field sizes, respectively. For this
7654 mechanism, the only allowed values are 255 and 448 as RFC 7748 only defines curves of these two
7655 sizes. A Cryptoki implementation may support one or both of these curves and should set the
7656 *ulMinKeySize* and *ulMaxKeySize* fields accordingly.

7657 6.3.12 ECDSA without hashing

7658 Refer section 6.3.1 for signature encoding.

7659 The ECDSA without hashing mechanism, denoted **CKM_ECDSA**, is a mechanism for single-part
7660 signatures and verification for ECDSA. (This mechanism corresponds only to the part of ECDSA that
7661 processes the hash value, which should not be longer than 1024 bits; it does not compute the hash
7662 value.)

7663 This mechanism does not have a parameter.

7664 Constraints on key types and the length of data are summarized in the following table:

7665 *Table 71, ECDSA without hashing: Key and Data Length*

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

7666 ¹ Single-part operations only.

7667 ² Data length, signature length.

7668 ³ Input the entire raw digest. Internally, this will be truncated to the appropriate number of bits.

7669 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7670 specify the minimum and maximum supported number of bits in the field sizes, respectively. For
7671 example, if a Cryptoki library supports only ECDSA using a field of characteristic 2 which has between
7672 2^{200} and 2^{300} elements (inclusive), then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in
7673 binary notation, the number 2^{200} consists of a 1 bit followed by 200 0 bits. It is therefore a 201-bit number.
7674 Similarly, 2^{300} is a 301-bit number).

7675 6.3.13 ECDSA with hashing

7676 Refer to section 6.3.1 for signature encoding.

7677 The ECDSA with SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, SHA3-224, SHA3-256, SHA3-384,
7678 SHA3-512 mechanism, denoted
CKM_ECDSA_[SHA1|SHA224|SHA256|SHA384|SHA512|SHA3_224|SHA3_256|SHA3_384|SHA3_51
7679 **2]** respectively, is a mechanism for single- and multiple-part signatures and verification for ECDSA. This
7680 mechanism computes the entire ECDSA specification, including the hashing with SHA-1, SHA-224, SHA-
7681 256, SHA-384, SHA-512, SHA3-224, SHA3-256, SHA3-384, SHA3-512 respectively.

7683 This mechanism does not have a parameter.

7684 Constraints on key types and the length of data are summarized in the following table:

7685 *Table 72, ECDSA with hashing: Key and Data Length*

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

7686 ² Data length, signature length.

7687 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7688 specify the minimum and maximum supported number of bits in the field sizes, respectively. For
7689 example, if a Cryptoki library supports only ECDSA using a field of characteristic 2 which has between
7690 2^{200} and 2^{300} elements, then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in binary
7691 notation, the number 2^{200} consists of a 1 bit followed by 200 0 bits. It is therefore a 201-bit number.
7692 Similarly, 2^{300} is a 301-bit number).

6.3.14 EdDSA

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

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

Table 73, Mapping to RFC 8032 Signature Schemes

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

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

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

Table 74, EdDSA: Key and Data Length

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

² Data length, signature length.

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

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

6.3.15 XEdDSA

The XEdDSA mechanism, denoted **CKM_XEDDSA**, is a mechanism for single-part signatures and verification for XEdDSA. This mechanism implements the XEdDSA signature scheme defined in [XEDDSA]. CKM_XEDDSA operates on CKK_EC_MONTGOMERY type EC keys, which allows these keys to be used both for signing/verification and for Diffie-Hellman style key-exchanges. This double use is necessary for the Extended Triple Diffie-Hellman where the long-term identity key is used to sign short-term keys and also contributes to the DH key-exchange.

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

Table 75, XEdDSA: Key and Data Length

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

7724 2 Data length, signature length.

7725 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 7726 specify the minimum and maximum supported number of bits in the field sizes, respectively. For this
 7727 mechanism, the only allowed values are 255 and 448 as [XEDDSA] only defines curves of these two
 7728 sizes. A Cryptoki implementation may support one or both of these curves and should set the
 7729 *ulMinKeySize* and *ulMaxKeySize* fields accordingly.

7730 6.3.16 EC mechanism parameters

7731 ♦ **CK_EDDSA_PARAMS**, **CK_EDDSA_PARAMS_PTR**

7732 **CK_EDDSA_PARAMS** is a structure that provides the parameters for the **CKM_EDDSA** signature
 7733 mechanism. The structure is defined as follows:

```
7734     typedef struct CK_EDDSA_PARAMS {
7735         CK_BBOOL      phFlag;
7736         CK ULONG      ulContextDataLen;
7737         CK BYTE PTR   pContextData;
7738     } CK_EDDSA_PARAMS;
```

7739

7740 The fields of the structure have the following meanings:

7741 phFlag Boolean value which indicates if Prehashed variant of EdDSA should be used

7742 ulContextDataLen the length in bytes of the context data where 0 ≤ ulContextDataLen ≤ 255.

7743 pContextData context data shared between the signer and verifier

7744 **CK_EDDSA_PARAMS_PTR** is a pointer to a **CK_EDDSA_PARAMS**.

7745

7746 ♦ **CK_XEDDSA_PARAMS**, **CK_XEDDSA_PARAMS_PTR**

7747 **CK_XEDDSA_PARAMS** is a structure that provides the parameters for the **CKM_XEDDSA** signature
 7748 mechanism. The structure is defined as follows:

```
7749     typedef struct CK_XEDDSA_PARAMS {
7750         CK_XEDDSA_HASH_TYPE hash;
7751     } CK_XEDDSA_PARAMS;
```

7752

7753 The fields of the structure have the following meanings:

7754 hash a Hash mechanism to be used by the mechanism.

7755 **CK_XEDDSA_PARAMS_PTR** is a pointer to a **CK_XEDDSA_PARAMS**.

7756

7757 ♦ **CK_XEDDSA_HASH_TYPE**, **CK_XEDDSA_HASH_TYPE_PTR**

7758 **CK_XEDDSA_HASH_TYPE** is used to indicate the hash function used in XEDDSA. It is defined as
 7759 follows:

```
7760     typedef CK ULONG CK_XEDDSA_HASH_TYPE;
```

7761

7762 The following table lists the defined functions.

7763 *Table 76, EC: Key Derivation Functions*

| Source Identifier |
|-------------------|
| CKM_BLAKE2B_256 |
| CKM_BLAKE2B_512 |
| CKM_SHA3_256 |
| CKM_SHA3_512 |
| CKM_SHA256 |
| CKM_SHA512 |

7764

7765 **CK_XEDDSA_HASH_TYPE_PTR** is a pointer to a **CK_XEDDSA_HASH_TYPE**.

7766

7767 ♦ **CK_EC_KDF_TYPE, CK_EC_KDF_TYPE_PTR**

7768 **CK_EC_KDF_TYPE** is used to indicate the Key Derivation Function (KDF) applied to derive keying data
7769 from a shared secret. The key derivation function will be used by the EC key agreement schemes. It is
7770 defined as follows:

7771 **typedef CK ULONG CK_EC_KDF_TYPE;**

7772

7773 The following table lists the defined functions.

7774 *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 |

7775 The key derivation function **CKD_NULL** produces a raw shared secret value without applying any key
 7776 derivation function.

7777 The key derivation functions
 7778 **CKD_[SHA1|SHA224|SHA384|SHA512|SHA3_224|SHA3_256|SHA3_384|SHA3_512]_KDF**, which are
 7779 based on SHA-1, SHA-224, SHA-384, SHA-512, SHA3-224, SHA3-256, SHA3-384, SHA3-512
 7780 respectively, derive keying data from the shared secret value as defined in [ANSI X9.63].

7781 The key derivation functions
 7782 **CKD_[SHA1|SHA224|SHA384|SHA512|SHA3_224|SHA3_256|SHA3_384|SHA3_512]_KDF_SP800**,
 7783 which are based on SHA-1, SHA-224, SHA-384, SHA-512, SHA3-224, SHA3-256, SHA3-384, SHA3-512
 7784 respectively, derive keying data from the shared secret value as defined in [FIPS SP800-56A] section
 7785 5.8.1.1.

7786 The key derivation functions **CKD_BLAKE2B_[160|256|384|512]_KDF**, which are based on the Blake2b
 7787 family of hashes, derive keying data from the shared secret value as defined in [FIPS SP800-56A] section
 7788 5.8.1.1. **CK_EC_KDF_TYPE_PTR** is a pointer to a **CK_EC_KDF_TYPE**.

7789

7790 ♦ **CK_ECDH1_DERIVE_PARAMS**, **CK_ECDH1_DERIVE_PARAMS_PTR**

7791 **CK_ECDH1_DERIVE_PARAMS** is a structure that provides the parameters for the
 7792 **CKM_ECDH1_DERIVE** and **CKM_ECDH1_COFACTOR_DERIVE** key derivation mechanisms, where
 7793 each party contributes one key pair. The structure is defined as follows:

```
7794     typedef struct CK_ECDH1_DERIVE_PARAMS {
7795         CK_EC_KDF_TYPE    kdf;
7796         CK ULONG          ulSharedDataLen;
7797         CK BYTE PTR       pSharedData;
7798         CK ULONG          ulPublicDataLen;
7799         CK BYTE PTR       pPublicData;
7800     } CK_ECDH1_DERIVE_PARAMS;
```

7801

7802 The fields of the structure have the following meanings:

| | |
|---|---|
| 7803 kdf | key derivation function used on the shared secret value |
| 7804 ulSharedDataLen | the length in bytes of the shared info |
| 7805 pSharedData | some data shared between the two parties |
| 7806 ulPublicDataLen | the length in bytes of the other party's EC public key |

7807 pPublicData¹ pointer to other party's EC public key value. For short Weierstrass
 7808 EC keys: a token MUST be able to accept this value encoded as a
 7809 raw octet string (as per section A.5.2 of [ANSI X9.62]). A token
 7810 MAY, in addition, support accepting this value as a DER-encoded
 7811 ECPoint (as per section E.6 of [ANSI X9.62]) i.e. the same as a
 7812 CKA_EC_POINT encoding. The calling application is responsible
 7813 for converting the offered public key to the compressed or
 7814 uncompressed forms of these encodings if the token does not
 7815 support the offered form.
 7816 For Montgomery keys: the public key is provided as bytes in little
 7817 endian order as defined in RFC 7748.

7818 With the key derivation function **CKD_NULL**, *pSharedData* must be NULL and *ulSharedDataLen* must be zero. With the key derivation functions
 7819 **CKD_[SHA1|SHA224|SHA384|SHA512|SHA3_224|SHA3_256|SHA3_384|SHA3_512]_KDF**,
 7820 **CKD_[SHA1|SHA224|SHA384|SHA512|SHA3_224|SHA3_256|SHA3_384|SHA3_512]_KDF_SP800**, an
 7821 optional *pSharedData* may be supplied, which consists of some data shared by the two parties intending
 7822 to share the shared secret. Otherwise, *pSharedData* must be NULL and *ulSharedDataLen* must be zero.
 7823

7824 **CK_ECDH1_DERIVE_PARAMS_PTR** is a pointer to a **CK_ECDH1_DERIVE_PARAMS**.
 7825 ♦ **CK_ECDH2_DERIVE_PARAMS**, **CK_ECDH2_DERIVE_PARAMS_PTR**

7826 **CK_ECDH2_DERIVE_PARAMS** is a structure that provides the parameters to the
 7827 **CKM_ECMQV_DERIVE** key derivation mechanism, where each party contributes two key pairs. The
 7828 structure is defined as follows:

```

    7829                    typedef struct CK_ECDH2_DERIVE_PARAMS {
    7830                        CK_EKDF_TYPE kdf;
    7831                        CK ULONG ulSharedDataLen;
    7832                        CK_BYTE_PTR pSharedData;
    7833                        CK ULONG ulPublicDataLen;
    7834                        CK_BYTE_PTR pPublicData;
    7835                        CK ULONG ulPrivateDataLen;
    7836                        CK_OBJECT_HANDLE hPrivateData;
    7837                        CK ULONG ulPublicDataLen2;
    7838                        CK_BYTE_PTR pPublicData2;
    7839                    } CK_ECDH2_DERIVE_PARAMS;
  
```

7840

7841 The fields of the structure have the following meanings:

| | |
|--|--|
| 7842 kdf | key derivation function used on the shared secret value |
| 7843 ulSharedDataLen | the length in bytes of the shared info |
| 7844 pSharedData | some data shared between the two parties |
| 7845 ulPublicDataLen | the length in bytes of the other party's first EC public key |
| 7846 pPublicData | pointer to other party's first EC public key value. Encoding rules are as per pPublicData of CK_ECDH1_DERIVE_PARAMS |
| 7848 ulPrivateDataLen | the length in bytes of the second EC private key |

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.

7849 hPrivateData key handle for second EC private key value
7850 ulPublicDataLen2 the length in bytes of the other party's second EC public key
7851 pPublicData2 pointer to other party's second EC public key value. Encoding rules
7852 are as per pPublicData of CK_ECDH1_DERIVE_PARAMS

7853 With the key derivation function **CKD_NULL**, *pSharedData* must be NULL and *ulSharedDataLen* must be zero. With the key derivation function **CKD_SHA1_KDF**, an optional *pSharedData* may be supplied, which consists of some data shared by the two parties intending to share the shared secret. Otherwise, *pSharedData* must be NULL and *ulSharedDataLen* must be zero.

7854 **CK_ECDH2_DERIVE_PARAMS_PTR** is a pointer to a **CK_ECDH2_DERIVE_PARAMS**.

7855

7856 ♦ **CK_ECMQV_DERIVE_PARAMS, CK_ECMQV_DERIVE_PARAMS_PTR**

7857 **CK_ECMQV_DERIVE_PARAMS** is a structure that provides the parameters to the
7858 **CKM_ECMQV_DERIVE** key derivation mechanism, where each party contributes two key pairs. The
7859 structure is defined as follows:

```
7860      typedef struct CK_ECMQV_DERIVE_PARAMS {  
7861          CK_EC_KDF_TYPE      kdf;  
7862          CK ULONG            ulSharedDataLen;  
7863          CK_BYTE_PTR        pSharedData;  
7864          CK ULONG            ulPublicDataLen;  
7865          CK_BYTE_PTR        pPublicData;  
7866          CK ULONG            ulPrivateKeyLen;  
7867          CK_OBJECT_HANDLE   hPrivateKey;  
7868          CK ULONG            ulPublicKeyLen;  
7869          CK_BYTE_PTR        pPublicKey;  
7870          CK_OBJECT_HANDLE   publicKey;  
7871      } CK_ECMQV_DERIVE_PARAMS;
```

7872

7873 The fields of the structure have the following meanings:

7874 kdf key derivation function used on the shared secret value
7875 ulSharedDataLen the length in bytes of the shared info
7876 pSharedData some data shared between the two parties
7877 ulPublicDataLen the length in bytes of the other party's first EC public key
7878 pPublicData pointer to other party's first EC public key value. Encoding rules are
7879 as per pPublicData of CK_ECDH1_DERIVE_PARAMS
7880 ulPrivateKeyLen the length in bytes of the second EC private key
7881 hPrivateKey key handle for second EC private key value
7882 ulPublicKeyLen the length in bytes of the other party's second EC public key
7883 pPublicKey pointer to other party's second EC public key value. Encoding rules
7884 are as per pPublicKey of CK_ECDH1_DERIVE_PARAMS
7885 publicKey Handle to the first party's ephemeral public key

7886 With the key derivation function **CKD_NULL**, *pSharedData* must be NULL and *ulSharedDataLen* must be zero. With the key derivation functions
7887 **CKD_[SHA1|SHA224|SHA384|SHA512|SHA3_224|SHA3_256|SHA3_384|SHA3_512]_KDF**,
7888 **CKD_[SHA1|SHA224|SHA384|SHA512|SHA3_224|SHA3_256|SHA3_384|SHA3_512]_KDF_SP800**, an

7893 optional *pSharedData* may be supplied, which consists of some data shared by the two parties intending
7894 to share the shared secret. Otherwise, *pSharedData* must be NULL and *ulSharedDataLen* must be zero.
7895 **CK_ECMQV_DERIVE_PARAMS_PTR** is a pointer to a **CK_ECMQV_DERIVE_PARAMS**.

7896 6.3.17 Elliptic Curve Diffie-Hellman key derivation

7897 The Elliptic Curve Diffie-Hellman (ECDH) key derivation mechanism, denoted **CKM_ECDH1_DERIVE**, is
7898 a mechanism for key derivation based on the Diffie-Hellman version of the Elliptic Curve key agreement
7899 scheme, as defined in ANSI X9.63 for short Weierstrass EC keys and RFC 7748 for Montgomery keys,
7900 where each party contributes one key pair all using the same EC domain parameters.

7901 It has a parameter, a **CK_ECDH1_DERIVE_PARAMS** structure.

7902 This mechanism derives a secret value, and truncates the result according to the **CKA_KEY_TYPE**
7903 attribute of the template and, if it has one and the key type supports it, the **CKA_VALUE_LEN** attribute of
7904 the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism
7905 contributes the result as the **CKA_VALUE** attribute of the new key; other attributes required by the key
7906 type must be specified in the template.

7907 This mechanism has the following rules about key sensitivity and extractability:

- 7908 • The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both
7909 be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some
7910 default value.
- 7911 • If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key
7912 will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the
7913 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
7914 **CKA_SENSITIVE** attribute.
- 7915 • Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
7916 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
7917 CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
7918 value from its **CKA_EXTRACTABLE** attribute.

7919 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7920 specify the minimum and maximum supported number of bits in the field sizes, respectively. For
7921 example, if a Cryptoki library supports only EC using a field of characteristic 2 which has between 2^{200}
7922 and 2^{300} elements, then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in binary notation,
7923 the number 2^{200} consists of a 1 bit followed by 200 0 bits. It is therefore a 201-bit number. Similarly, 2^{300}
7924 is a 301-bit number).

7925 Constraints on key types are summarized in the following table:

7926 Table 78: ECDH: Allowed Key Types

| Function | Key type |
|----------|-----------------------------|
| C_Derive | CKK_EC or CKK_EC_MONTGOMERY |

7927 6.3.18 Elliptic Curve Diffie-Hellman with cofactor key derivation

7928 The Elliptic Curve Diffie-Hellman (ECDH) with cofactor key derivation mechanism, denoted
7929 **CKM_ECDH1_COFACTOR_DERIVE**, is a mechanism for key derivation based on the cofactor Diffie-
7930 Hellman version of the Elliptic Curve key agreement scheme, as defined in ANSI X9.63, where each party
7931 contributes one key pair all using the same EC domain parameters. Cofactor multiplication is
7932 computationally efficient and helps to prevent security problems like small group attacks.

7933 It has a parameter, a **CK_ECDH1_DERIVE_PARAMS** structure.

7934 This mechanism derives a secret value, and truncates the result according to the **CKA_KEY_TYPE**
7935 attribute of the template and, if it has one and the key type supports it, the **CKA_VALUE_LEN** attribute of
7936 the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism
7937 contributes the result as the **CKA_VALUE** attribute of the new key; other attributes required by the key
7938 type must be specified in the template.

7939 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.

7951 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7952 specify the minimum and maximum supported number of bits in the field sizes, respectively. For
7953 example, if a Cryptoki library supports only EC using a field of characteristic 2 which has between 2^{200}
7954 and 2^{300} elements, then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in binary notation,
7955 the number 2^{200} consists of a 1 bit followed by 200 0 bits. It is therefore a 201-bit number. Similarly, 2^{300}
7956 is a 301-bit number).

7957 Constraints on key types are summarized in the following table:

7958 *Table 79: ECDH with cofactor: Allowed Key Types*

| Function | Key type |
|----------|----------|
| C_Derive | CKK_EC |

7959 6.3.19 Elliptic Curve Menezes-Qu-Vanstone key derivation

7960 The Elliptic Curve Menezes-Qu-Vanstone (ECMQV) key derivation mechanism, denoted
7961 **CKM_ECMQV_DERIVE**, is a mechanism for key derivation based the MQV version of the Elliptic Curve
7962 key agreement scheme, as defined in ANSI X9.63, where each party contributes two key pairs all using
7963 the same EC domain parameters.

7964 It has a parameter, a **CK_ECMQV_DERIVE_PARAMS** structure.

7965 This mechanism derives a secret value, and truncates the result according to the **CKA_KEY_TYPE**
7966 attribute of the template and, if it has one and the key type supports it, the **CKA_VALUE_LEN** attribute of
7967 the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism
7968 contributes the result as the **CKA_VALUE** attribute of the new key; other attributes required by the key
7969 type must be specified in the template.

7970 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.

7982 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7983 specify the minimum and maximum supported number of bits in the field sizes, respectively. For
7984 example, if a Cryptoki library supports only EC using a field of characteristic 2 which has between 2^{200}
7985 and 2^{300} elements, then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in binary notation,

7986 the number 2^{200} consists of a 1 bit followed by 200 0 bits. It is therefore a 201-bit number. Similarly, 2^{300}
7987 is a 301-bit number).

7988 Constraints on key types are summarized in the following table:

7989 *Table 80: ECDH MQV: Allowed Key Types*

| Function | Key type |
|----------|----------|
| C_Derive | CKK_EC |

7990 6.3.20 ECDH AES KEY WRAP

7991 The ECDH AES KEY WRAP mechanism, denoted **CKM_ECDH_AES_KEY_WRAP**, is a mechanism
7992 based on Elliptic Curve public-key crypto-system and the AES key wrap mechanism. It supports single-
7993 part key wrapping; and key unwrapping.

7994 It has a parameter, a **CK_ECDH_AES_KEY_WRAP_PARAMS** structure.

7995

7996 The mechanism can wrap and unwrap an asymmetric target key of any length and type using an EC
7997 key.

- 7998 - A temporary AES key is derived from a temporary EC key and the wrapping EC key
7999 using the **CKM_ECDH1_DERIVE** mechanism.
8000 - The derived AES key is used for wrapping the target key using the
8001 **CKM_AES_KEY_WRAP_KWP** mechanism.

8002

8003 For wrapping, the mechanism -

- 8004 • Generates a temporary random EC key (transport key) having the same parameters as the
8005 wrapping EC key (and domain parameters). Saves the transport key public key material.
8006 • Performs ECDH operation using **CKM_ECDH1_DERIVE** with parameters of kdf, ulSharedDataLen
8007 and pSharedData using the private key of the transport EC key and the public key of wrapping EC
8008 key and gets the first ulAESKeyBits bits of the derived key to be the temporary AES key.
8009 • Wraps the target key with the temporary AES key using **CKM_AES_KEY_WRAP_KWP**.
8010 • Zeroizes the temporary AES key and EC transport private key.
8011 • Concatenates public key material of the transport key and output the concatenated blob. The first
8012 part is the public key material of the transport key and the second part is the wrapped target key.

8013

8014 The private target key will be encoded as defined in section 6.7.

8015

8016 The use of Attributes in the PrivateKeyInfo structure is OPTIONAL. In case of conflicts between the
8017 object attribute template, and Attributes in the PrivateKeyInfo structure, an error should be thrown.

8018

8019 For unwrapping, the mechanism -

- 8020 • Splits the input into two parts. The first part is the public key material of the transport key and the
8021 second part is the wrapped target key. The length of the first part is equal to the length of the public
8022 key material of the unwrapping EC key.
8023 *Note: since the transport key and the wrapping EC key share the same domain, the length of the*
8024 *public key material of the transport key is the same length of the public key material of the*
8025 *unwrapping EC key.*
- 8026 • Performs ECDH operation using **CKM_ECDH1_DERIVE** with parameters of kdf, ulSharedDataLen
8027 and pSharedData using the private part of unwrapping EC key and the public part of the transport
8028 EC key and gets first ulAESKeyBits bits of the derived key to be the temporary AES key.

- 8029 • Un-wraps the target key from the second part with the temporary AES key using
 8030 **CKM_AES_KEY_WRAP_KWP**.
 8031 • Zeroizes the temporary AES key.

8032

8033 *Table 81, CKM_ECDH_AES_KEY_WRAP Mechanisms vs. Functions*

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

¹SR = SignRecover, VR = VerifyRecover

8034

8035 Constraints on key types are summarized in the following table:

8036 *Table 82: ECDH AES Key Wrap: Allowed Key Types*

| Function | Key type |
|----------|-----------------------------|
| C_Wrap / | CKK_EC or CKK_EC_MONTGOMERY |
| C_Unwrap | |

8037 6.3.21 ECDH AES KEY WRAP mechanism parameters

8038 ♦ **CK_ECDH_AES_KEY_WRAP_PARAMS; CK_ECDH_AES_KEY_WRAP_PARAMS_PTR**

8039 **CK_ECDH_AES_KEY_WRAP_PARAMS** is a structure that provides the parameters to the
 8040 **CKM_ECDH_AES_KEY_WRAP** mechanism. It is defined as follows:

```
8041
8042     typedef struct CK_ECDH_AES_KEY_WRAP_PARAMS {
8043         CK ULONG          ulAESKeyBits;
8044         CK EC KDF TYPE   kdf;
8045         CK ULONG          ulSharedDataLen;
8046         CK BYTE PTR      pSharedData;
8047     } CK_ECDH_AES_KEY_WRAP_PARAMS;
```

8048 The fields of the structure have the following meanings:

8049 ulAESKeyBits length of the temporary AES key in bits. Can be only 128, 192 or
 8050 256.

8051 kdf key derivation function used on the shared secret value to generate
 8052 AES key.

8053 ulSharedDataLen the length in bytes of the shared info

8054 pSharedData Some data shared between the two parties

8055

8056 **CK_ECDH_AES_KEY_WRAP_PARAMS_PTR** is a pointer to a
 8057 **CK_ECDH_AES_KEY_WRAP_PARAMS**.

8058

8061 6.3.22 FIPS 186-4

8062 When CKM_ECDSA is operated in FIPS mode, the curves SHALL either be NIST recommended curves
8063 (with a fixed set of domain parameters) or curves with domain parameters generated as specified by
8064 ANSI X9.64. The NIST recommended curves are:

8065

8066 P-192, P-224, P-256, P-384, P-521
8067 K-163, B-163, K-233, B-233
8068 K-283, B-283, K-409, B-409
8069 K-571, B-571

8070 6.4 Diffie-Hellman

8071 *Table 83, Diffie-Hellman Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|----------------------------|-------------------|---------------|----------------------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_DH_PKCS_KEY_PAIR_GEN | | | | | ✓ | | |
| CKM_DH_PKCS_PARAMETER_GEN | | | | | ✓ | | |
| CKM_DH_PKCS_DERIVE | | | | | | | ✓ |
| CKM_X9_42_DH_KEY_PAIR_GEN | | | | | ✓ | | |
| CKM_X9_42_DH_PARAMETER_GEN | | | | | ✓ | | |
| CKM_X9_42_DH_DERIVE | | | | | | | ✓ |
| CKM_X9_42_DH_HYBRID_DERIVE | | | | | | | ✓ |
| CKM_X9_42_MQV_DERIVE | | | | | | | ✓ |

8072 6.4.1 Definitions

8073 This section defines the key type “CKK_DH” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE
8074 attribute of [DH] key objects.

8075 Mechanisms:

8076 CKM_DH_PKCS_KEY_PAIR_GEN
8077 CKM_DH_PKCS_PARAMETER_GEN
8078 CKM_DH_PKCS_DERIVE
8079 CKM_X9_42_DH_KEY_PAIR_GEN
8080 CKM_X9_42_DH_PARAMETER_GEN
8081 CKM_X9_42_DH_DERIVE
8082 CKM_X9_42_DH_HYBRID_DERIVE
8083 CKM_X9_42_MQV_DERIVE
8084

8085 6.4.2 Diffie-Hellman public key objects

8086 Diffie-Hellman public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_DH**) hold Diffie-
8087 Hellman public keys. The following table defines the Diffie-Hellman public key object attributes, in
8088 addition to the common attributes defined for this object class:

8089 *Table 84, Diffie-Hellman Public Key Object Attributes*

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

8090 Refer to Table 11 for footnotes

8091 The **CKA_PRIME** and **CKA_BASE** attribute values are collectively the “Diffie-Hellman domain
8092 parameters”. Depending on the token, there may be limits on the length of the key components. See
8093 [PKCS #3] for more information on Diffie-Hellman keys.

8094 The following is a sample template for creating a Diffie-Hellman public key object:

```
8095 CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
8096 CK_KEY_TYPE keyType = CKK_DH;
8097 CK_UTF8CHAR label[] = "A Diffie-Hellman public key object";
8098 CK_BYTE prime[] = {...};
8099 CK_BYTE base[] = {...};
8100 CK_BYTE value[] = {...};
8101 CK_BBOOL true = CK_TRUE;
8102 CK_ATTRIBUTE template[] = {
8103     {CKA_CLASS, &class, sizeof(class)},
8104     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
8105     {CKA_TOKEN, &true, sizeof(true)},
8106     {CKA_LABEL, label, sizeof(label)-1},
8107     {CKA_PRIME, prime, sizeof(prime)},
8108     {CKA_BASE, base, sizeof(base)},
8109     {CKA_VALUE, value, sizeof(value)}}
8110 };
```

8111 6.4.3 X9.42 Diffie-Hellman public key objects

8112 X9.42 Diffie-Hellman public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_X9_42_DH**)
8113 hold X9.42 Diffie-Hellman public keys. The following table defines the X9.42 Diffie-Hellman public key
8114 object attributes, in addition to the common attributes defined for this object class:

8115 *Table 85, X9.42 Diffie-Hellman Public Key Object Attributes*

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

8116 Refer to Table 11 for footnotes

8117 The **CKA_PRIME**, **CKA_BASE** and **CKA_SUBPRIME** attribute values are collectively the “X9.42 Diffie-
8118 Hellman domain parameters”. See the ANSI X9.42 standard for more information on X9.42 Diffie-
8119 Hellman keys.

8120 The following is a sample template for creating a X9.42 Diffie-Hellman public key object:

```
8121     CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
8122     CK_KEY_TYPE keyType = CKK_X9_42_DH;
8123     CK_UTF8CHAR label[] = "A X9.42 Diffie-Hellman public key
8124         object";
8125     CK_BYTE prime[] = {...};
8126     CK_BYTE base[] = {...};
8127     CK_BYTE subprime[] = {...};
8128     CK_BYTE value[] = {...};
8129     CK_BBOOL true = CK_TRUE;
8130     CK_ATTRIBUTE template[] = {
8131         {CKA_CLASS, &class, sizeof(class)},
8132         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
8133         {CKA_TOKEN, &true, sizeof(true)},
8134         {CKA_LABEL, label, sizeof(label)-1},
8135         {CKA_PRIME, prime, sizeof(prime)},
8136         {CKA_BASE, base, sizeof(base)},
8137         {CKA_SUBPRIME, subprime, sizeof(subprime)},
8138         {CKA_VALUE, value, sizeof(value)}
8139     };
```

8140 6.4.4 Diffie-Hellman private key objects

8141 Diffie-Hellman private key objects (object class **CKO_PRIVATE_KEY**, key type **CKK_DH**) hold Diffie-
8142 Hellman private keys. The following table defines the Diffie-Hellman private key object attributes, in
8143 addition to the common attributes defined for this object class:

8144 *Table 86, Diffie-Hellman Private Key Object Attributes*

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

8145 ^ Refer to Table 11 for footnotes

8146 The **CKA_PRIME** and **CKA_BASE** attribute values are collectively the “Diffie-Hellman domain
8147 parameters”. Depending on the token, there may be limits on the length of the key components. See
8148 [PKCS #3] for more information on Diffie-Hellman keys.

8149 Note that when generating a Diffie-Hellman private key, the Diffie-Hellman parameters are *not* specified in
8150 the key’s template. This is because Diffie-Hellman private keys are only generated as part of a Diffie-
8151 Hellman key pair, and the Diffie-Hellman parameters for the pair are specified in the template for the
8152 Diffie-Hellman public key.

8153 The following is a sample template for creating a Diffie-Hellman private key object:

```
8154     CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
8155     CK_KEY_TYPE keyType = CKK_DH;
8156     CK_UTF8CHAR label[] = "A Diffie-Hellman private key object";
8157     CK_BYTE subject[] = {...};
8158     CK_BYTE id[] = {123};
8159     CK_BYTE prime[] = {...};
8160     CK_BYTE base[] = {...};
```

```

8161 CK_BYTE value[] = { ... };
8162 CK_BBOOL true = CK_TRUE;
8163 CK_ATTRIBUTE template[] = {
8164     {CKA_CLASS, &class, sizeof(class)},
8165     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
8166     {CKA_TOKEN, &true, sizeof(true)},
8167     {CKA_LABEL, label, sizeof(label)-1},
8168     {CKA SUBJECT, subject, sizeof(subject)},
8169     {CKA_ID, id, sizeof(id)},
8170     {CKA_SENSITIVE, &true, sizeof(true)},
8171     {CKA_DERIVE, &true, sizeof(true)},
8172     {CKA_PRIME, prime, sizeof(prime)},
8173     {CKA_BASE, base, sizeof(base)},
8174     {CKA_VALUE, value, sizeof(value)}
8175 };
```

8176 6.4.5 X9.42 Diffie-Hellman private key objects

8177 X9.42 Diffie-Hellman private key objects (object class **CKO_PRIVATE_KEY**, key type **CKK_X9_42_DH**)
8178 hold X9.42 Diffie-Hellman private keys. The following table defines the X9.42 Diffie-Hellman private key
8179 object attributes, in addition to the common attributes defined for this object class:

8180 *Table 87, X9.42 Diffie-Hellman Private Key Object Attributes*

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

8181 Refer to Table 11 for footnotes

8182 The **CKA_PRIME**, **CKA_BASE** and **CKA_SUBPRIME** attribute values are collectively the “X9.42 Diffie-
8183 Hellman domain parameters”. Depending on the token, there may be limits on the length of the key
8184 components. See the ANSI X9.42 standard for more information on X9.42 Diffie-Hellman keys.

8185 Note that when generating a X9.42 Diffie-Hellman private key, the X9.42 Diffie-Hellman domain
8186 parameters are *not* specified in the key’s template. This is because X9.42 Diffie-Hellman private keys are
8187 only generated as part of a X9.42 Diffie-Hellman key pair, and the X9.42 Diffie-Hellman domain
8188 parameters for the pair are specified in the template for the X9.42 Diffie-Hellman public key.

8189 The following is a sample template for creating a X9.42 Diffie-Hellman private key object:

```

8190 CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
8191 CK_KEY_TYPE keyType = CKK_X9_42_DH;
8192 CK_UTF8CHAR label[] = "A X9.42 Diffie-Hellman private key object";
8193 CK_BYTE subject[] = { ... };
8194 CK_BYTE id[] = {123};
8195 CK_BYTE prime[] = { ... };
8196 CK_BYTE base[] = { ... };
8197 CK_BYTE subprime[] = { ... };
8198 CK_BYTE value[] = { ... };
8199 CK_BBOOL true = CK_TRUE;
8200 CK_ATTRIBUTE template[] = {
8201     {CKA_CLASS, &class, sizeof(class)},
```

```

8202     { CKA_KEY_TYPE, &keyType, sizeof(keyType) },
8203     { CKA_TOKEN, &true, sizeof(true) },
8204     { CKA_LABEL, label, sizeof(label)-1 },
8205     { CKA SUBJECT, subject, sizeof(subject) },
8206     { CKA_ID, id, sizeof(id) },
8207     { CKA_SENSITIVE, &true, sizeof(true) },
8208     { CKA_DERIVE, &true, sizeof(true) },
8209     { CKA_PRIME, prime, sizeof(prime) },
8210     { CKA_BASE, base, sizeof(base) },
8211     { CKA_SUBPRIME, subprime, sizeof(subprime) },
8212     { CKA_VALUE, value, sizeof(value) }
8213 };

```

8214 6.4.6 Diffie-Hellman domain parameter objects

8215 Diffie-Hellman domain parameter objects (object class **CKO_DOMAIN_PARAMETERS**, key type
 8216 **CKK_DH**) hold Diffie-Hellman domain parameters. The following table defines the Diffie-Hellman domain
 8217 parameter object attributes, in addition to the common attributes defined for this object class:

8218 *Table 88, Diffie-Hellman Domain Parameter Object Attributes*

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

8219 ¹ Refer to Table 11 for footnotes

8220 The **CKA_PRIME** and **CKA_BASE** attribute values are collectively the “Diffie-Hellman domain
 8221 parameters”. Depending on the token, there may be limits on the length of the key components. See
 8222 [PKCS #3] for more information on Diffie-Hellman domain parameters.

8223 The following is a sample template for creating a Diffie-Hellman domain parameter object:

```

8224     CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
8225     CK_KEY_TYPE keyType = CKK_DH;
8226     CK_UTF8CHAR label[] = "A Diffie-Hellman domain parameters
8227         object";
8228     CK_BYTE prime[] = {...};
8229     CK_BYTE base[] = {...};
8230     CK_BBOOL true = CK_TRUE;
8231     CK_ATTRIBUTE template[] = {
8232         { CKA_CLASS, &class, sizeof(class) },
8233         { CKA_KEY_TYPE, &keyType, sizeof(keyType) },
8234         { CKA_TOKEN, &true, sizeof(true) },
8235         { CKA_LABEL, label, sizeof(label)-1 },
8236         { CKA_PRIME, prime, sizeof(prime) },
8237         { CKA_BASE, base, sizeof(base) },
8238     };

```

8239 6.4.7 X9.42 Diffie-Hellman domain parameters objects

8240 X9.42 Diffie-Hellman domain parameters objects (object class **CKO_DOMAIN_PARAMETERS**, key type
 8241 **CKK_X9_42_DH**) hold X9.42 Diffie-Hellman domain parameters. The following table defines the X9.42

8242 Diffie-Hellman domain parameters object attributes, in addition to the common attributes defined for this
8243 object class:

8244 *Table 89, X9.42 Diffie-Hellman Domain Parameters Object Attributes*

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

8245 Refer to Table 11 for footnotes

8246 The **CKA_PRIME**, **CKA_BASE** and **CKA_SUBPRIME** attribute values are collectively the “X9.42 Diffie-
8247 Hellman domain parameters”. Depending on the token, there may be limits on the length of the domain
8248 parameters components. See the ANSI X9.42 standard for more information on X9.42 Diffie-Hellman
8249 domain parameters.

8250 The following is a sample template for creating a X9.42 Diffie-Hellman domain parameters object:

```
8251 CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
8252 CK_KEY_TYPE keyType = CKK_X9_42_DH;
8253 CK_UTF8CHAR label[] = "A X9.42 Diffie-Hellman domain
8254     parameters object";
8255 CK_BYTE prime[] = {...};
8256 CK_BYTE base[] = {...};
8257 CK_BYTE subprime[] = {...};
8258 CK_BBOOL true = CK_TRUE;
8259 CK_ATTRIBUTE template[] = {
8260     {CKA_CLASS, &class, sizeof(class)},
8261     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
8262     {CKA_TOKEN, &true, sizeof(true)},
8263     {CKA_LABEL, label, sizeof(label)-1},
8264     {CKA_PRIME, prime, sizeof(prime)},
8265     {CKA_BASE, base, sizeof(base)},
8266     {CKA_SUBPRIME, subprime, sizeof(subprime)},
8267 };
```

8268 6.4.8 PKCS #3 Diffie-Hellman key pair generation

8269 The PKCS #3 Diffie-Hellman key pair generation mechanism, denoted
8270 **CKM_DH_PKCS_KEY_PAIR_GEN**, is a key pair generation mechanism based on Diffie-Hellman key
8271 agreement, as defined in [PKCS #3]. This is what PKCS #3 calls “phase I”. It does not have a
8272 parameter.

8273 The mechanism generates Diffie-Hellman public/private key pairs with a particular prime and base, as
8274 specified in the **CKA_PRIME** and **CKA_BASE** attributes of the template for the public key. If the
8275 **CKA_VALUE_BITS** attribute of the private key is specified, the mechanism limits the length in bits of the
8276 private value, as described in [PKCS #3].

8277 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
8278 public key and the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_BASE**, and **CKA_VALUE** (and
8279 the **CKA_VALUE_BITS** attribute, if it is not already provided in the template) attributes to the new private
8280 key; other attributes required by the Diffie-Hellman public and private key types must be specified in the
8281 templates.

8282 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
8283 specify the supported range of Diffie-Hellman prime sizes, in bits.

8284 **6.4.9 PKCS #3 Diffie-Hellman domain parameter generation**

8285 The PKCS #3 Diffie-Hellman domain parameter generation mechanism, denoted
8286 **CKM_DH_PKCS_PARAMETER_GEN**, is a domain parameter generation mechanism based on Diffie-
8287 Hellman key agreement, as defined in [PKCS #3].

8288 It does not have a parameter.

8289 The mechanism generates Diffie-Hellman domain parameters with a particular prime length in bits, as
8290 specified in the **CKA_PRIME_BITS** attribute of the template.

8291 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_BASE**, and
8292 **CKA_PRIME_BITS** attributes to the new object. Other attributes supported by the Diffie-Hellman domain
8293 parameter types may also be specified in the template, or else are assigned default initial values.

8294 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
8295 specify the supported range of Diffie-Hellman prime sizes, in bits.

8296 **6.4.10 PKCS #3 Diffie-Hellman key derivation**

8297 The PKCS #3 Diffie-Hellman key derivation mechanism, denoted **CKM_DH_PKCS_DERIVE**, is a
8298 mechanism for key derivation based on Diffie-Hellman key agreement, as defined in [PKCS #3]. This is
8299 what PKCS #3 calls “phase II”.

8300 It has a parameter, which is the public value of the other party in the key agreement protocol, represented
8301 as a Cryptoki “Big integer” (*i.e.*, a sequence of bytes, most-significant byte first).

8302 This mechanism derives a secret key from a Diffie-Hellman private key and the public value of the other
8303 party. It computes a Diffie-Hellman secret value from the public value and private key according to
8304 [PKCS #3], and truncates the result according to the **CKA_KEY_TYPE** attribute of the template and, if it
8305 has one and the key type supports it, the **CKA_VALUE_LEN** attribute of the template. (The truncation
8306 removes bytes from the leading end of the secret value.) The mechanism contributes the result as the
8307 **CKA_VALUE** attribute of the new key; other attributes required by the key type must be specified in the
8308 template.

8309 This mechanism has the following rules about key sensitivity and extractability²:

- 8310 • The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both
8311 be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some
8312 default value.
- 8313 • If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key
8314 will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the
8315 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
8316 **CKA_SENSITIVE** attribute.
- 8317 • Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
8318 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
8319 CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
8320 value from its **CKA_EXTRACTABLE** attribute.

8321 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
8322 specify the supported range of Diffie-Hellman prime sizes, in bits.

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

8323 **6.4.11 X9.42 Diffie-Hellman mechanism parameters**

- 8324 ◆ **CK_X9_42_DH_KDF_TYPE, CK_X9_42_DH_KDF_TYPE_PTR**

8325 **CK_X9_42_DH_KDF_TYPE** is used to indicate the Key Derivation Function (KDF) applied to derive
8326 keying data from a shared secret. The key derivation function will be used by the X9.42 Diffie-Hellman
8327 key agreement schemes. It is defined as follows:

8328 **typedef CK ULONG CK_X9_42_DH_KDF_TYPE;**

8329 The following table lists the defined functions.

8330 *Table 90, X9.42 Diffie-Hellman Key Derivation Functions*

| Source Identifier |
|---------------------------------|
| CKD_NULL |
| CKD_SHA1_KDF ASN1 |
| CKD_SHA1_KDF CONCATENATE |

8332 The key derivation function **CKD_NULL** produces a raw shared secret value without applying any key
8333 derivation function whereas the key derivation functions **CKD_SHA1_KDF ASN1** and
8334 **CKD_SHA1_KDF CONCATENATE**, which are both based on SHA-1, derive keying data from the
8335 shared secret value as defined in the ANSI X9.42 standard.

8336 **CK_X9_42_DH_KDF_TYPE_PTR** is a pointer to a **CK_X9_42_DH_KDF_TYPE**.

- 8337 ◆ **CK_X9_42_DH1_DERIVE_PARAMS, CK_X9_42_DH1_DERIVE_PARAMS_PTR**

8338 **CK_X9_42_DH1_DERIVE_PARAMS** is a structure that provides the parameters to the
8339 **CKM_X9_42_DH_DERIVE** key derivation mechanism, where each party contributes one key pair. The
8340 structure is defined as follows:

8341 **typedef struct CK_X9_42_DH1_DERIVE_PARAMS {**
8342 **CK_X9_42_DH_KDF_TYPE kdf;**
8343 **CK ULONG ulOtherInfoLen;**
8344 **CK_BYTE_PTR pOtherInfo;**
8345 **CK ULONG ulPublicDataLen;**
8346 **CK_BYTE_PTR pPublicData;**
8347 **} CK_X9_42_DH1_DERIVE_PARAMS;**

8348

8349 The fields of the structure have the following meanings:

8350 **kdf** key derivation function used on the shared secret value
8351 **ulOtherInfoLen** the length in bytes of the other info
8352 **pOtherInfo** some data shared between the two parties
8353 **ulPublicDataLen** the length in bytes of the other party's X9.42 Diffie-Hellman public
8354 key
8355 **pPublicData** pointer to other party's X9.42 Diffie-Hellman public key value

8356 With the key derivation function **CKD_NULL**, *pOtherInfo* must be NULL and *ulOtherInfoLen* must be zero.
8357 With the key derivation function **CKD_SHA1_KDF ASN1**, *pOtherInfo* must be supplied, which contains
8358 an octet string, specified in ASN.1 DER encoding, consisting of mandatory and optional data shared by
8359 the two parties intending to share the shared secret. With the key derivation function
8360 **CKD_SHA1_KDF CONCATENATE**, an optional *pOtherInfo* may be supplied, which consists of some

8361 data shared by the two parties intending to share the shared secret. Otherwise, *pOtherInfo* must be
8362 NULL and *ulOtherInfoLen* must be zero.
8363 **CK_X9_42_DH1_DERIVE_PARAMS_PTR** is a pointer to a **CK_X9_42_DH1_DERIVE_PARAMS**.

8364 • **CK_X9_42_DH2_DERIVE_PARAMS, CK_X9_42_DH2_DERIVE_PARAMS_PTR**

8365 **CK_X9_42_DH2_DERIVE_PARAMS** is a structure that provides the parameters to the
8366 **CKM_X9_42_DH_HYBRID_DERIVE** and **CKM_X9_42_MQV_DERIVE** key derivation mechanisms,
8367 where each party contributes two key pairs. The structure is defined as follows:

```
8368     typedef struct CK_X9_42_DH2_DERIVE_PARAMS {  
8369         CK_X9_42_DH_KDF_TYPE          kdf;  
8370         CK ULONG                  ulOtherInfoLen;  
8371         CK BYTE_PTR                pOtherInfo;  
8372         CK ULONG                  ulPublicDataLen;  
8373         CK BYTE_PTR                pPublicData;  
8374         CK ULONG                  ulPrivateDataLen;  
8375         CK OBJECT_HANDLE           hPrivateKey;  
8376         CK ULONG                  ulPublicDataLen2;  
8377         CK BYTE_PTR                pPublicData2;  
8378     } CK_X9_42_DH2_DERIVE_PARAMS;
```

8379 The fields of the structure have the following meanings:

| | | |
|------|------------------|---|
| 8381 | kdf | key derivation function used on the shared secret value |
| 8382 | ulOtherInfoLen | the length in bytes of the other info |
| 8383 | pOtherInfo | some data shared between the two parties |
| 8384 | ulPublicDataLen | the length in bytes of the other party's first X9.42 Diffie-Hellman public key |
| 8385 | pPublicData | pointer to other party's first X9.42 Diffie-Hellman public key value |
| 8386 | ulPrivateDataLen | the length in bytes of the second X9.42 Diffie-Hellman private key |
| 8387 | hPrivateKey | key handle for second X9.42 Diffie-Hellman private key value |
| 8388 | ulPublicDataLen2 | the length in bytes of the other party's second X9.42 Diffie-Hellman public key |
| 8389 | pPublicData2 | pointer to other party's second X9.42 Diffie-Hellman public key value |
| 8390 | | |

8391 With the key derivation function **CKD_NULL**, *pOtherInfo* must be NULL and *ulOtherInfoLen* must be zero.
8392 With the key derivation function **CKD_SHA1_KDF ASN1**, *pOtherInfo* must be supplied, which contains
8393 an octet string, specified in ASN.1 DER encoding, consisting of mandatory and optional data shared by
8394 the two parties intending to share the shared secret. With the key derivation function
8395 **CKD_SHA1_KDF CONCATENATE**, an optional *pOtherInfo* may be supplied, which consists of some
8396 data shared by the two parties intending to share the shared secret. Otherwise, *pOtherInfo* must be
8397 NULL and *ulOtherInfoLen* must be zero.

8398 **CK_X9_42_DH2_DERIVE_PARAMS_PTR** is a pointer to a **CK_X9_42_DH2_DERIVE_PARAMS**.

8401 • **CK_X9_42_MQV_DERIVE_PARAMS, CK_X9_42_MQV_DERIVE_PARAMS_PTR**

8402 **CK_X9_42_MQV_DERIVE_PARAMS** is a structure that provides the parameters to the
8403 **CKM_X9_42_MQV_DERIVE** key derivation mechanism, where each party contributes two key pairs. The
8404 structure is defined as follows:

```
8405     typedef struct CK_X9_42_MQV_DERIVE_PARAMS {  
8406         CK_X9_42_DH_KDF_TYPE    kdf;  
8407         CK ULONG                ulOtherInfoLen;  
8408         CK BYTE PTR             pOtherInfo;  
8409         CK ULONG                ulPublicDataLen;  
8410         CK BYTE PTR             pPublicData;  
8411         CK ULONG                ulPrivateDataLen;  
8412         CK OBJECT_HANDLE        hPrivateKey;  
8413         CK ULONG                ulPublicDataLen2;  
8414         CK BYTE PTR             pPublicData2;  
8415         CK OBJECT_HANDLE        publicKey;  
8416     } CK_X9_42_MQV_DERIVE_PARAMS;
```

8417 The fields of the structure have the following meanings:

| | | |
|------|------------------|---|
| 8419 | kdf | key derivation function used on the shared secret value |
| 8420 | ulOtherInfoLen | the length in bytes of the other info |
| 8421 | pOtherInfo | some data shared between the two parties |
| 8422 | ulPublicDataLen | the length in bytes of the other party's first X9.42 Diffie-Hellman public key |
| 8423 | pPublicData | pointer to other party's first X9.42 Diffie-Hellman public key value |
| 8425 | ulPrivateDataLen | the length in bytes of the second X9.42 Diffie-Hellman private key |
| 8426 | hPrivateKey | key handle for second X9.42 Diffie-Hellman private key value |
| 8427 | ulPublicDataLen2 | the length in bytes of the other party's second X9.42 Diffie-Hellman public key |
| 8428 | pPublicData2 | pointer to other party's second X9.42 Diffie-Hellman public key value |
| 8429 | publicKey | Handle to the first party's ephemeral public key |

8432 With the key derivation function **CKD_NULL**, *pOtherInfo* must be NULL and *ulOtherInfoLen* must be zero.
8433 With the key derivation function **CKD_SHA1_KDF ASN1**, *pOtherInfo* must be supplied, which contains
8434 an octet string, specified in ASN.1 DER encoding, consisting of mandatory and optional data shared by
8435 the two parties intending to share the shared secret. With the key derivation function
8436 **CKD_SHA1_KDF CONCATENATE**, an optional *pOtherInfo* may be supplied, which consists of some
8437 data shared by the two parties intending to share the shared secret. Otherwise, *pOtherInfo* must be
8438 NULL and *ulOtherInfoLen* must be zero.

8439 **CK_X9_42_MQV_DERIVE_PARAMS_PTR** is a pointer to a **CK_X9_42_MQV_DERIVE_PARAMS**.

8440 **6.4.12 X9.42 Diffie-Hellman key pair generation**

8441 The X9.42 Diffie-Hellman key pair generation mechanism, denoted **CKM_X9_42_DH_KEY_PAIR_GEN**,
8442 is a key pair generation mechanism based on Diffie-Hellman key agreement, as defined in the ANSI
8443 X9.42 standard.

8444 It does not have a parameter.

8445 The mechanism generates X9.42 Diffie-Hellman public/private key pairs with a particular prime, base and
8446 subprime, as specified in the **CKA_PRIME**, **CKA_BASE** and **CKA_SUBPRIME** attributes of the template
8447 for the public key.
8448 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
8449 public key and the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_BASE**, **CKA_SUBPRIME**, and
8450 **CKA_VALUE** attributes to the new private key; other attributes required by the X9.42 Diffie-Hellman
8451 public and private key types must be specified in the templates.
8452 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
8453 specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits, for the **CKA_PRIME** attribute.

8454 **6.4.13 X9.42 Diffie-Hellman domain parameter generation**

8455 The X9.42 Diffie-Hellman domain parameter generation mechanism, denoted
8456 **CKM_X9_42_DH_PARAMETER_GEN**, is a domain parameters generation mechanism based on X9.42
8457 Diffie-Hellman key agreement, as defined in the ANSI X9.42 standard.
8458 It does not have a parameter.
8459 The mechanism generates X9.42 Diffie-Hellman domain parameters with particular prime and subprime
8460 length in bits, as specified in the **CKA_PRIME_BITS** and **CKA_SUBPRIME_BITS** attributes of the
8461 template for the domain parameters.
8462 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_BASE**,
8463 **CKA_SUBPRIME**, **CKA_PRIME_BITS** and **CKA_SUBPRIME_BITS** attributes to the new object. Other
8464 attributes supported by the X9.42 Diffie-Hellman domain parameter types may also be specified in the
8465 template for the domain parameters, or else are assigned default initial values.
8466 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
8467 specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits.

8468 **6.4.14 X9.42 Diffie-Hellman key derivation**

8469 The X9.42 Diffie-Hellman key derivation mechanism, denoted **CKM_X9_42_DH_DERIVE**, is a
8470 mechanism for key derivation based on the Diffie-Hellman key agreement scheme, as defined in the
8471 ANSI X9.42 standard, where each party contributes one key pair, all using the same X9.42 Diffie-Hellman
8472 domain parameters.
8473 It has a parameter, a **CK_X9_42_DH1_DERIVE_PARAMS** structure.
8474 This mechanism derives a secret value, and truncates the result according to the **CKA_KEY_TYPE**
8475 attribute of the template and, if it has one and the key type supports it, the **CKA_VALUE_LEN** attribute of
8476 the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism
8477 contributes the result as the **CKA_VALUE** attribute of the new key; other attributes required by the key
8478 type must be specified in the template. Note that in order to validate this mechanism it may be required to
8479 use the **CKA_VALUE** attribute as the key of a general-length MAC mechanism (e.g.
8480 **CKM_SHA_1_HMAC_GENERAL**) over some test data.

8481 This mechanism has the following rules about key sensitivity and extractability:

- 8482 • The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both
8483 be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some
8484 default value.
- 8485 • If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key
8486 will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the
8487 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
8488 **CKA_SENSITIVE** attribute.
- 8489 • Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
8490 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
8491 CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
8492 value from its **CKA_EXTRACTABLE** attribute.

8493 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
8494 specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits, for the **CKA_PRIME** attribute.

8495 6.4.15 X9.42 Diffie-Hellman hybrid key derivation

8496 The X9.42 Diffie-Hellman hybrid key derivation mechanism, denoted
8497 **CKM_X9_42_DH_HYBRID_DERIVE**, is a mechanism for key derivation based on the Diffie-Hellman
8498 hybrid key agreement scheme, as defined in the ANSI X9.42 standard, where each party contributes two
8499 key pair, all using the same X9.42 Diffie-Hellman domain parameters.

8500 It has a parameter, a **CK_X9_42_DH2_DERIVE_PARAMS** structure.

8501 This mechanism derives a secret value, and truncates the result according to the **CKA_KEY_TYPE**
8502 attribute of the template and, if it has one and the key type supports it, the **CKA_VALUE_LEN** attribute of
8503 the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism
8504 contributes the result as the **CKA_VALUE** attribute of the new key; other attributes required by the key
8505 type must be specified in the template. Note that in order to validate this mechanism it may be required to
8506 use the **CKA_VALUE** attribute as the key of a general-length MAC mechanism (e.g.
8507 **CKM_SHA_1_HMAC_GENERAL**) over some test data.

8508 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.

8516 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
8517 specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits, for the **CKA_PRIME** attribute.

8522 6.4.16 X9.42 Diffie-Hellman Menezes-Qu-Vanstone key derivation

8523 The X9.42 Diffie-Hellman Menezes-Qu-Vanstone (MQV) key derivation mechanism, denoted
8524 **CKM_X9_42_MQV_DERIVE**, is a mechanism for key derivation based the MQV scheme, as defined in
8525 the ANSI X9.42 standard, where each party contributes two key pairs, all using the same X9.42 Diffie-
8526 Hellman domain parameters.

8527 It has a parameter, a **CK_X9_42_MQV_DERIVE_PARAMS** structure.

8528 This mechanism derives a secret value, and truncates the result according to the **CKA_KEY_TYPE**
8529 attribute of the template and, if it has one and the key type supports it, the **CKA_VALUE_LEN** attribute of
8530 the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism
8531 contributes the result as the **CKA_VALUE** attribute of the new key; other attributes required by the key
8532 type must be specified in the template. Note that in order to validate this mechanism it may be required to
8533 use the **CKA_VALUE** attribute as the key of a general-length MAC mechanism (e.g.
8534 **CKM_SHA_1_HMAC_GENERAL**) over some test data.

8535 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

8541 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
8542 **CKA_SENSITIVE** attribute.

- 8543 • Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
8544 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
8545 CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
8546 value from its **CKA_EXTRACTABLE** attribute.

8547 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
8548 specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits, for the **CKA_PRIME** attribute.

8549 **6.5 Extended Triple Diffie-Hellman (x3dh)**

8550 The Extended Triple Diffie-Hellman mechanism described here is the one described in
8551 [SIGNAL].

8552 *Table 91, Extended Triple Diffie-Hellman Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|---------------------|-------------------|---------------|----------------------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_X3DH_INITIALIZE | | | | | | | ✓ |
| CKM_X3DH RESPOND | | | | | | | ✓ |

8554 **6.5.1 Definitions**

8555 Mechanisms:

8556 CKM_X3DH_INITIALIZE

8557 CKM_X3DH_RESPOND

8558 **6.5.2 Extended Triple Diffie-Hellman key objects**

8559 Extended Triple Diffie-Hellman uses Elliptic Curve keys in Montgomery representation
8560 (**CKK_EC_MONTGOMERY**). Three different kinds of keys are used, they differ in their lifespan:

- 8561 • identity keys are long-term keys, which identify the peer,
8562 • prekeys are short-term keys, which should be rotated often (weekly to hourly)
8563 • onetime prekeys are keys, which should be used only once.

8564 Any peer intending to be contacted using X3DH must publish their so-called prekey-bundle, consisting of
8565 their:

- 8566 • public Identity key,
8567 • current prekey, signed using XEDDSA with their identity key
8568 • optionally a batch of One-time public keys.

8569 **6.5.3 Initiating an Extended Triple Diffie-Hellman key exchange**

8570 Initiating an Extended Triple Diffie-Hellman key exchange starts by retrieving the following required public
8571 keys (the so-called prekey-bundle) of the other peer: the Identity key, the signed public Prekey, and
8572 optionally one One-time public key.

8573 When the necessary key material is available, the initiating party calls CKM_X3DH_INITIALIZE, also
8574 providing the following additional parameters:

- 8575 • the initiators identity key

- 8576 • the initiators ephemeral key (a fresh, one-time **CKK_EC_MONTGOMERY** type key)

8577

8578 **CK_X3DH_INITIATE_PARAMS** is a structure that provides the parameters to the
 8579 **CKM_X3DH_INITIALIZE** key exchange mechanism. The structure is defined as follows:

```
8580     typedef struct CK_X3DH_INITIATE_PARAMS {
8581         CK_X3DH_KDF_TYPE    kdf;
8582         CK_OBJECT_HANDLE   pPeer_identity;
8583         CK_OBJECT_HANDLE   pPeer_prekey;
8584         CK_BYTE_PTR        pPrekey_signature;
8585         CK_BYTE_PTR        pOnetime_key;
8586         CK_OBJECT_HANDLE   pOwn_identity;
8587         CK_OBJECT_HANDLE   pOwn_ephemeral;
8588     } CK_X3DH_INITIATE_PARAMS;
```

8589 *Table 92, Extended Triple Diffie-Hellman Initiate Message parameters:*

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

8590

8591 **6.5.4 Responding to an Extended Triple Diffie-Hellman key exchange**

8592 Responding an Extended Triple Diffie-Hellman key exchange is done by executing a
 8593 **CKM_X3DH RESPOND** mechanism. **CK_X3DH RESPOND_PARAMS** is a structure that provides the
 8594 parameters to the **CKM_X3DH RESPOND** key exchange mechanism. All these parameter should be
 8595 supplied by the Initiator in a message to the responder. The structure is defined as follows:

```
8596     typedef struct CK_X3DH_RESPOND_PARAMS {
8597         CK_X3DH_KDF_TYPE    kdf;
8598         CK_BYTE_PTR        pIdentity_id;
8599         CK_BYTE_PTR        pPrekey_id;
8600         CK_BYTE_PTR        pOnetime_id;
8601         CK_OBJECT_HANDLE   pInitiator_identity;
8602         CK_BYTE_PTR        pInitiator_ephemeral;
8603     } CK_X3DH_RESPOND_PARAMS;
```

8604

8605 *Table 93, Extended Triple Diffie-Hellman 1st Message parameters:*

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

8606

8607 Where the *_id fields are identifiers marking which key has been used from the prekey-bundle, these
 8608 identifiers could be the keys themselves.

8609

8610 This mechanism has the following rules about key sensitivity and extractability³:

- 8611 1 The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both
 8612 be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some
 8613 default value.
- 8614 2 If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key
 8615 will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the
 8616 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
 8617 **CKA_SENSITIVE** attribute.
- 8618 3 Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
 8619 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
 8620 CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
 8621 value from its **CKA_EXTRACTABLE** attribute.

8622 6.5.5 Extended Triple Diffie-Hellman parameters

8623 • CK_X3DH_KDF_TYPE, CK_X3DH_KDF_TYPE_PTR

8624 **CK_X3DH_KDF_TYPE** is used to indicate the Key Derivation Function (KDF) applied to derive keying
 8625 data from a shared secret. The key derivation function will be used by the X3DH key agreement
 8626 schemes. It is defined as follows:

```
8627     typedef CK ULONG CK_X3DH_KDF_TYPE;
```

8628

8629 The following table lists the defined functions.

8630 *Table 94, X3DH: Key Derivation Functions*

| Source Identifier |
|---------------------|
| CKD_NULL |
| CKD_BLAKE2B_256_KDF |
| CKD_BLAKE2B_512_KDF |
| CKD_SHA3_256_KDF |

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

| |
|------------------|
| CKD_SHA256_KDF |
| CKD_SHA3_512_KDF |
| CKD_SHA512_KDF |

8631 6.6 Double Ratchet

8632 The Double Ratchet is a key management algorithm managing the ongoing renewal and maintenance of
 8633 short-lived session keys providing forward secrecy and break-in recovery for encrypt/decrypt operations.
 8634 The algorithm is described in [DoubleRatchet]. The Signal protocol uses X3DH to exchange a shared
 8635 secret in the first step, which is then used to derive a Double Ratchet secret key.

8636 *Table 95, Double Ratchet Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|--------------------------|-------------------|---------------|----------------------|--------|-------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/Key Pair | Wrap & Unwrap | Derive |
| CKM_X2RATCHET_INITIALIZE | | | | | | | ✓ |
| CKM_X2RATCHET RESPOND | | | | | | | ✓ |
| CKM_X2RATCHET_ENCRYPT | ✓ | | | | | ✓ | |
| CKM_X2RATCHET_DECRYPT | ✓ | | | | | ✓ | |

8637

8638 6.6.1 Definitions

8639 This section defines the key type "CKK_X2RATCHET" for type CK_KEY_TYPE as used in the
 8640 CKA_KEY_TYPE attribute of key objects.

8641 Mechanisms:

8642 CKM_X2RATCHET_INITIALIZE
 8643 CKM_X2RATCHET_RESPOND
 8644 CKM_X2RATCHET_ENCRYPT
 8645 CKM_X2RATCHET_DECRYPT

8646 6.6.2 Double Ratchet secret key objects

8647 Double Ratchet secret key objects (object class CKO_SECRET_KEY, key type CKK_X2RATCHET) hold
 8648 Double Ratchet keys. Double Ratchet secret keys can only be derived from shared secret keys using the
 8649 mechanism CKM_X2RATCHET_INITIALIZE or CKM_X2RATCHET_RESPOND. In the Signal protocol
 8650 these are seeded with the shared secret derived from an Extended Triple Diffie-Hellman [X3DH] key-
 8651 exchange. The following table defines the Double Ratchet secret key object attributes, in addition to the
 8652 common attributes defined for this object class:

8653 *Table 96, Double Ratchet Secret Key Object Attributes*

| Attribute | Data type | Meaning |
|--------------------|------------|--------------------------|
| CKA_X2RATCHET_RK | Byte array | Root key |
| CKA_X2RATCHET_HKS | Byte array | Sender Header key |
| CKA_X2RATCHET_HKR | Byte array | Receiver Header key |
| CKA_X2RATCHET_NHKS | Byte array | Next Sender Header Key |
| CKA_X2RATCHET_NHKR | Byte array | Next Receiver Header Key |
| CKA_X2RATCHET_CKS | Byte array | Sender Chain key |

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

8654 6.6.3 Double Ratchet key derivation

8655 The Double Ratchet key derivation mechanisms depend on who is the initiating party, and who the
 8656 receiving, denoted **CKM_X2RATCHET_INITIALIZE** and **CKM_X2RATCHET RESPOND**, are the key
 8657 derivation mechanisms for the Double Ratchet. Usually the keys are derived from a shared secret by
 8658 executing a X3DH key exchange.

8659 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
 8660 key. Additionally the attribute flags indicating which functions the key supports are also contributed by the
 8661 mechanism.

8662 For this mechanism, the only allowed values are 255 and 448 as RFC 8032 only defines curves of these
 8663 two sizes. A Cryptoki implementation may support one or both of these curves and should set the
 8664 *ulMinKeySize* and *ulMaxKeySize* fields accordingly.

- 8665 • **CK_X2RATCHET_INITIALIZE_PARAMS;**
 8666 **CK_X2RATCHET_INITIALIZE_PARAMS_PTR**

8667 **CK_X2RATCHET_INITIALIZE_PARAMS** provides the parameters to the
 8668 **CKM_X2RATCHET_INITIALIZE** mechanism. It is defined as follows:

```
8669     typedef struct CK_X2RATCHET_INITIALIZE_PARAMS {
8670         CK_BYTE_PTR                      sk;
8671         CK_OBJECT_HANDLE                  peer_public_prekey;
8672         CK_OBJECT_HANDLE                  peer_public_identity;
8673         CK_OBJECT_HANDLE                  own_public_identity;
8674         CK_BBOOL                          bEncryptedHeader;
8675         CK ULONG                          eCurve;
8676         CK_MECHANISM_TYPE                aeadMechanism;
8677         CK_X2RATCHET_KDF_TYPE            kdfMechanism;
8678     } CK_X2RATCHET_INITIALIZE_PARAMS;
```

8679 The fields of the structure have the following meanings:

- | | |
|-----------------------------------|---|
| 8680 <i>sk</i> | <i>the shared secret with peer (derived using X3DH)</i> |
| 8682 <i>peers_public_prekey</i> | <i>Peers public prekey which the Initiator used in the X3DH</i> |
| 8683 <i>peers_public_identity</i> | <i>Peers public identity which the Initiator used in the X3DH</i> |

8684 *own_public_identity* *Initiators public identity as used in the X3DH*
 8685 *bEncryptedHeader* *whether the headers are encrypted*
 8686 *eCurve* *255 for curve 25519 or 448 for curve 448*
 8687 *aeadMechanism* *a mechanism supporting AEAD encryption*
 8688 *kdfMechanism* *a Key Derivation Mechanism, such as*
 8689 *CKD_BLAKE2B_512_KDF*

- 8690 • **CK_X2RATCHET_RESPOND_PARAMS;**
 8691 **CK_X2RATCHET_RESPOND_PARAMS_PTR**

8692 **CK_X2RATCHET_RESPOND_PARAMS** provides the parameters to the
 8693 **CKM_X2RATCHET_RESPOND** mechanism. It is defined as follows:

```

8694     typedef struct CK_X2RATCHET_RESPOND_PARAMS {
8695         CK_BYTE_PTR           sk;
8696         CK_OBJECT_HANDLE      own_prekey;
8697         CK_OBJECT_HANDLE      initiator_identity;
8698         CK_OBJECT_HANDLE      own_public_identity;
8699         CK_BBOOL               bEncryptedHeader;
8700         CK ULONG              eCurve;
8701         CK_MECHANISM_TYPE    aeadMechanism;
8702         CK_X2RATCHET_KDF_TYPE kdfMechanism;
8703     } CK_X2RATCHET_RESPOND_PARAMS;
  
```

8704 The fields of the structure have the following meanings:

| | |
|--|---|
| 8705 <i>sk</i> 8706 <i>own_prekey</i> 8707 <i>initiator_identity</i> 8708 <i>own_public_identity</i> 8709 <i>bEncryptedHeader</i> 8710 <i>eCurve</i> 8711 <i>aeadMechanism</i> 8712 <i>kdfMechanism</i> | <i>shared secret with the Initiator</i> <i>Own Prekey pair that the Initiator used</i> <i>Initiators public identity key used</i> <i>as used in the prekey bundle by the initiator in the X3DH</i> <i>whether the headers are encrypted</i> <i>255 for curve 25519 or 448 for curve 448</i> <i>a mechanism supporting AEAD encryption</i> <i>a Key Derivation Mechanism, such as</i> <i>CKD_BLAKE2B_512_KDF</i> |
|--|---|

8715 **6.6.4 Double Ratchet Encryption mechanism**

8716 The Double Ratchet encryption mechanism, denoted **CKM_X2RATCHET_ENCRYPT** and
 8717 **CKM_X2RATCHET_DECRYPT**, are a mechanisms for single part encryption and decryption based on
 8718 the Double Ratchet and its underlying AEAD cipher.

8719 **6.6.5 Double Ratchet parameters**

- 8720 • **CK_X2RATCHET_KDF_TYPE, CK_X2RATCHET_KDF_TYPE_PTR**

8721 **CK_X2RATCHET_KDF_TYPE** is used to indicate the Key Derivation Function (KDF) applied to derive
8722 keying data from a shared secret. The key derivation function will be used by the X key derivation
8723 scheme. It is defined as follows:

8724 **typedef CK ULONG CK_X2RATCHET_KDF_TYPE;**

8725

8726 The following table lists the defined functions.

8727 *Table 97, X2RATCHET: Key Derivation Functions*

| Source Identifier |
|---------------------|
| CKD_NULL |
| CKD_BLAKE2B_256_KDF |
| CKD_BLAKE2B_512_KDF |
| CKD_SHA3_256_KDF |
| CKD_SHA256_KDF |
| CKD_SHA3_512_KDF |
| CKD_SHA512_KDF |

8728

8729 **6.7 Wrapping/unwrapping private keys**

8730 Cryptoki Versions 2.01 and up allow the use of secret keys for wrapping and unwrapping RSA private
8731 keys, Diffie-Hellman private keys, X9.42 Diffie-Hellman private keys, short Weierstrass EC private keys
8732 and DSA private keys. For wrapping, a private key is BER-encoded according to [PKCS #8]
8733 PrivateKeyInfo ASN.1 type. PKCS #8 requires an algorithm identifier for the type of the private key. The
8734 object identifiers for the required algorithm identifiers are as follows:

```
8735     rsaEncryption OBJECT IDENTIFIER ::= { pkcs-1 1 }
8736
8737     dhKeyAgreement OBJECT IDENTIFIER ::= { pkcs-3 1 }
8738
8739     dhpublicnumber OBJECT IDENTIFIER ::= { iso(1) member-body(2)
8740                         us(840) ansi-x942(10046) number-type(2) 1 }
8741
8742     id-ecPublicKey OBJECT IDENTIFIER ::= { iso(1) member-body(2)
8743                         us(840) ansi-x9-62(10045) publicKeyType(2) 1 }
8744
8745     id-dsa OBJECT IDENTIFIER ::= {
8746         iso(1) member-body(2) us(840) x9-57(10040) x9cm(4) 1 }
8747
8748     where
8749     pkcs-1 OBJECT IDENTIFIER ::= {
8750         iso(1) member-body(2) US(840) rsadsi(113549) pkcs(1) 1 }
8751
8752     pkcs-3 OBJECT IDENTIFIER ::= {
8753         iso(1) member-body(2) US(840) rsadsi(113549) pkcs(1) 3 }
```

```

8755      These parameters for the algorithm identifiers have the
8756          following types, respectively:
8757          NULL
8758
8759          DHParameter ::= SEQUENCE {
8760              prime                  INTEGER,    -- p
8761              base                   INTEGER,    -- g
8762              privateValueLength     INTEGER OPTIONAL
8763          }
8764
8765          DomainParameters ::= SEQUENCE {
8766              prime                  INTEGER,    -- p
8767              base                   INTEGER,    -- g
8768              subprime                INTEGER,    -- q
8769              cofactor                INTEGER OPTIONAL, -- j
8770              validationParms         ValidationParms OPTIONAL
8771          }
8772
8773          ValidationParms ::= SEQUENCE {
8774              Seed                   BIT STRING, -- seed
8775              PGenCounter            INTEGER      -- parameter verification
8776          }
8777
8778          Parameters ::= CHOICE {
8779              ecParameters           ECParameters,
8780              namedCurve             CURVES.&id({CurveNames}),
8781              implicitlyCA          NULL
8782          }
8783
8784          Dss_Parms ::= SEQUENCE {
8785              p INTEGER,
8786              q INTEGER,
8787              g INTEGER
8788          }
8789
8790      For the X9.42 Diffie-Hellman domain parameters, the cofactor and the validationParms optional fields
8791      should not be used when wrapping or unwrapping X9.42 Diffie-Hellman private keys since their values
8792      are not stored within the token.
8793      For the EC domain parameters, the use of namedCurve is recommended over the choice
8794      ecParameters. The choice implicitlyCA must not be used in Cryptoki.
8795      Within the PrivateKeyInfo type:
8796      • RSA private keys are BER-encoded according to PKCS #1's RSAPrivateKey ASN.1 type. This type
8797          requires values to be present for all the attributes specific to Cryptoki's RSA private key objects. In
8798          other words, if a Cryptoki library does not have values for an RSA private key's CKA_MODULUS,
8799          CKA_PUBLIC_EXPONENT, CKA_PRIVATE_EXPONENT, CKA_PRIME_1, CKA_PRIME_2,
8800          CKA_EXPONENT_1, CKA_EXPONENT_2, and CKA_COEFFICIENT values, it must not create an
8801          RSAPrivateKey BER-encoding of the key, and so it must not prepare it for wrapping.
8802      • Diffie-Hellman private keys are represented as BER-encoded ASN.1 type INTEGER.

```

- 8803 • X9.42 Diffie-Hellman private keys are represented as BER-encoded ASN.1 type INTEGER.
 8804 • Short Weierstrass EC private keys are BER-encoded according to SEC G SEC 1 ECPrivateKey
 8805 ASN.1 type:

```
8806   ECPrivateKey ::= SEQUENCE {
8807     Version      INTEGER { ecPrivkeyVer1(1) }
8808     (ecPrivkeyVer1),
8809     privateKey   OCTET STRING,
8810     parameters   [0] Parameters OPTIONAL,
8811     publicKey    [1] BIT STRING OPTIONAL
8812   }
```

8813
 8814 Since the EC domain parameters are placed in the PKCS #8's privateKeyAlgorithm field, the optional
 8815 **parameters** field in an ECPrivateKey must be omitted. A Cryptoki application must be able to
 8816 unwrap an ECPrivateKey that contains the optional **publicKey** field; however, what is done with this
 8817 **publicKey** field is outside the scope of Cryptoki.

- 8818 • DSA private keys are represented as BER-encoded ASN.1 type INTEGER.

8819 Once a private key has been BER-encoded as a PrivateKeyInfo type, the resulting string of bytes is
 8820 encrypted with the secret key. This encryption is defined in the section for the respective key wrapping
 8821 mechanism.

8822 Unwrapping a wrapped private key undoes the above procedure. The ciphertext is decrypted as defined
 8823 for the respective key unwrapping mechanism. The data thereby obtained are parsed as a
 8824 PrivateKeyInfo type. An error will result if the original wrapped key does not decrypt properly, or if the
 8825 decrypted data does not parse properly, or its type does not match the key type specified in the template
 8826 for the new key. The unwrapping mechanism contributes only those attributes specified in the
 8827 PrivateKeyInfo type to the newly-unwrapped key; other attributes must be specified in the template, or will
 8828 take their default values.

8829 Earlier drafts of PKCS #11 Version 2.0 and Version 2.01 used the object identifier

```
8830   DSA OBJECT IDENTIFIER ::= { algorithm 12 }
8831   algorithm OBJECT IDENTIFIER ::= {
8832     iso(1) identifier-organization(3) oiw(14) secsig(3)
8833       algorithm(2) }
```

8835 with associated parameters

```
8836   DSAParameters ::= SEQUENCE {
8837     prime1 INTEGER, -- modulus p
8838     prime2 INTEGER, -- modulus q
8839     base INTEGER -- base g
8840   }
```

8841 for wrapping DSA private keys. Note that although the two structures for holding DSA domain
 8842 parameters appear identical when instances of them are encoded, the two corresponding object
 8843 identifiers are different.

8845 6.8 Generic secret key

8846 *Table 98, Generic Secret Key Mechanisms vs. Functions*

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

6.8.1 Definitions

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

Mechanisms:

CKM_GENERIC_SECRET_KEY_GEN

6.8.2 Generic secret key objects

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

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

Table 99, Generic Secret Key Object Attributes

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

Refer to Table 11 for footnotes

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

```

CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_GENERIC_SECRET;
CK_UTF8CHAR label[] = "A generic secret key object";
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_DERIVE, &true, sizeof(true)},
    {CKA_VALUE, value, sizeof(value)}}
};
```

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

8877 6.8.3 Generic secret key generation

8878 The generic secret key generation mechanism, denoted **CKM_GENERIC_SECRET_KEY_GEN**, is used
8879 to generate generic secret keys. The generated keys take on any attributes provided in the template
8880 passed to the **C_GenerateKey** call, and the **CKA_VALUE_LEN** attribute specifies the length of the key
8881 to be generated.

8882 It does not have a parameter.

8883 The template supplied must specify a value for the **CKA_VALUE_LEN** attribute. If the template specifies
8884 an object type and a class, they must have the following values:

```
8885     CK_OBJECT_CLASS = CKO_SECRET_KEY;  
8886     CK_KEY_TYPE = CKK_GENERIC_SECRET;
```

8887 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
8888 specify the supported range of key sizes, in bits.

8889 6.9 HMAC mechanisms

8890 Refer to **RFC2104** and **FIPS 198** for HMAC algorithm description. The HMAC secret key shall correspond
8891 to the PKCS11 generic secret key type or the mechanism specific key types (see mechanism definition).
8892 Such keys, for use with HMAC operations can be created using **C_CreateObject** or **C_GenerateKey**.

8893 The RFC also specifies test vectors for the various hash function based HMAC mechanisms described in
8894 the respective hash mechanism descriptions. The RFC should be consulted to obtain these test vectors.

8895 6.9.1 General block cipher mechanism parameters

8896 • **CK_MAC_GENERAL_PARAMS; CK_MAC_GENERAL_PARAMS_PTR**

8897 **CK_MAC_GENERAL_PARAMS** provides the parameters to the general-length MACing mechanisms of
8898 the DES, DES3 (triple-DES), AES, Camellia, SEED, and ARIA ciphers. It also provides the parameters to
8899 the general-length HMACing mechanisms (i.e., SHA-1, SHA-256, SHA-384, SHA-512, and SHA-512/T
8900 family) and the two SSL 3.0 MACing mechanisms, (i.e., MD5 and SHA-1). It holds the length of the MAC
8901 that these mechanisms produce. It is defined as follows:

```
8902     typedef CK ULONG CK_MAC_GENERAL_PARAMS;
```

8904 **CK_MAC_GENERAL_PARAMS_PTR** is a pointer to a **CK_MAC_GENERAL_PARAMS**.

8905 6.10 AES

8906 For the Advanced Encryption Standard (AES) see [FIPS PUB 197].

8907 *Table 100, AES Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|---------------------|-------------------|---------------|----------------------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_AES_KEY_GEN | | | | | ✓ | | |
| CKM_AES_ECB | ✓ | | | | | ✓ | |
| CKM_AES_CBC | ✓ | | | | | ✓ | |
| CKM_AES_CBC_PAD | ✓ | | | | | ✓ | |
| CKM_AES_MAC_GENERAL | | ✓ | | | | | |

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

8908 6.10.1 Definitions

8909 This section defines the key type “CKK_AES” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE
 8910 attribute of key objects.

8911 Mechanisms:

8912 CKM_AES_KEY_GEN
 8913 CKM_AES_ECB
 8914 CKM_AES_CBC
 8915 CKM_AES_MAC
 8916 CKM_AES_MAC_GENERAL
 8917 CKM_AES_CBC_PAD
 8918 CKM_AES_OFB
 8919 CKM_AES_CFB64
 8920 CKM_AES_CFB8
 8921 CKM_AES_CFB128
 8922 CKM_AES_CFB1
 8923 CKM_AES_XCBC_MAC
 8924 CKM_AES_XCBC_MAC_96

8925 6.10.2 AES secret key objects

8926 AES secret key objects (object class **CKO_SECRET_KEY**, key type **CKK_AES**) hold AES keys. The
 8927 following table defines the AES secret key object attributes, in addition to the common attributes defined
 8928 for this object class:

8929 *Table 101, AES Secret Key Object Attributes*

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

8930 ¹Refer to Table 11 for footnotes

8931 The following is a sample template for creating an AES secret key object:

```

8932     CK_OBJECT_CLASS class = CKO_SECRET_KEY;
8933     CK_KEY_TYPE keyType = CKK_AES;
8934     CK_UTF8CHAR label[] = "An AES secret key object";
8935     CK_BYTE value[] = { ... };
8936     CK_BBOOL true = CK_TRUE;
8937     CK_ATTRIBUTE template[] = {
8938         {CKA_CLASS, &class, sizeof(class)},
8939         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
8940         {CKA_TOKEN, &true, sizeof(true)},
8941         {CKA_LABEL, label, sizeof(label)-1},
8942         {CKA_ENCRYPT, &true, sizeof(true)},
8943         {CKA_VALUE, value, sizeof(value)}}
8944     } ;
8945
8946 CKA_CHECK_VALUE: The value of this attribute is derived from the key object by taking the first three
8947 bytes of the ECB encryption of a single block of null (0x00) bytes, using the default cipher associated with
8948 the key type of the secret key object.

```

8949 6.10.3 AES key generation

8950 The AES key generation mechanism, denoted **CKM_AES_KEY_GEN**, is a key generation mechanism for
8951 NIST's Advanced Encryption Standard.

8952 It does not have a parameter.

8953 The mechanism generates AES keys with a particular length in bytes, as specified in the
8954 **CKA_VALUE_LEN** attribute of the template for the key.

8955 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
8956 key. Other attributes supported by the AES key type (specifically, the flags indicating which functions the
8957 key supports) may be specified in the template for the key, or else are assigned default initial values.

8958 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
8959 specify the supported range of AES key sizes, in bytes.

8960 6.10.4 AES-ECB

8961 AES-ECB, denoted **CKM_AES_ECB**, is a mechanism for single- and multiple-part encryption and
8962 decryption; key wrapping; and key unwrapping, based on NIST Advanced Encryption Standard and
8963 electronic codebook mode.

8964 It does not have a parameter.

8965 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to
8966 wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the
8967 **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus
8968 one null bytes so that the resulting length is a multiple of the block size. The output data is the same
8969 length as the padded input data. It does not wrap the key type, key length, or any other information about
8970 the key; the application must convey these separately.

8971 For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the
8972 **CKA_KEY_TYPE** attribute of the template and, if it has one, and the key type supports it, the
8973 **CKA_VALUE_LEN** attribute of the template. The mechanism contributes the result as the **CKA_VALUE**
8974 attribute of the new key; other attributes required by the key type must be specified in the template.

8975 Constraints on key types and the length of data are summarized in the following table:

8976 *Table 102, AES-ECB: Key And Data Length*

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

8977 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 8978 specify the supported range of AES key sizes, in bytes.

8979 6.10.5 AES-CBC

8980 AES-CBC, denoted **CKM_AES_CBC**, is a mechanism for single- and multiple-part encryption and
 8981 decryption; key wrapping; and key unwrapping, based on NIST's Advanced Encryption Standard and
 8982 cipher-block chaining mode.

8983 It has a parameter, a 16-byte initialization vector.

8984 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to
 8985 wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the
 8986 **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus
 8987 one null bytes so that the resulting length is a multiple of the block size. The output data is the same
 8988 length as the padded input data. It does not wrap the key type, key length, or any other information about
 8989 the key; the application must convey these separately.

8990 For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the
 8991 **CKA_KEY_TYPE** attribute of the template and, if it has one, and the key type supports it, the
 8992 **CKA_VALUE_LEN** attribute of the template. The mechanism contributes the result as the **CKA_VALUE**
 8993 attribute of the new key; other attributes required by the key type must be specified in the template.

8994 Constraints on key types and the length of data are summarized in the following table:

8995 *Table 103, AES-CBC: Key And Data Length*

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

8996 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 8997 specify the supported range of AES key sizes, in bytes.

8998 6.10.6 AES-CBC with PKCS padding

8999 AES-CBC with PKCS padding, denoted **CKM_AES_CBC_PAD**, is a mechanism for single- and multiple-
 9000 part encryption and decryption; key wrapping; and key unwrapping, based on NIST's Advanced

9001 Encryption Standard; cipher-block chaining mode; and the block cipher padding method detailed in
 9002 [PKCS #7].
 9003 It has a parameter, a 16-byte initialization vector.
 9004 The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the
 9005 ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified
 9006 for the **CKA_VALUE_LEN** attribute.
 9007 In addition to being able to wrap and unwrap secret keys, this mechanism can wrap and unwrap RSA,
 9008 Diffie-Hellman, X9.42 Diffie-Hellman, short Weierstrass EC and DSA private keys (see Section 6.7 for
 9009 details). The entries in the table below for data length constraints when wrapping and unwrapping keys
 9010 do not apply to wrapping and unwrapping private keys.
 9011 Constraints on key types and the length of data are summarized in the following table:
 9012

Table 104, AES-CBC with PKCS Padding: Key And Data Length

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

9013 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 9014 specify the supported range of AES key sizes, in bytes.

6.10.7 AES-OFB

9016 AES-OFB, denoted CKM_AES_OFB. It is a mechanism for single and multiple-part encryption and
 9017 decryption with AES. AES-OFB mode is described in [NIST sp800-38a].
 9018 It has a parameter, an initialization vector for this mode. The initialization vector has the same length as
 9019 the block size.
 9020
 9021 Constraints on key types and the length of data are summarized in the following table:
 9022
 9023

Table 105, AES-OFB: Key And Data Length

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

9024 For this mechanism the CK_MECHANISM_INFO structure is as specified for CBC mode.

6.10.8 AES-CFB

9026 Cipher AES has a cipher feedback mode, AES-CFB, denoted CKM_AES_CFB8, CKM_AES_CFB64, and
 9027 CKM_AES_CFB128. It is a mechanism for single and multiple-part encryption and decryption with AES.
 9028 AES-OFB mode is described [NIST sp800-38a].
 9029 It has a parameter, an initialization vector for this mode. The initialization vector has the same length as
 9030 the block size.
 9031

9032 Constraints on key types and the length of data are summarized in the following table:

9033

9034 *Table 106, AES-CFB: Key And Data Length*

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

9035 For this mechanism the CK_MECHANISM_INFO structure is as specified for CBC mode.

9036 6.10.9 General-length AES-MAC

9037 General-length AES-MAC, denoted **CKM_AES_MAC_GENERAL**, is a mechanism for single- and
9038 multiple-part signatures and verification, based on NIST Advanced Encryption Standard as defined in
9039 FIPS PUB 197 and data authentication as defined in FIPS PUB 113.

9040 It has a parameter, a **CK_MAC_GENERAL_PARAMS** structure, which specifies the output length
9041 desired from the mechanism.

9042 The output bytes from this mechanism are taken from the start of the final AES cipher block produced in
9043 the MACing process.

9044 Constraints on key types and the length of data are summarized in the following table:

9045 *Table 107, General-length AES-MAC: Key And Data Length*

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

9046 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
9047 specify the supported range of AES key sizes, in bytes.

9048 6.10.10 AES-MAC

9049 AES-MAC, denoted by **CKM_AES_MAC**, is a special case of the general-length AES-MAC mechanism.
9050 AES-MAC always produces and verifies MACs that are half the block size in length.

9051 It does not have a parameter.

9052 Constraints on key types and the length of data are summarized in the following table:

9053 *Table 108, AES-MAC: Key And Data Length*

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

9054 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
9055 specify the supported range of AES key sizes, in bytes.

9056 6.10.11 AES-XCBC-MAC

9057 AES-XCBC-MAC, denoted **CKM_AES_XCBC_MAC**, is a mechanism for single and multiple part
9058 signatures and verification; based on NIST's Advanced Encryption Standard and [RFC 3566].

9059 It does not have a parameter.

9060 Constraints on key types and the length of data are summarized in the following table:

9061 *Table 109, AES-XCBC-MAC: Key And Data Length*

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

9062 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 9063 specify the supported range of AES key sizes, in bytes.

9064 **6.10.12 AES-XCBC-MAC-96**

9065 AES-XCBC-MAC-96, denoted **CKM_AES_XCBC_MAC_96**, is a mechanism for single and multiple part
 9066 signatures and verification; based on NIST's Advanced Encryption Standard and [RFC 3566].

9067 It does not have a parameter.

9068 Constraints on key types and the length of data are summarized in the following table:

9069 *Table 110, AES-XCBC-MAC: Key And Data Length*

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

9070 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 9071 specify the supported range of AES key sizes, in bytes.

9072 **6.11 AES with Counter**

9073 *Table 111, AES with Counter Mechanisms vs. Functions*

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

9074 **6.11.1 Definitions**

9075 Mechanisms:

9076 CKM_AES_CTR

9077 **6.11.2 AES with Counter mechanism parameters**

9078 ♦ **CK_AES_CTR_PARAMS; CK_AES_CTR_PARAMS_PTR**

9079 **CK_AES_CTR_PARAMS** is a structure that provides the parameters to the **CKM_AES_CTR** mechanism.
 9080 It is defined as follows:

```
9081     typedef struct CK_AES_CTR_PARAMS {
9082         CK ULONG ulCounterBits;
9083         CK BYTE cb[16];
9084     } CK_AES_CTR_PARAMS;
```

9086 ulCounterBits specifies the number of bits in the counter block (cb) that shall be incremented. This
 9087 number shall be such that $0 < \text{ulCounterBits} \leq 128$. For any values outside this range the mechanism
 9088 shall return **CKR_MECHANISM_PARAM_INVALID**.

9089 It's up to the caller to initialize all of the bits in the counter block including the counter
9090 bits are the least significant bits of the counter block (cb). They are a big-endian value usually starting
9091 with 1. The rest of 'cb' is for the nonce, and maybe an optional IV.

9092 E.g. as defined in [RFC 3686]:

```
9093    0          1          2          3
9094    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
9095 +---+---+---+---+---+---+---+---+---+---+---+---+---+
9096 |           Nonce           |
9097 +---+---+---+---+---+---+---+---+---+---+---+---+---+
9098 |           Initialization Vector (IV)           |
9099 |           |
9100 +---+---+---+---+---+---+---+---+---+---+---+---+---+
9101 |           Block Counter           |
9102 +---+---+---+---+---+---+---+---+---+---+---+---+---+
```

9103
9104 This construction permits each packet to consist of up to $2^{32}-1$ blocks = 4,294,967,295 blocks =
9105 68,719,476,720 octets.

9106 **CK_AES_CTR_PARAMS_PTR** is a pointer to a **CK_AES_CTR_PARAMS**.

9107 6.11.3 AES with Counter Encryption / Decryption

9108 Generic AES counter mode is described in NIST Special Publication 800-38A and in RFC 3686. These
9109 describe encryption using a counter block which may include a nonce to guarantee uniqueness of the
9110 counter block. Since the nonce is not incremented, the mechanism parameter must specify the number of
9111 counter bits in the counter block.

9112 The block counter is incremented by 1 after each block of plaintext is processed. There is no support for
9113 any other increment functions in this mechanism.

9114 If an attempt to encrypt/decrypt is made which will cause an overflow of the counter block's counter bits,
9115 then the mechanism shall return **CKR_DATA_LEN_RANGE**. Note that the mechanism should allow the
9116 final post increment of the counter to overflow (if it implements it this way) but not allow any further
9117 processing after this point. E.g. if ulCounterBits = 2 and the counter bits start as 1 then only 3 blocks of
9118 data can be processed.

9119

9120 6.12 AES CBC with Cipher Text Stealing CTS

9121 Ref [NIST AES CTS]

9122 This mode allows unpadded data that has length that is not a multiple of the block size to be encrypted to
9123 the same length of cipher text.

9124 *Table 112, AES CBC with Cipher Text Stealing CTS Mechanisms vs. Functions*

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

9125 6.12.1 Definitions

9126 Mechanisms:

9127 **CKM_AES_CTS**

9128 **6.12.2 AES CTS mechanism parameters**

9129 It has a parameter, a 16-byte initialization vector.

9130 *Table 113, AES-CTS: Key And Data Length*

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

9131

9132 **6.13 Additional AES Mechanisms**

9133 *Table 114, Additional AES Mechanisms vs. Functions*

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

9134

9135 **6.13.1 Definitions**

9136 Mechanisms:

9137 CKM_AES_GCM

9138 CKM_AES_CCM

9139 CKM_AES_GMAC

9140 Generator Functions:

9141 CKG_NO_GENERATE

9142 CKG_GENERATE

9143 CKG_GENERATE_COUNTER

9144 CKG_GENERATE_RANDOM

9145 CKG_GENERATE_COUNTER_XOR

9146 **6.13.2 AES-GCM Authenticated Encryption / Decryption**

9147 Generic GCM mode is described in [GCM]. To set up for AES-GCM use the following process, where K (key) and AAD (additional authenticated data) are as described in [GCM]. AES-GCM uses CK_GCM_PARAMS for Encrypt, Decrypt and CK_GCM_MESSAGE_PARAMS for MessageEncrypt and MessageDecrypt.

9151 Encrypt:

- 9152 • Set the IV length $ullvLen$ in the parameter block.

- 9153 • Set the IV data pIV in the parameter block.

- 9154 • Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD* may be NULL if
9155 *ulAADLen* is 0.

- 9156 • Set the tag length *ulTagBits* in the parameter block.
9157 • Call C_EncryptInit() for **CKM_AES_GCM** mechanism with parameters and key *K*.
9158 • Call C_Encrypt(), or C_EncryptUpdate()⁴ C_EncryptFinal(), for the plaintext obtaining ciphertext
9159 and authentication tag output.

9160 Decrypt:

- 9161 • Set the IV length *ullvLen* in the parameter block.
9162 • Set the IV data *p/v* in the parameter block.
9163 • Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD* may be NULL if
9164 *ulAADLen* is 0.
9165 • Set the tag length *ulTagBits* in the parameter block.
9166 • Call C_DecryptInit() for **CKM_AES_GCM** mechanism with parameters and key *K*.
9167 • Call C_Decrypt(), or C_DecryptUpdate()⁵ C_DecryptFinal(), for the ciphertext, including the
9168 appended tag, obtaining plaintext output. Note: since **CKM_AES_GCM** is an AEAD cipher, no data
9169 should be returned until C_Decrypt() or C_DecryptFinal().

9170 MessageEncrypt:

- 9171 • Set the IV length *ullvLen* in the parameter block.
9172 • Set *p/v* to hold the IV data returned from C_EncryptMessage() and C_EncryptMessageBegin(). If
9173 *ullvFixedBits* is not zero, then the most significant bits of *p/V* contain the fixed IV. If *ivGenerator* is
9174 set to CKG_NO_GENERATE, *p/v* is an input parameter with the full IV.
9175 • Set the *ullvFixedBits* and *ivGenerator* fields in the parameter block.
9176 • Set the tag length *ulTagBits* in the parameter block.
9177 • Set *pTag* to hold the tag data returned from C_EncryptMessage() or the final
9178 C_EncryptMessageNext().
9179 • Call C_MessageEncryptInit() for **CKM_AES_GCM** mechanism key *K*.
9180 • Call C_EncryptMessage(), or C_EncryptMessageBegin() followed by C_EncryptMessageNext()⁵.
9181 The mechanism parameter is passed to all three of these functions.
9182 • Call C_MessageEncryptFinal() to close the message decryption.

9183 MessageDecrypt:

- 9184 • Set the IV length *ullvLen* in the parameter block.
9185 • Set the IV data *p/v* in the parameter block.
9186 • The *ullvFixedBits* and *ivGenerator* fields are ignored.
9187 • Set the tag length *ulTagBits* in the parameter block.
9188 • Set the tag data *pTag* in the parameter block before C_DecryptMessage() or the final
9189 C_DecryptMessageNext().

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

5 "*" indicates 0 or more calls may be made as required

- 9190 • Call C_MessageDecryptInit() for **CKM_AES_GCM** mechanism key K .
 9191 • Call C_DecryptMessage(), or C_DecryptMessageBegin followed by C_DecryptMessageNext()⁶.
 9192 The mechanism parameter is passed to all three of these functions.
 9193 • Call C_MessageDecryptFinal() to close the message decryption.

9194 In pIV the least significant bit of the initialization vector is the rightmost bit. $ullvLen$ is the length of the
 9195 initialization vector in bytes.

9196 On MessageEncrypt, the meaning of $ivGenerator$ is as follows: CKG_NO_GENERATE means the IV is
 9197 passed in on MessageEncrypt and no internal IV generation is done. CKG_GENERATE means that the
 9198 non-fixed portion of the IV is generated by the module internally. The generation method is not defined.

9199 CKG_GENERATE_COUNTER means that the non-fixed portion of the IV is generated by the module
 9200 internally by use of an incrementing counter, the initial IV counter is zero.

9201 CKG_GENERATE_COUNTER_XOR means that the non-fixed portion of the IV is xored with a counter.
 9202 The value of the non-fixed portion passed must not vary from call to call. Like
 9203 CKG_GENERATE_COUNTER, the counter starts at zero.

9204 CKG_GENERATE_RANDOM means that the non-fixed portion of the IV is generated by the module
 9205 internally using a PRNG. In any case the entire IV, including the fixed portion, is returned in pIV .

9206 Modules must implement CKG_GENERATE. Modules may also reject $ullvFixedBits$ values which are too
 9207 large. Zero is always an acceptable value for $ullvFixedBits$.

9208 In Encrypt and Decrypt the tag is appended to the cipher text and the least significant bit of the tag is the
 9209 rightmost bit and the tag bits are the rightmost $ulTagBits$ bits. In MessageEncrypt the tag is returned in
 9210 the $pTag$ field of CK_GCM_MESSAGE_PARAMS. In MessageDecrypt the tag is provided by the $pTag$
 9211 field of CK_GCM_MESSAGE_PARAMS.

9212 The key type for K must be compatible with **CKM_AES_ECB** and the
 9213 C_EncryptInit()/C_DecryptInit()/C_MessageEncryptInit()/C_MessageDecryptInit() calls shall behave, with
 9214 respect to K , as if they were called directly with **CKM_AES_ECB**, K and NULL parameters.

9215 **6.13.3 AES-CCM authenticated Encryption / Decryption**

9216 For IPsec (RFC 4309) and also for use in ZFS encryption. Generic CCM mode is described in [RFC
 9217 3610].

9218 To set up for AES-CCM use the following process, where K (key), nonce and additional authenticated
 9219 data are as described in [RFC 3610]. AES-CCM uses CK_CCM_PARAMS for Encrypt and Decrypt, and
 9220 CK_CCM_MESSAGE_PARAMS for MessageEncrypt and MessageDecrypt.

9221 Encrypt:

- 9222 • Set the message/data length $ulDataLen$ in the parameter block.
- 9223 • Set the nonce length $ulNonceLen$ and the nonce data $pNonce$ in the parameter block.
- 9224 • Set the AAD data $pAAD$ and size $ulAADLen$ in the parameter block. $pAAD$ may be NULL if
 9225 $ulAADLen$ is 0.
- 9226 • Set the MAC length $ulMACLen$ in the parameter block.
- 9227 • Call C_EncryptInit() for **CKM_AES_CCM** mechanism with parameters and key K .
- 9228 • Call C_Encrypt(), C_EncryptUpdate(), or C_EncryptFinal(), for the plaintext obtaining the final
 9229 ciphertext output and the MAC. The total length of data processed must be $ulDataLen$. The output
 9230 length will be $ulDataLen + ulMACLen$.

9231 Decrypt:

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

- 9232 • Set the message/data length *ulDataLen* in the parameter block. This length must not include the
9233 length of the MAC that is appended to the cipher text.

- 9234 • Set the nonce length *ulNonceLen* and the nonce data *pNonce* in the parameter block.

- 9235 • Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD* may be NULL if
9236 *ulAADLen* is 0.

- 9237 • Set the MAC length *ulMACLen* in the parameter block.

- 9238 • Call *C_DecryptInit()* for **CKM_AES_CCM** mechanism with parameters and key *K*.

- 9239 • Call *C_Decrypt()*, *C_DecryptUpdate()*, or *C_DecryptFinal()*, for the ciphertext, including the
9240 appended MAC, obtaining plaintext output. The total length of data processed must be *ulDataLen*
9241 + *ulMACLen*. Note: since **CKM_AES_CCM** is an AEAD cipher, no data should be returned until
9242 *C_Decrypt()* or *C_DecryptFinal()*.

9243 MessageEncrypt:

- 9244 • Set the message/data length *ulDataLen* in the parameter block.

- 9245 • Set the nonce length *ulNonceLen*.

- 9246 • Set *pNonce* to hold the nonce data returned from *C_EncryptMessage()* and
9247 *C_EncryptMessageBegin()*. If *ulNonceFixedBits* is not zero, then the most significant bits of *pNonce*
9248 contain the fixed nonce. If *nonceGenerator* is set to *CKG_NO_GENERATE*, *pNonce* is an input
9249 parameter with the full nonce.

- 9250 • Set the *ulNonceFixedBits* and *nonceGenerator* fields in the parameter block.

- 9251 • Set the MAC length *ulMACLen* in the parameter block.

- 9252 • Set *pMAC* to hold the MAC data returned from *C_EncryptMessage()* or the final
9253 *C_EncryptMessageNext()*.

- 9254 • Call *C_MessageEncryptInit()* for **CKM_AES_CCM** mechanism key *K*.

- 9255 • Call *C_EncryptMessage()*, or *C_EncryptMessageBegin()* followed by *C_EncryptMessageNext()*⁷.
9256 The mechanism parameter is passed to all three functions.

- 9257 • Call *C_MessageEncryptFinal()* to close the message encryption.

- 9258 • The MAC is returned in *pMac* of the *CK_CCM_MESSAGE_PARAMS* structure.

9259 MessageDecrypt:

- 9260 • Set the message/data length *ulDataLen* in the parameter block.

- 9261 • Set the nonce length *ulNonceLen* and the nonce data *pNonce* in the parameter block

- 9262 • The *ulNonceFixedBits* and *nonceGenerator* fields in the parameter block are ignored.

- 9263 • Set the MAC length *ulMACLen* in the parameter block.

- 9264 • Set the MAC data *pMAC* in the parameter block before *C_DecryptMessage()* or the final
9265 *C_DecryptMessageNext()*.

- 9266 • Call *C_MessageDecryptInit()* for **CKM_AES_CCM** mechanism key *K*.

- 9267 • Call *C_DecryptMessage()*, or *C_DecryptMessageBegin()* followed by *C_DecryptMessageNext()*⁸.
9268 The mechanism parameter is passed to all three functions.

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

⁸ ^{**} indicates 0 or more calls may be made as required

- 9269 • Call C_MessageDecryptFinal() to close the message decryption.

9270 In *pNonce* the least significant bit of the nonce is the rightmost bit. *ulNonceLen* is the length of the nonce
9271 in bytes.

9272 On MessageEncrypt, the meaning of *nonceGenerator* is as follows: CKG_NO_GENERATE means the
9273 nonce is passed in on MessageEncrypt and no internal MAC generation is done. CKG_GENERATE
9274 means that the non-fixed portion of the nonce is generated by the module internally. The generation
9275 method is not defined.

9276 CKG_GENERATE_COUNTER means that the non-fixed portion of the nonce is generated by the module
9277 internally by use of an incrementing counter, the initial IV counter is zero.

9278 CKG_GENERATE_COUNTER_XOR means that the non-fixed portion of the IV is xored with a counter.
9279 The value of the non-fixed portion passed must not vary from call to call. Like
9280 CKG_GENERATE_COUNTER, the counter starts at zero.

9281 CKG_GENERATE_RANDOM means that the non-fixed portion of the nonce is generated by the module
9282 internally using a PRNG. In any case the entire nonce, including the fixed portion, is returned in *pNonce*.

9283 Modules must implement CKG_GENERATE. Modules may also reject *ulNonceFixedBits* values which are
9284 too large. Zero is always an acceptable value for *ulNonceFixedBits*.

9285 In Encrypt and Decrypt the MAC is appended to the cipher text and the least significant byte of the MAC
9286 is the rightmost byte and the MAC bytes are the rightmost *ulMACLen* bytes. In MessageEncrypt the MAC
9287 is returned in the *pMAC* field of CK_CCM_MESSAGE_PARAMS. In MessageDecrypt the MAC is
9288 provided by the *pMAC* field of CK_CCM_MESSAGE_PARAMS.

9289 The key type for K must be compatible with CKM_AES_ECB and the
9290 C_EncryptInit()/C_DecryptInit()/C_MessageEncryptInit()/C_MessageDecryptInit() calls shall behave, with
9291 respect to K, as if they were called directly with CKM_AES_ECB, K and NULL parameters.

9292 6.13.4 AES-GMAC

9293 AES-GMAC, denoted CKM_AES_GMAC, is a mechanism for single and multiple-part signatures and
9294 verification. It is described in NIST Special Publication 800-38D [GMAC]. GMAC is a special case of
9295 GCM that authenticates only the Additional Authenticated Data (AAD) part of the GCM mechanism
9296 parameters. When GMAC is used with C_Sign or C_Verify, pData points to the AAD. GMAC does not
9297 use plaintext or ciphertext.

9298 The signature produced by GMAC, also referred to as a Tag, the tag's length is determined by the
9299 CK_GCM_PARAMS field *ulTagBits*.

9300 The IV length is determined by the CK_GCM_PARAMS field *ullvLen*.

9301 Constraints on key types and the length of data are summarized in the following table:

9302 *Table 115, 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 |

9303 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK_MECHANISM_INFO structure
9304 specify the supported range of AES key sizes, in bytes.

9305 6.13.5 AES GCM and CCM Mechanism parameters

9306 ♦ CK_GENERATOR_FUNCTION

9307 Functions to generate unique IVs and nonces.

```
9308     typedef CK_ULONG CK_GENERATOR_FUNCTION;  
  
9309 ◆ CK_GCM_PARAMS; CK_GCM_PARAMS_PTR
```

9310 CK_GCM_PARAMS is a structure that provides the parameters to the CKM_AES_GCM mechanism
9311 when used for Encrypt or Decrypt. It is defined as follows:

```
9312     typedef struct CK_GCM_PARAMS {  
9313         CK_BYTE_PTR    pIv;  
9314         CK ULONG      ulIvLen;  
9315         CK ULONG      ulIvBits;  
9316         CK_BYTE_PTR    pAAD;  
9317         CK ULONG      ulAADLen;  
9318         CK ULONG      ulTagBits;  
9319     } CK_GCM_PARAMS;
```

9320

9321 The fields of the structure have the following meanings:

| | | |
|------|-----------|--|
| 9322 | plv | pointer to initialization vector |
| 9323 | ullvLen | length of initialization vector in bytes. The length of the initialization vector can be any number between 1 and $(2^{32}) - 1$. 96-bit (12 byte) IV values can be processed more efficiently, so that length is recommended for situations in which efficiency is critical. |
| 9327 | ullvBits | length of initialization vector in bits. Do not use ullvBits to specify the length of the initialization vector, but ullvLen instead. |
| 9329 | pAAD | pointer to additional authentication data. This data is authenticated but not encrypted. |
| 9331 | ulAADLen | length of pAAD in bytes. The length of the AAD can be any number between 0 and $(2^{32}) - 1$. |
| 9333 | ulTagBits | length of authentication tag (output following cipher text) in bits. Can be any value between 0 and 128. |

9335 **CK_GCM_PARAMS_PTR** is a pointer to a **CK_GCM_PARAMS**.

```
9336 ◆ CK_GCM_MESSAGE_PARAMS; CK_GCM_MESSAGE_PARAMS_PTR
```

9337 CK_GCM_MESSAGE_PARAMS is a structure that provides the parameters to the CKM_AES_GCM mechanism when used for MessageEncrypt or MessageDecrypt. It is defined as follows:

```
9339     typedef struct CK_GCM_MESSAGE_PARAMS {  
9340         CK_BYTE_PTR    pIv;  
9341         CK ULONG      ulIvLen;  
9342         CK ULONG      ulIvFixedBits;  
9343         CK_GENERATOR_FUNCTION ivGenerator;  
9344         CK_BYTE_PTR    pTag;  
9345         CK ULONG      ulTagBits;  
9346     } CK_GCM_MESSAGE_PARAMS;
```

9347

9348 The fields of the structure have the following meanings:

9349 plv pointer to initialization vector

9350
 9351 vector can be any number between 1 and (2³²) - 1. 96-bit (12 byte)
 9352 IV values can be processed more efficiently, so that length is
 9353 recommended for situations in which efficiency is critical.
 9354
 9355 new IV. These bits are counted from the Most significant bits (to the
 9356 right).
 9357 ivGenerator Function used to generate a new IV. Each IV must be unique for a
 9358 given session.
 9359 pTag location of the authentication tag which is returned on
 9360 MessageEncrypt, and provided on MessageDecrypt.
 9361 ulTagBits length of authentication tag in bits. Can be any value between 0 and
 9362 128.

9363 **CK_GCM_MESSAGE_PARAMS_PTR** is a pointer to a **CK_GCM_MESSAGE_PARAMS**.

9364

9365 ◆ **CK_CCM_PARAMS; CK_CCM_PARAMS_PTR**

9366 **CK_CCM_PARAMS** is a structure that provides the parameters to the **CKM_AES_CCM** mechanism
 9367 when used for Encrypt or Decrypt. It is defined as follows:

```

 9368        typedef struct CK_CCM_PARAMS {
 9369            CK ULONG       ulDataLen; /*plaintext or ciphertext*/
 9370            CK BYTE PTR  pNonce;
 9371            CK ULONG       ulNonceLen;
 9372            CK BYTE PTR  pAAD;
 9373            CK ULONG       ulAADLen;
 9374            CK ULONG       ulMACLen;
 9375        } CK_CCM_PARAMS;
  
```

9376 The fields of the structure have the following meanings, where L is the size in bytes of the data length's
 9377 length (2 ≤ L ≤ 8):

| | |
|---|---|
| 9378 ulDataLen 9379 pNonce 9380 ulNonceLen 9381 pAAD 9383 ulAADLen 9384 ulMACLen | length of the data where 0 ≤ ulDataLen < 2^(8L). the nonce. length of pNonce in bytes where 7 ≤ ulNonceLen ≤ 13. Additional authentication data. This data is authenticated but not encrypted. length of pAAD in bytes where 0 ≤ ulAADLen ≤ (2 ³²) - 1. length of the MAC (output following cipher text) in bytes. Valid values are 4, 6, 8, 10, 12, 14, and 16. |
|---|---|

9386 **CK_CCM_PARAMS_PTR** is a pointer to a **CK_CCM_PARAMS**.

9387 ◆ **CK_CCM_MESSAGE_PARAMS; CK_CCM_MESSAGE_PARAMS_PTR**

9388 **CK_CCM_MESSAGE_PARAMS** is a structure that provides the parameters to the **CKM_AES_CCM**
 9389 mechanism when used for MessageEncrypt or MessageDecrypt. It is defined as follows:

```

 9390        typedef struct CK_CCM_MESSAGE_PARAMS {
 9391            CK ULONG       ulDataLen; /*plaintext or ciphertext*/
 9392            CK BYTE PTR  pNonce;
  
```

```

9393     CK ULONG      ulNonceLen;
9394     CK ULONG      ulNonceFixedBits;
9395     CK GENERATOR FUNCTION  nonceGenerator;
9396     CK BYTE PTR   pMAC;
9397     CK ULONG      ulMACLen;
9398 } CK_CCM_MESSAGE_PARAMS;

9399
9400 The fields of the structure have the following meanings, where L is the size in bytes of the data length's
9401 length (2 ≤ L ≤ 8):
9402     ulDataLen    length of the data where 0 ≤ ulDataLen < 2^(8L).
9403     pNonce       the nonce.
9404     ulNonceLen   length of pNonce in bytes where 7 ≤ ulNonceLen ≤ 13.
9405     ulNonceFixedBits number of bits of the original nonce to preserve when generating a
9406                           new nonce. These bits are counted from the Most significant bits (to
9407                           the right).
9408     nonceGenerator Function used to generate a new nonce. Each nonce must be
9409                           unique for a given session.
9410     pMAC          location of the CCM MAC returned on MessageEncrypt, provided on
9411                           MessageDecrypt
9412     ulMACLen     length of the MAC (output following cipher text) in bytes. Valid
9413                           values are 4, 6, 8, 10, 12, 14, and 16.

9414 CK_CCM_MESSAGE_PARAMS_PTR is a pointer to a CK_CCM_MESSAGE_PARAMS.
9415

```

6.14 AES CMAC

Table 116, Mechanisms vs. Functions

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

¹ SR = SignRecover, VR = VerifyRecover.

6.14.1 Definitions

Mechanisms:

```

CKM_AES_CMAC_GENERAL
CKM_AES_CMAC

```

6.14.2 Mechanism parameters

CKM_AES_CMAC_GENERAL uses the existing CK_MAC_GENERAL_PARAMS structure.
CKM_AES_CMAC does not use a mechanism parameter.

6.14.3 General-length AES-CMAC

General-length AES-CMAC, denoted **CKM_AES_CMAC_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 117, General-length AES-CMAC: Key And Data Length

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

References [NIST SP800-38B] and [RFC 4493] recommend that the output MAC is not truncated to less than 64 bits. The MAC length must be specified before the communication starts, and must not be changed during the lifetime of the key. It is the caller's responsibility to follow these rules.

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

6.14.4 AES-CMAC

AES-CMAC, denoted **CKM_AES_CMAC**, is a special case of the general-length AES-CMAC mechanism. AES-MAC always produces and verifies MACs that are a full block size in length, the default output length specified by [RFC 4493].

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

Table 118, AES-CMAC: Key And Data Length

| Function | Key type | Data length | Signature length |
|----------|----------|-------------|-----------------------|
| C_Sign | CKK_AES | any | Block size (16 bytes) |
| C_Verify | CKK_AES | any | Block size (16 bytes) |

References [NIST SP800-38B] and [RFC 4493] recommend that the output MAC is not truncated to less than 64 bits. The MAC length must be specified before the communication starts, and must not be changed during the lifetime of the key. It is the caller's responsibility to follow these rules.

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

6.15 AES XTS

Table 119, Mechanisms vs. Functions

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

6.15.1 Definitions

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

Mechanisms:

CKM_AES_XTS

CKM_AES_XTS_KEY_GEN

6.15.2 AES-XTS secret key objects

Table 120, AES-XTS Secret Key Object Attributes

| Attribute | Data type | Meaning |
|--------------------------------|------------|------------------------------|
| CKA_VALUE ^{1,4,6,7} | Byte array | Key value (32 or 64 bytes) |
| CKA_VALUE_LEN ^{2,3,6} | CK ULONG | Length in bytes of key value |

Refer to Table 11 for footnotes

6.15.3 AES-XTS key generation

The double-length AES-XTS key generation mechanism, denoted **CKM_AES_XTS_KEY_GEN**, is a key generation mechanism for double-length AES-XTS keys.

The mechanism generates AES-XTS keys with a particular length in bytes as specified in the CKA_VALUE_LEN attributes of the template for the key.

This mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, and CKA_VALUE attributes to the new key. Other attributes supported by the double-length AES-XTS key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

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

6.15.4 AES-XTS

AES-XTS (XEX-based Tweaked CodeBook mode with CipherText Stealing), denoted **CKM_AES_XTS**, is a mechanism for single- and multiple-part encryption and decryption. It is specified in NIST SP800-38E.

Its single parameter is a Data Unit Sequence Number 16 bytes long. Supported key lengths are 32 and 64 bytes. Keys are internally split into half-length sub-keys of 16 and 32 bytes respectively. Constraints on key types and the length of data are summarized in the following table:

Table 121, AES-XTS: Key And Data Length

| Function | Key type | Input length | Output length | Comments |
|-----------|-------------|------------------------------|----------------------|---------------|
| C_Encrypt | CKK_AES_XTS | Any, ≥ block size (16 bytes) | Same as input length | No final part |
| C_Decrypt | CKK_AES_XTS | Any, ≥ block size (16 bytes) | Same as input length | No final part |

9480

6.16 AES Key Wrap

Table 122, AES Key Wrap Mechanisms vs. Functions

9483

| Mechanism | Functions | | | | | | |
|------------------------|-------------------|---------------|----------------------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_AES_KEY_WRAP | ✓ | | | | | ✓ | |
| CKM_AES_KEY_WRAP_PAD | ✓ | | | | | ✓ | |
| CKM_AES_KEY_WRAP_KWP | ✓ | | | | | ✓ | |
| CKM_AES_KEY_WRAP_PKCS7 | ✓ | | | | | ✓ | |

¹SR = SignRecover, VR = VerifyRecover

6.16.1 Definitions

9485 Mechanisms:

9486 CKM_AES_KEY_WRAP
 9487 CKM_AES_KEY_WRAP_PAD
 9488 CKM_AES_KEY_WRAP_KWP
 9489 CKM_AES_KEY_WRAP_PKCS7

6.16.2 AES Key Wrap Mechanism parameters

9491 The mechanisms will accept an optional mechanism parameter as the Initialization vector which, if
 9492 present, must be a fixed size array of 8 bytes for CKM_AES_KEY_WRAP and
 9493 CKM_AES_KEY_WRAP_PKCS7, resp. 4 bytes for CKM_AES_KEY_WRAP_KWP; and, if NULL, will use
 9494 the default initial value defined in Section 4.3 resp. 6.2 / 6.3 of [AES KEYWRAP].

9495 The type of this parameter is CK_BYTE_PTR and the pointer points to the array of bytes to be used as
 9496 the initial value. The length shall be either 0 and the pointer NULL; or 8 for CKM_AES_KEY_WRAP and
 9497 CKM_AES_KEY_WRAP_PKCS7, resp. 4 for CKM_AES_KEY_WRAP_KWP, and the pointer non-NULL.

6.16.3 AES Key Wrap

9499 The mechanisms support only single-part operations, i.e. single part wrapping and unwrapping, and
 9500 single-part encryption and decryption.

◆ CKM_AES_KEY_WRAP

9502 The CKM_AES_KEY_WRAP mechanism can wrap a key of any length. A secret key whose length is not
 9503 a multiple of the AES Key Wrap semiblock size (8 bytes) will be zero padded to fit. Semiblock size is
 9504 defined in Section 5.2 of [AES KEYWRAP]. A private key will be encoded as defined in section 6.7; the
 9505 encoded private key will be zero padded to fit if necessary.

9506

9507 The CKM_AES_KEY_WRAP mechanism can only encrypt a block of data whose size is an exact multiple
9508 of the AES Key Wrap algorithm semiblock size.

9509

9510 For unwrapping, the mechanism decrypts the wrapped key. In case of a secret key, it truncates the result
9511 according to the CKA_KEY_TYPE attribute of the template and, if it has one and the key type supports it,
9512 the CKA_VALUE_LEN attribute of the template. The length specified in the template must not be less
9513 than n-7 bytes, where n is the length of the wrapped key. In case of a private key, the mechanism parses
9514 the encoding as defined in section 6.7 and ignores trailing zero bytes.

9515 ◆ **CKM_AES_KEY_WRAP_PAD**

9516 The CKM_AES_KEY_WRAP_PAD mechanism is deprecated. CKM_AES_KEY_WRAP_KWP resp.
9517 CKM_AES_KEY_WRAP_PKCS7 shall be used instead.

9518 ◆ **CKM_AES_KEY_WRAP_KWP**

9519 The CKM_AES_KEY_WRAP_KWP mechanism can wrap a key or encrypt block of data of any length.
9520 The input is zero-padded and wrapped / encrypted as defined in Section 6.3 of [AES KEYWRAP], which
9521 produces same results as RFC 5649.

9522 ◆ **CKM_AES_KEY_WRAP_PKCS7**

9523 The CKM_AES_KEY_WRAP_PKCS7 mechanism can wrap a key or encrypt a block of data of any
9524 length. It does the padding detailed in [PKCS #7] of inputs (keys or data blocks) up to a semiblock size to
9525 make it an exact multiple of AES Key Wrap algorithm semiblock size (8bytes), always producing
9526 wrapped output that is larger than the input key/data to be wrapped. This padding is done by the token
9527 before being passed to the AES key wrap algorithm, which then wraps / encrypts the padded block of
9528 data as defined in Section 6.2 of [AES KEYWRAP].

6.17 Key derivation by data encryption – DES & AES

9530 These mechanisms allow derivation of keys using the result of an encryption operation as the key value.
9531 They are for use with the C_DeriveKey function.

9532 *Table 123, Key derivation by data encryption Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|---------------------------|-------------------|---------------|----------------------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_DES_ECB_ENCRYPT_DATA | | | | | | | ✓ |
| CKM_DES_CBC_ENCRYPT_DATA | | | | | | | ✓ |
| CKM_DES3_ECB_ENCRYPT_DATA | | | | | | | ✓ |
| CKM_DES3_CBC_ENCRYPT_DATA | | | | | | | ✓ |
| CKM_AES_ECB_ENCRYPT_DATA | | | | | | | ✓ |
| CKM_AES_CBC_ENCRYPT_DATA | | | | | | | ✓ |

6.17.1 Definitions

9534 Mechanisms:

```

9535      CKM_DES_ECB_ENCRYPT_DATA
9536      CKM_DES_CBC_ENCRYPT_DATA
9537      CKM_DES3_ECB_ENCRYPT_DATA
9538      CKM_DES3_CBC_ENCRYPT_DATA
9539      CKM_AES_ECB_ENCRYPT_DATA
9540      CKM_AES_CBC_ENCRYPT_DATA
9541
9542      typedef struct CK_DES_CBC_ENCRYPT_DATA_PARAMS {
9543          CK_BYTE           iv[8];
9544          CK_BYTE_PTR      pData;
9545          CK ULONG         length;
9546      } CK_DES_CBC_ENCRYPT_DATA_PARAMS;
9547
9548      typedef CK_DES_CBC_ENCRYPT_DATA_PARAMS CK_PTR
9549                  CK_DES_CBC_ENCRYPT_DATA_PARAMS_PTR;
9550
9551      typedef struct CK_AES_CBC_ENCRYPT_DATA_PARAMS {
9552          CK_BYTE           iv[16];
9553          CK_BYTE_PTR      pData;
9554          CK ULONG         length;
9555      } CK_AES_CBC_ENCRYPT_DATA_PARAMS;
9556
9557      typedef CK_AES_CBC_ENCRYPT_DATA_PARAMS CK_PTR
9558                  CK_AES_CBC_ENCRYPT_DATA_PARAMS_PTR;

```

9559 6.17.2 Mechanism Parameters

9560 Uses CK_KEY_DERIVATION_STRING_DATA as defined in section 6.43.2

9561 *Table 124, Mechanism Parameters*

| | |
|---|--|
| CKM_DES_ECB_ENCRYPT_DATA CKM_DES3_ECB_ENCRYPT_DATA | Uses CK_KEY_DERIVATION_STRING_DATA structure. Parameter is the data to be encrypted and must be a multiple of 8 bytes long. |
| CKM_AES_ECB_ENCRYPT_DATA | Uses CK_KEY_DERIVATION_STRING_DATA structure. Parameter is the data to be encrypted and must be a multiple of 16 long. |
| CKM_DES_CBC_ENCRYPT_DATA CKM_DES3_CBC_ENCRYPT_DATA | Uses CK_DES_CBC_ENCRYPT_DATA_PARAMS. Parameter is an 8 byte IV value followed by the data. The data value part must be a multiple of 8 bytes long. |
| CKM_AES_CBC_ENCRYPT_DATA | Uses CK_AES_CBC_ENCRYPT_DATA_PARAMS. Parameter is an 16 byte IV value followed by the data. The data value part must be a multiple of 16 bytes long. |

9562 6.17.3 Mechanism Description

9563 The mechanisms will function by performing the encryption over the data provided using the base key.
 9564 The resulting cipher text shall be used to create the key value of the resulting key. If not all the cipher text
 9565 is used then the part discarded will be from the trailing end (least significant bytes) of the cipher text data.
 9566 The derived key shall be defined by the attribute template supplied but constrained by the length of cipher
 9567 text available for the key value and other normal PKCS11 derivation constraints.

9568 Attribute template handling, attribute defaulting and key value preparation will operate as per the SHA-1
 9569 Key Derivation mechanism in section 6.20.5.
 9570 If the data is too short to make the requested key then the mechanism returns
 9571 CKR_DATA_LEN_RANGE.

9572 **6.18 Double and Triple-length DES**

9573 *Table 125, Double and Triple-Length DES Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|----------------------|-------------------|---------------|----------------------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_DES2_KEY_GEN | | | | | ✓ | | |
| CKM_DES3_KEY_GEN | | | | | ✓ | | |
| CKM_DES3_ECB | ✓ | | | | | ✓ | |
| CKM_DES3_CBC | ✓ | | | | | ✓ | |
| CKM_DES3_CBC_PAD | ✓ | | | | | ✓ | |
| CKM_DES3_MAC_GENERAL | | ✓ | | | | | |
| CKM_DES3_MAC | | ✓ | | | | | |

9574 **6.18.1 Definitions**

9575 This section defines the key type “CKK_DES2” and “CKK_DES3” for type CK_KEY_TYPE as used in the
 9576 CKA_KEY_TYPE attribute of key objects.

9577 Mechanisms:

9578 CKM_DES2_KEY_GEN
 9579 CKM_DES3_KEY_GEN
 9580 CKM_DES3_ECB
 9581 CKM_DES3_CBC
 9582 CKM_DES3_MAC
 9583 CKM_DES3_MAC_GENERAL
 9584 CKM_DES3_CBC_PAD

9585 **6.18.2 DES2 secret key objects**

9586 DES2 secret key objects (object class **CKO_SECRET_KEY**, key type **CKK_DES2**) hold double-length
 9587 DES keys. The following table defines the DES2 secret key object attributes, in addition to the common
 9588 attributes defined for this object class:

9589 *Table 126, DES2 Secret Key Object Attributes*

| Attribute | Data type | Meaning |
|------------------------------|------------|----------------------------------|
| CKA_VALUE ^{1,4,6,7} | Byte array | Key value (always 16 bytes long) |

9590 ¹Refer to Table 11 for footnotes

9591 DES2 keys must always have their parity bits properly set as described in FIPS PUB 46-3 (*i.e.*, each of
 9592 the DES keys comprising a DES2 key must have its parity bits properly set). Attempting to create or
 9593 unwrap a DES2 key with incorrect parity will return an error.

9594 The following is a sample template for creating a double-length DES secret key object:

9595 CK_OBJECT_CLASS class = CKO_SECRET_KEY;

```

9596 CK_KEY_TYPE keyType = CKK_DES2;
9597 CK_UTF8CHAR label[] = "A DES2 secret key object";
9598 CK_BYTE value[16] = {...};
9599 CK_BBOOL true = CK_TRUE;
9600 CK_ATTRIBUTE template[] = {
9601     {CKA_CLASS, &class, sizeof(class)},
9602     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
9603     {CKA_TOKEN, &true, sizeof(true)},
9604     {CKA_LABEL, label, sizeof(label)-1},
9605     {CKA_ENCRYPT, &true, sizeof(true)},
9606     {CKA_VALUE, value, sizeof(value)}}
9607 };
9608
9609 CKA_CHECK_VALUE: The value of this attribute is derived from the key object by taking the first three
9610 bytes of the ECB encryption of a single block of null (0x00) bytes, using the default cipher associated with
9611 the key type of the secret key object.

```

6.18.3 DES3 secret key objects

DES3 secret key objects (object class **CKO_SECRET_KEY**, key type **CKK_DES3**) hold triple-length DES keys. The following table defines the DES3 secret key object attributes, in addition to the common attributes defined for this object class:

Table 127, DES3 Secret Key Object Attributes

| Attribute | Data type | Meaning |
|------------------------------|------------|----------------------------------|
| CKA_VALUE ^{1,4,6,7} | Byte array | Key value (always 24 bytes long) |

¹Refer to Table 11 for footnotes

DES3 keys must always have their parity bits properly set as described in FIPS PUB 46-3 (*i.e.*, each of the DES keys comprising a DES3 key must have its parity bits properly set). Attempting to create or unwrap a DES3 key with incorrect parity will return an error.

The following is a sample template for creating a triple-length DES secret key object:

```

9622 CK_OBJECT_CLASS class = CKO_SECRET_KEY;
9623 CK_KEY_TYPE keyType = CKK_DES3;
9624 CK_UTF8CHAR label[] = "A DES3 secret key object";
9625 CK_BYTE value[24] = {...};
9626 CK_BBOOL true = CK_TRUE;
9627 CK_ATTRIBUTE template[] = {
9628     {CKA_CLASS, &class, sizeof(class)},
9629     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
9630     {CKA_TOKEN, &true, sizeof(true)},
9631     {CKA_LABEL, label, sizeof(label)-1},
9632     {CKA_ENCRYPT, &true, sizeof(true)},
9633     {CKA_VALUE, value, sizeof(value)}}
9634 };
9635

```

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.

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:

$$\begin{aligned} \text{DES3-E}(\{K1, K2, K3\}, P) &= E(K3, D(K2, E(K1, P))) \\ \text{DES3-D}(\{K1, K2, K3\}, C) &= D(K1, E(K2, D(K3, P))) \end{aligned}$$

6.18.6 Triple-length DES in CBC Mode

Triple-length DES operations in CBC mode, with double or triple-length keys, are performed using outer CBC as defined in X9.52. X9.52 describes this mode as TCBC. The mathematical representations of the CBC encrypt and decrypt operations are as follows:

$$\begin{aligned} \text{DES3-CBC-E}(\{K1, K2, K3\}, P) &= E(K3, D(K2, E(K1, P + I))) \\ \text{DES3-CBC-D}(\{K1, K2, K3\}, C) &= D(K1, E(K2, D(K3, P))) + I \end{aligned}$$

The value *I* is either an 8-byte initialization vector or the previous block of cipher text that is added to the current input block. The addition operation is used is addition modulo-2 (XOR).

6.18.7 DES and Triple length DES in OFB Mode

Table 128, DES and Triple Length DES in OFB Mode Mechanisms vs. Functions

| Mechanism | Functions | | | | | | |
|---------------|-------------------|---------------|----------------------|--------|-------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/Key Pair | Wrap & Unwrap | Derive |
| CKM_DES_OFB64 | ✓ | | | | | | |
| CKM_DES_OFB8 | ✓ | | | | | | |
| CKM_DES_CFB64 | ✓ | | | | | | |
| CKM_DES_CFB8 | ✓ | | | | | | |

9670

Cipher DES has a output feedback mode, DES-OFB, denoted **CKM_DES_OFB8** and **CKM_DES_OFB64**. It is a mechanism for single and multiple-part encryption and decryption with DES.

- 9673 It has a parameter, an initialization vector for this mode. The initialization vector has the same length as
 9674 the block size.
- 9675 Constraints on key types and the length of data are summarized in the following table:
- 9676 *Table 129, OFB: Key And Data Length*

| Function | Key type | Input length | Output length | Comments |
|-----------|-----------------------------------|--------------|----------------------|---------------|
| C_Encrypt | CKK_DES, CKK_DES2, CKK_DES3 | any | same as input length | no final part |
| C_Decrypt | CKK_DES, CKK_DES2, CKK_DES3 | any | same as input length | no final part |

- 9677 For this mechanism the **CK_MECHANISM_INFO** structure is as specified for CBC mode.

6.18.8 DES and Triple length DES in CFB Mode

- 9679 Cipher DES has a cipher feedback mode, DES-CFB, denoted **CKM_DES_CFB8** and **CKM_DES_CFB64**.
 9680 It is a mechanism for single and multiple-part encryption and decryption with DES.
 9681 It has a parameter, an initialization vector for this mode. The initialization vector has the same length as
 9682 the block size.
 9683 Constraints on key types and the length of data are summarized in the following table:
 9684 *Table 130, CFB: Key And Data Length*

| Function | Key type | Input length | Output length | Comments |
|-----------|-----------------------------------|--------------|----------------------|---------------|
| C_Encrypt | CKK_DES, CKK_DES2, CKK_DES3 | any | same as input length | no final part |
| C_Decrypt | CKK_DES, CKK_DES2, CKK_DES3 | any | same as input length | no final part |

- 9685 For this mechanism the **CK_MECHANISM_INFO** structure is as specified for CBC mode.

6.19 Double and Triple-length DES CMAC

- 9686 *Table 131, Double and Triple-length DES CMAC Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|-----------------------|-------------------|---------------|----------------------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_DES3_CMAC_GENERAL | | ✓ | | | | | |
| CKM_DES3_CMAC | | ✓ | | | | | |

9688 ¹ SR = SignRecover, VR = VerifyRecover.

6.19.1 Definitions

- 9690 Mechanisms:

9691 **CKM_DES3_CMAC_GENERAL**
 9692 **CKM_DES3_CMAC**

9693 **6.19.2 Mechanism parameters**

9694 CKM_DES3_CMAC_GENERAL uses the existing **CK_MAC_GENERAL_PARAMS** structure.
9695 CKM_DES3_CMAC does not use a mechanism parameter.

9696 **6.19.3 General-length DES3-MAC**

9697 General-length DES3-CMAC, denoted **CKM_DES3_CMAC_GENERAL**, is a mechanism for single- and
9698 multiple-part signatures and verification with DES3 or DES2 keys, based on [NIST sp800-38b].

9699 It has a parameter, a **CK_MAC_GENERAL_PARAMS** structure, which specifies the output length
9700 desired from the mechanism.

9701 The output bytes from this mechanism are taken from the start of the final DES3 cipher block produced in
9702 the MACing process.

9703 Constraints on key types and the length of data are summarized in the following table:

9704 *Table 132, General-length DES3-CMAC: Key And Data Length*

| Function | Key type | Data length | Signature length |
|----------|----------------------|-------------|--|
| C_Sign | CKK_DES3 CKK_DES2 | any | 1-block size, as specified in parameters |
| C_Verify | CKK_DES3 CKK_DES2 | any | 1-block size, as specified in parameters |

9705 Reference [NIST sp800-38b] recommends that the output MAC is not truncated to less than 64 bits
9706 (which means using the entire block for DES). The MAC length must be specified before the
9707 communication starts, and must not be changed during the lifetime of the key. It is the caller's
9708 responsibility to follow these rules.

9709 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
9710 are not used

9711 **6.19.4 DES3-CMAC**

9712 DES3-CMAC, denoted **CKM_DES3_CMAC**, is a special case of the general-length DES3-CMAC
9713 mechanism. DES3-MAC always produces and verifies MACs that are a full block size in length, since the
9714 DES3 block length is the minimum output length recommended by [NIST sp800-38b].

9715 Constraints on key types and the length of data are summarized in the following table:

9716 *Table 133, DES3-CMAC: Key And Data Length*

| Function | Key type | Data length | Signature length |
|----------|----------------------|-------------|----------------------|
| C_Sign | CKK_DES3 CKK_DES2 | any | Block size (8 bytes) |
| C_Verify | CKK_DES3 CKK_DES2 | any | Block size (8 bytes) |

9717 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
9718 are not used.

9719 **6.20 SHA-1**

9720 *Table 134, SHA-1 Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|-------------------------|-------------------|---------------|----------------------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_SHA_1 | | | | ✓ | | | |
| CKM_SHA_1_HMAC_GENERAL | | ✓ | | | | | |
| CKM_SHA_1_HMAC | | ✓ | | | | | |
| CKM_SHA1_KEY_DERIVATION | | | | | | | ✓ |
| CKM_SHA_1_KEY_GEN | | | | | ✓ | | |

6.20.1 Definitions

This section defines the key type "CKK_SHA_1_HMAC" for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

- CKM_SHA_1
- CKM_SHA_1_HMAC
- CKM_SHA_1_HMAC_GENERAL
- CKM_SHA1_KEY_DERIVATION
- CKM_SHA_1_KEY_GEN

6.20.2 SHA-1 digest

The SHA-1 mechanism, denoted **CKM_SHA_1**, is a mechanism for message digesting, following the Secure Hash Algorithm with a 160-bit message digest defined in FIPS PUB 180-2.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 135, SHA-1: Data Length

| Function | Input length | Digest length |
|----------|--------------|---------------|
| C_Digest | any | 20 |

6.20.3 General-length SHA-1-HMAC

The general-length SHA-1-HMAC mechanism, denoted **CKM_SHA_1_HMAC_GENERAL**, is a mechanism for signatures and verification. It uses the HMAC construction, based on the SHA-1 hash function. The keys it uses are generic secret keys and CKK_SHA_1_HMAC.

It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired output. This length should be in the range 1-20 (the output size of SHA-1 is 20 bytes). Signatures (MACs) produced by this mechanism will be taken from the start of the full 20-byte HMAC output.

Table 136, General-length SHA-1-HMAC: Key And Data Length

| Function | Key type | Data length | Signature length |
|----------|----------------------------------|-------------|-------------------------------|
| C_Sign | generic secret CKK_SHA_1_HMAC | any | 1-20, depending on parameters |
| C_Verify | generic secret CKK_SHA_1_HMAC | any | 1-20, depending on parameters |

9746 6.20.4 SHA-1-HMAC

9747 The SHA-1-HMAC mechanism, denoted **CKM_SHA_1_HMAC**, is a special case of the general-length
 9748 SHA-1-HMAC mechanism in Section 6.20.3.

9749 It has no parameter, and always produces an output of length 20.

9750 6.20.5 SHA-1 key derivation

9751 SHA-1 key derivation, denoted **CKM_SHA1_KEY_DERIVATION**, is a mechanism which provides the
 9752 capability of deriving a secret key by digesting the value of another secret key with SHA-1.

9753 The value of the base key is digested once, and the result is used to make the value of derived secret
 9754 key.

- 9755 • If no length or key type is provided in the template, then the key produced by this mechanism will be a
 9756 generic secret key. Its length will be 20 bytes (the output size of SHA-1).
- 9757 • If no key type is provided in the template, but a length is, then the key produced by this mechanism
 9758 will be a generic secret key of the specified length.
- 9759 • If no length was provided in the template, but a key type is, then that key type must have a well-
 9760 defined length. If it does, then the key produced by this mechanism will be of the type specified in the
 9761 template. If it doesn't, an error will be returned.
- 9762 • If both a key type and a length are provided in the template, the length must be compatible with that
 9763 key type. The key produced by this mechanism will be of the specified type and length.

9764 If a DES, DES2, or CDMF key is derived with this mechanism, the parity bits of the key will be set
 9765 properly.

9766 If the requested type of key requires more than 20 bytes, such as DES3, an error is generated.

9767 This mechanism has the following rules about key sensitivity and extractability:

- 9768 • The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both
 9769 be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some
 9770 default value.
- 9771 • If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key
 9772 will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the
 9773 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
 9774 **CKA_SENSITIVE** attribute.
- 9775 • Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
 9776 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
 9777 CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
 9778 value from its **CKA_EXTRACTABLE** attribute.

9779 6.20.6 SHA-1 HMAC key generation

9780 The SHA-1-HMAC key generation mechanism, denoted **CKM_SHA_1_KEY_GEN**, is a key generation
 9781 mechanism for NIST's SHA-1-HMAC.

9782 It does not have a parameter.

9783 The mechanism generates SHA-1-HMAC keys with a particular length in bytes, as specified in the
 9784 **CKA_VALUE_LEN** attribute of the template for the key.
 9785 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
 9786 key. Other attributes supported by the SHA-1-HMAC key type (specifically, the flags indicating which
 9787 functions the key supports) may be specified in the template for the key, or else are assigned default
 9788 initial values.
 9789 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 9790 specify the supported range of **CKM_SHA_1_HMAC** key sizes, in bytes.

9791 **6.21 SHA-224**

9792 *Table 137, SHA-224 Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|---------------------------|------------------------------|--------------------------|--------------------------------|---------------|---------------------------|--------------------------|---------------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR¹ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_SHA224 | | | | ✓ | | | |
| CKM_SHA224_HMAC | | ✓ | | | | | |
| CKM_SHA224_HMAC_GENERAL | | ✓ | | | | | |
| CKM_SHA224_KEY_DERIVATION | | | | | | | ✓ |
| CKM_SHA224_KEY_GEN | | | | | ✓ | | |

9793 **6.21.1 Definitions**

9794 This section defines the key type "CKK_SHA224_HMAC" for type CK_KEY_TYPE as used in the
 9795 CKA_KEY_TYPE attribute of key objects.

9796 Mechanisms:

9797 CKM_SHA224
 9798 CKM_SHA224_HMAC
 9799 CKM_SHA224_HMAC_GENERAL
 9800 CKM_SHA224_KEY_DERIVATION
 9801 CKM_SHA224_KEY_GEN

9802 **6.21.2 SHA-224 digest**

9803 The SHA-224 mechanism, denoted **CKM_SHA224**, is a mechanism for message digesting, following the
 9804 Secure Hash Algorithm with a 224-bit message digest defined in FIPS PUB 180-4.

9805 It does not have a parameter.

9806 Constraints on the length of input and output data are summarized in the following table. For single-part
 9807 digesting, the data and the digest may begin at the same location in memory.

9808 *Table 138, SHA-224: Data Length*

| Function | Input length | Digest length |
|-----------------|---------------------|----------------------|
| C_Digest | any | 28 |

9809 **6.21.3 General-length SHA-224-HMAC**

9810 The general-length SHA-224-HMAC mechanism, denoted **CKM_SHA224_HMAC_GENERAL**, is the
 9811 same as the general-length SHA-1-HMAC mechanism except that it uses the HMAC construction based

9812 on the SHA-224 hash function and length of the output should be in the range 1-28. The keys it uses are
9813 generic secret keys and CKK_SHA224_HMAC. FIPS-198 compliant tokens may require the key length to
9814 be at least 14 bytes; that is, half the size of the SHA-224 hash output.

9815 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired
9816 output. This length should be in the range 1-28 (the output size of SHA-224 is 28 bytes). FIPS-198
9817 compliant tokens may constrain the output length to be at least 4 or 14 (half the maximum length).
9818 Signatures (MACs) produced by this mechanism will be taken from the start of the full 28-byte HMAC
9819 output.

9820 *Table 139, General-length SHA-224-HMAC: Key And Data Length*

| Function | Key type | Data length | Signature length |
|----------|-----------------------------------|-------------|-------------------------------|
| C_Sign | generic secret CKK_SHA224_HMAC | Any | 1-28, depending on parameters |
| C_Verify | generic secret CKK_SHA224_HMAC | Any | 1-28, depending on parameters |

9821 **6.21.4 SHA-224-HMAC**

9822 The SHA-224-HMAC mechanism, denoted **CKM_SHA224_HMAC**, is a special case of the general-length
9823 SHA-224-HMAC mechanism.

9824 It has no parameter, and always produces an output of length 28.

9825 **6.21.5 SHA-224 key derivation**

9826 SHA-224 key derivation, denoted **CKM_SHA224_KEY_DERIVATION**, is the same as the SHA-1 key
9827 derivation mechanism in Section 6.20.5 except that it uses the SHA-224 hash function and the relevant
9828 length is 28 bytes.

9829 **6.21.6 SHA-224 HMAC key generation**

9830 The SHA-224-HMAC key generation mechanism, denoted **CKM_SHA224_KEY_GEN**, is a key
9831 generation mechanism for NIST's SHA224-HMAC.

9832 It does not have a parameter.

9833 The mechanism generates SHA224-HMAC keys with a particular length in bytes, as specified in the
9834 **CKA_VALUE_LEN** attribute of the template for the key.

9835 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
9836 key. Other attributes supported by the SHA224-HMAC key type (specifically, the flags indicating which
9837 functions the key supports) may be specified in the template for the key, or else are assigned default
9838 initial values.

9839 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
9840 specify the supported range of **CKM_SHA224_HMAC** key sizes, in bytes.

9841 **6.22 SHA-256**

9842 *Table 140, SHA-256 Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|---------------------------|-------------------|---------------|----------------------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_SHA256 | | | | ✓ | | | |
| CKM_SHA256_HMAC_GENERAL | | ✓ | | | | | |
| CKM_SHA256_HMAC | | ✓ | | | | | |
| CKM_SHA256_KEY_DERIVATION | | | | | | | ✓ |
| CKM_SHA256_KEY_GEN | | | | | ✓ | | |

6.22.1 Definitions

This section defines the key type “CKK_SHA256_HMAC” for type CK_KEY_TYPE as used in the 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 180-2.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 141, SHA-256: Data Length

| Function | Input length | Digest length |
|----------|--------------|---------------|
| C_Digest | any | 32 |

6.22.3 General-length SHA-256-HMAC

The general-length SHA-256-HMAC mechanism, denoted **CKM_SHA256_HMAC_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in Section 6.20.3, except that it uses the HMAC construction based on the SHA-256 hash function and length of the output should be in the range 1-32. The keys it uses are generic secret keys and CKK_SHA256_HMAC. FIPS-198 compliant tokens may require the key length to be at least 16 bytes; that is, half the size of the SHA-256 hash output.

It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired output. This length should be in the range 1-32 (the output size of SHA-256 is 32 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 16 (half the maximum length). Signatures (MACs) produced by this mechanism will be taken from the start of the full 32-byte HMAC output.

Table 142, General-length SHA-256-HMAC: Key And Data Length

| Function | Key type | Data length | Signature length |
|----------|------------------------------------|-------------|-------------------------------|
| C_Sign | generic secret, CKK_SHA256_HMAC | Any | 1-32, depending on parameters |
| C_Verify | generic secret, CKK_SHA256_HMAC | Any | 1-32, depending on parameters |

6.22.4 SHA-256-HMAC

The SHA-256-HMAC mechanism, denoted **CKM_SHA256_HMAC**, is a special case of the general-length SHA-256-HMAC mechanism in Section 6.22.3.

It has no parameter, and always produces an output of length 32.

6.22.5 SHA-256 key derivation

SHA-256 key derivation, denoted CKM_SHA256_KEY_DERIVATION, is the same as the SHA-1 key derivation mechanism in Section 6.20.5, except that it uses the SHA-256 hash function and the relevant length is 32 bytes.

6.22.6 SHA-256 HMAC key generation

The SHA-256-HMAC key generation mechanism, denoted **CKM_SHA256_KEY_GEN**, is a key generation mechanism for NIST's SHA256-HMAC.

It does not have a parameter.

The mechanism generates SHA256-HMAC keys with a particular length in bytes, as specified in the **CKA_VALUE_LEN** attribute of the template for the key.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the SHA256-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of **CKM_SHA256_HMAC** key sizes, in bytes.

6.23 SHA-384

Table 143, SHA-384 Mechanisms vs. Functions

| Mechanism | Functions | | | | | | |
|---------------------------|-------------------|---------------|----------------------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_SHA384 | | | | ✓ | | | |
| CKM_SHA384_HMAC_GENERAL | | ✓ | | | | | |
| CKM_SHA384_HMAC | | ✓ | | | | | |
| CKM_SHA384_KEY_DERIVATION | | | | | | | ✓ |
| CKM_SHA384_KEY_GEN | | | | | ✓ | | |

6.23.1 Definitions

This section defines the key type "CKK_SHA384_HMAC" for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

9896 CKM_SHA384
 9897 CKM_SHA384_HMAC
 9898 CKM_SHA384_HMAC_GENERAL
 9899 CKM_SHA384_KEY_DERIVATION
 9900 CKM_SHA384_KEY_GEN

9901 **6.23.2 SHA-384 digest**

9902 The SHA-384 mechanism, denoted **CKM_SHA384**, is a mechanism for message digesting, following the
 9903 Secure Hash Algorithm with a 384-bit message digest defined in FIPS PUB 180-2.

9904 It does not have a parameter.

9905 Constraints on the length of input and output data are summarized in the following table. For single-part
 9906 digesting, the data and the digest may begin at the same location in memory.

9907 *Table 144, SHA-384: Data Length*

| Function | Input length | Digest length |
|----------|--------------|---------------|
| C_Digest | any | 48 |

9908 **6.23.3 General-length SHA-384-HMAC**

9909 The general-length SHA-384-HMAC mechanism, denoted **CKM_SHA384_HMAC_GENERAL**, is the
 9910 same as the general-length SHA-1-HMAC mechanism in Section 6.20.3, except that it uses the HMAC
 9911 construction based on the SHA-384 hash function and length of the output should be in the range 1-48.

9912 The keys it uses are generic secret keys and CKK_SHA384_HMAC. FIPS-198 compliant tokens may
 9913 require the key length to be at least 24 bytes; that is, half the size of the SHA-384 hash output.

9914 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired
 9915 output. This length should be in the range 0-48 (the output size of SHA-384 is 48 bytes). FIPS-198
 9916 compliant tokens may constrain the output length to be at least 4 or 24 (half the maximum length).
 9917 Signatures (MACs) produced by this mechanism will be taken from the start of the full 48-byte HMAC
 9918 output.

9919 *Table 145, General-length SHA-384-HMAC: Key And Data Length*

| Function | Key type | Data length | Signature length |
|----------|------------------------------------|-------------|-------------------------------|
| C_Sign | generic secret, CKK_SHA384_HMAC | Any | 1-48, depending on parameters |
| C_Verify | generic secret, CKK_SHA384_HMAC | Any | 1-48, depending on parameters |

9920

9921 **6.23.4 SHA-384-HMAC**

9922 The SHA-384-HMAC mechanism, denoted **CKM_SHA384_HMAC**, is a special case of the general-length
 9923 SHA-384-HMAC mechanism.

9924 It has no parameter, and always produces an output of length 48.

9925 **6.23.5 SHA-384 key derivation**

9926 SHA-384 key derivation, denoted **CKM_SHA384_KEY_DERIVATION**, is the same as the SHA-1 key
 9927 derivation mechanism in Section 6.20.5, except that it uses the SHA-384 hash function and the relevant
 9928 length is 48 bytes.

6.23.6 SHA-384 HMAC key generation

The SHA-384-HMAC key generation mechanism, denoted **CKM_SHA384_KEY_GEN**, is a key generation mechanism for NIST's SHA384-HMAC.

It does not have a parameter.

The mechanism generates SHA384-HMAC keys with a particular length in bytes, as specified in the **CKA_VALUE_LEN** attribute of the template for the key.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the SHA384-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of **CKM_SHA384_HMAC** key sizes, in bytes.

6.24 SHA-512

Table 146, SHA-512 Mechanisms vs. Functions

| Mechanism | Functions | | | | | | |
|---------------------------|-------------------|---------------|----------------------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_SHA512 | | | | ✓ | | | |
| CKM_SHA512_HMAC_GENERAL | | ✓ | | | | | |
| CKM_SHA512_HMAC | | ✓ | | | | | |
| CKM_SHA512_KEY_DERIVATION | | | | | | | ✓ |
| CKM_SHA512_KEY_GEN | | | | | ✓ | | |

6.24.1 Definitions

This section defines the key type "CKK_SHA512_HMAC" for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

CKM_SHA512

CKM_SHA512_HMAC

CKM_SHA512_HMAC_GENERAL

CKM_SHA512_KEY_DERIVATION

CKM_SHA512_KEY_GEN

6.24.2 SHA-512 digest

The SHA-512 mechanism, denoted **CKM_SHA512**, is a mechanism for message digesting, following the Secure Hash Algorithm with a 512-bit message digest defined in FIPS PUB 180-2.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 147, SHA-512: Data Length

| Function | Input length | Digest length |
|----------|--------------|---------------|
| C_Digest | any | 64 |

9959 6.24.3 General-length SHA-512-HMAC

9960 The general-length SHA-512-HMAC mechanism, denoted **CKM_SHA512_HMAC_GENERAL**, is the
 9961 same as the general-length SHA-1-HMAC mechanism in Section 6.20.3, except that it uses the HMAC
 9962 construction based on the SHA-512 hash function and length of the output should be in the range 1-64.

9963 The keys it uses are generic secret keys and CKK_SHA512_HMAC. FIPS-198 compliant tokens may
 9964 require the key length to be at least 32 bytes; that is, half the size of the SHA-512 hash output.

9965 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired
 9966 output. This length should be in the range 0-64 (the output size of SHA-512 is 64 bytes). FIPS-198
 9967 compliant tokens may constrain the output length to be at least 4 or 32 (half the maximum length).
 9968 Signatures (MACs) produced by this mechanism will be taken from the start of the full 64-byte HMAC
 9969 output.

9970 *Table 148, General-length SHA-384-HMAC: Key And Data Length*

| Function | Key type | Data length | Signature length |
|----------|------------------------------------|-------------|-------------------------------|
| C_Sign | generic secret, CKK_SHA512_HMAC | Any | 1-64, depending on parameters |
| C_Verify | generic secret, CKK_SHA512_HMAC | Any | 1-64, depending on parameters |

9971

9972 6.24.4 SHA-512-HMAC

9973 The SHA-512-HMAC mechanism, denoted **CKM_SHA512_HMAC**, is a special case of the general-length
 9974 SHA-512-HMAC mechanism.

9975 It has no parameter, and always produces an output of length 64.

9976 6.24.5 SHA-512 key derivation

9977 SHA-512 key derivation, denoted **CKM_SHA512_KEY_DERIVATION**, is the same as the SHA-1 key
 9978 derivation mechanism in Section 6.20.5, except that it uses the SHA-512 hash function and the relevant
 9979 length is 64 bytes.

9980 6.24.6 SHA-512 HMAC key generation

9981 The SHA-512-HMAC key generation mechanism, denoted **CKM_SHA512_KEY_GEN**, is a key
 9982 generation mechanism for NIST's SHA512-HMAC.

9983 It does not have a parameter.

9984 The mechanism generates SHA512-HMAC keys with a particular length in bytes, as specified in the
 9985 **CKA_VALUE_LEN** attribute of the template for the key.

9986 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
 9987 key. Other attributes supported by the SHA512-HMAC key type (specifically, the flags indicating which
 9988 functions the key supports) may be specified in the template for the key, or else are assigned default
 9989 initial values.

9990 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 9991 specify the supported range of **CKM_SHA512_HMAC** key sizes, in bytes.

9992 **6.25 SHA-512/224**

9993 *Table 149, SHA-512/224 Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|-------------------------------|-------------------|---------------|----------------------|--------|-------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/Key Pair | Wrap & Unwrap | Derive |
| CKM_SHA512_224 | | | | ✓ | | | |
| CKM_SHA512_224_HMAC_GENERAL | | ✓ | | | | | |
| CKM_SHA512_224_HMAC | | ✓ | | | | | |
| CKM_SHA512_224_KEY_DERIVATION | | | | | | | ✓ |
| CKM_SHA512_224_KEY_GEN | | | | | ✓ | | |

9994 **6.25.1 Definitions**

9995 This section defines the key type “CKK_SHA512_224_HMAC” for type CK_KEY_TYPE as used in the
9996 CKA_KEY_TYPE attribute of key objects.

9997 Mechanisms:

9998 CKM_SHA512_224

9999 CKM_SHA512_224_HMAC

10000 CKM_SHA512_224_HMAC_GENERAL

10001 CKM_SHA512_224_KEY_DERIVATION

10002 CKM_SHA512_224_KEY_GEN

10003 **6.25.2 SHA-512/224 digest**

10004 The SHA-512/224 mechanism, denoted **CKM_SHA512_224**, is a mechanism for message digesting,
10005 following the Secure Hash Algorithm defined in FIPS PUB 180-4, section 5.3.6. It is based on a 512-bit
10006 message digest with a distinct initial hash value and truncated to 224 bits. **CKM_SHA512_224** is the
10007 same as **CKM_SHA512_T** with a parameter value of 224.

10008 It does not have a parameter.

10009 Constraints on the length of input and output data are summarized in the following table. For single-part
10010 digesting, the data and the digest may begin at the same location in memory.

10011 *Table 150, SHA-512/224: Data Length*

| Function | Input length | Digest length |
|----------|--------------|---------------|
| C_Digest | any | 28 |

10012 **6.25.3 General-length SHA-512/224-HMAC**

10013 The general-length SHA-512/224-HMAC mechanism, denoted **CKM_SHA512_224_HMAC_GENERAL**,
10014 is the same as the general-length SHA-1-HMAC mechanism in Section 6.20.3, except that it uses the
10015 HMAC construction based on the SHA-512/224 hash function and length of the output should be in the
10016 range 1-28. The keys it uses are generic secret keys and CKK_SHA512_224_HMAC. FIPS-198
10017 compliant tokens may require the key length to be at least 14 bytes; that is, half the size of the SHA-
10018 512/224 hash output.

10019 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired
 10020 output. This length should be in the range 0-28 (the output size of SHA-512/224 is 28 bytes). FIPS-198
 10021 compliant tokens may constrain the output length to be at least 4 or 14 (half the maximum length).
 10022 Signatures (MACs) produced by this mechanism will be taken from the start of the full 28-byte HMAC
 10023 output.

10024 *Table 151, General-length SHA-384-HMAC: Key And Data Length*

| Function | Key type | Data length | Signature length |
|----------|--|-------------|-------------------------------|
| C_Sign | generic secret, CKK_SHA512_224_HMAC | Any | 1-28, depending on parameters |
| C_Verify | generic secret, CKK_SHA512_224_HMAC | Any | 1-28, depending on parameters |

10025

6.25.4 SHA-512/224-HMAC

10027 The SHA-512-HMAC mechanism, denoted **CKM_SHA512_224_HMAC**, is a special case of the general-
 10028 length SHA-512/224-HMAC mechanism.
 10029 It has no parameter, and always produces an output of length 28.

6.25.5 SHA-512/224 key derivation

10031 The SHA-512/224 key derivation, denoted **CKM_SHA512_224_KEY_DERIVATION**, is the same as the
 10032 SHA-512 key derivation mechanism in section 6.24.5, except that it uses the SHA-512/224 hash function
 10033 and the relevant length is 28 bytes.

6.25.6 SHA-512/224 HMAC key generation

10035 The SHA-512/224-HMAC key generation mechanism, denoted **CKM_SHA512_224_KEY_GEN**, is a key
 10036 generation mechanism for NIST's SHA512/224-HMAC.

10037 It does not have a parameter.

10038 The mechanism generates SHA512/224-HMAC keys with a particular length in bytes, as specified in the
 10039 **CKA_VALUE_LEN** attribute of the template for the key.

10040 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
 10041 key. Other attributes supported by the SHA512/224-HMAC key type (specifically, the flags indicating
 10042 which functions the key supports) may be specified in the template for the key, or else are assigned
 10043 default initial values.

10044 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 10045 specify the supported range of **CKM_SHA512_224_HMAC** key sizes, in bytes.

6.26 SHA-512/256

10047 *Table 152, SHA-512/256 Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|----------------|-------------------|---------------|----------------------|--------|-------------------|---------------|-------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/Key Pair | Wrap & Unwrap | Deriv |
| CKM_SHA512_256 | | | | ✓ | | | |

| Mechanism | Functions | | | | | | |
|-------------------------------|-------------------|---------------|----------------------|--------|-------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/Key Pair | Wrap & Unwrap | Derive |
| CKM_SHA512_256_HMAC_GENERAL | | ✓ | | | | | |
| CKM_SHA512_256_HMAC | | ✓ | | | | | |
| CKM_SHA512_256_KEY_DERIVATION | | | | | | | ✓ |
| CKM_SHA512_256_KEY_GEN | | | | | ✓ | | |

10048 6.26.1 Definitions

10049 This section defines the key type “CKK_SHA512_256_HMAC” for type CK_KEY_TYPE as used in the
 10050 CKA_KEY_TYPE attribute of key objects.

10051 Mechanisms:

10052 CKM_SHA512_256
 10053 CKM_SHA512_256_HMAC
 10054 CKM_SHA512_256_HMAC_GENERAL
 10055 CKM_SHA512_256_KEY_DERIVATION
 10056 CKM_SHA512_256_KEY_GEN

10057 6.26.2 SHA-512/256 digest

10058 The SHA-512/256 mechanism, denoted **CKM_SHA512_256**, is a mechanism for message digesting,
 10059 following the Secure Hash Algorithm defined in FIPS PUB 180-4, section 5.3.6. It is based on a 512-bit
 10060 message digest with a distinct initial hash value and truncated to 256 bits. **CKM_SHA512_256** is the
 10061 same as **CKM_SHA512_T** with a parameter value of 256.

10062 It does not have a parameter.

10063 Constraints on the length of input and output data are summarized in the following table. For single-part
 10064 digesting, the data and the digest may begin at the same location in memory.

10065 *Table 153, SHA-512/256: Data Length*

| Function | Input length | Digest length |
|----------|--------------|---------------|
| C_Digest | any | 32 |

10066 6.26.3 General-length SHA-512/256-HMAC

10067 The general-length SHA-512/256-HMAC mechanism, denoted **CKM_SHA512_256_HMAC_GENERAL**,
 10068 is the same as the general-length SHA-1-HMAC mechanism in Section 6.20.3, except that it uses the
 10069 HMAC construction based on the SHA-512/256 hash function and length of the output should be in the
 10070 range 1-32. The keys it uses are generic secret keys and CKK_SHA512_256_HMAC. FIPS-198
 10071 compliant tokens may require the key length to be at least 16 bytes; that is, half the size of the SHA-
 10072 512/256 hash output.

10073 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired
 10074 output. This length should be in the range 1-32 (the output size of SHA-512/256 is 32 bytes). FIPS-198
 10075 compliant tokens may constrain the output length to be at least 4 or 16 (half the maximum length).
 10076 Signatures (MACs) produced by this mechanism will be taken from the start of the full 32-byte HMAC
 10077 output.

10078 *Table 154, General-length SHA-384-HMAC: Key And Data Length*

| Function | Key type | Data length | Signature length |
|----------|--|-------------|-------------------------------|
| C_Sign | generic secret, CKK_SHA512_256_HMAC | Any | 1-32, depending on parameters |
| C_Verify | generic secret, CKK_SHA512_256_HMAC | Any | 1-32, depending on parameters |

10079

10080 **6.26.4 SHA-512/256-HMAC**

10081 The SHA-512-HMAC mechanism, denoted **CKM_SHA512_256_HMAC**, is a special case of the general-length SHA-512/256-HMAC mechanism.

10083 It has no parameter, and always produces an output of length 32.

10084 **6.26.5 SHA-512/256 key derivation**

10085 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.

10088 **6.26.6 SHA-512/256 HMAC key generation**

10089 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.

10091 It does not have a parameter.

10092 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.

10094 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.

10098 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.

10100 **6.27 SHA-512/t**

10101 *Table 155, SHA-512 / t Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|-----------------------------|-------------------|---------------|----------------------|--------|---------------------|---------------|---------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen . Key/ Key Pair | Wrap & Unwrap | Deriv e |
| CKM_SHA512_T | | | | ✓ | | | |
| CKM_SHA512_T_HMAC_GENERAL | | ✓ | | | | | |
| CKM_SHA512_T_HMAC | | ✓ | | | | | |
| CKM_SHA512_T_KEY_DERIVATION | | | | | | | ✓ |

| Mechanism | Functions | | | | | | |
|----------------------|-------------------|---------------|----------------------|--------|---------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ₁ | Digest | Gen . Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_SHA512_T_KEY_GEN | | | | | ✓ | | |

6.27.1 Definitions

This section defines the key type “CKK_SHA512_T_HMAC” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

- CKM_SHA512_T
- CKM_SHA512_T_HMAC
- CKM_SHA512_T_HMAC_GENERAL
- CKM_SHA512_T_KEY_DERIVATION
- CKM_SHA512_T_KEY_GEN

6.27.2 SHA-512/t digest

The SHA-512/t mechanism, denoted **CKM_SHA512_T**, is a mechanism for message digesting, following the Secure Hash Algorithm defined in FIPS PUB 180-4, section 5.3.6. It is based on a 512-bit message digest with a distinct initial hash value and truncated to t bits.

It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the value of t in bits. The length in bytes of the desired output should be in the range of 0– $\lceil t/8 \rceil$, where 0 < t < 512, and t > 384.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 156, SHA-512/256: Data Length

| Function | Input length | Digest length |
|----------|--------------|--|
| C_Digest | any | $\lceil t/8 \rceil$, where 0 < t < 512, and t > 384 |

6.27.3 General-length SHA-512/t-HMAC

The general-length SHA-512/t-HMAC mechanism, denoted **CKM_SHA512_T_HMAC_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in Section 6.20.3, except that it uses the HMAC construction based on the SHA-512/t hash function and length of the output should be in the range 0 – $\lceil t/8 \rceil$, where 0 < t < 512, and t > 384.

6.27.4 SHA-512/t-HMAC

The SHA-512/t-HMAC mechanism, denoted **CKM_SHA512_T_HMAC**, is a special case of the general-length SHA-512/t-HMAC mechanism.

It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the value of t in bits. The length in bytes of the desired output should be in the range of 0– $\lceil t/8 \rceil$, where 0 < t < 512, and t > 384.

10130 **6.27.5 SHA-512/t key derivation**

10131 The SHA-512/t key derivation, denoted **CKM_SHA512_T_KEY_DERIVATION**, is the same as the SHA-
10132 512 key derivation mechanism in section 6.24.5, except that it uses the SHA-512/t hash function and the
10133 relevant length is $\lceil t/8 \rceil$ bytes, where $0 < t < 512$, and $t \geq 384$.

10134 **6.27.6 SHA-512/t HMAC key generation**

10135 The SHA-512/t-HMAC key generation mechanism, denoted **CKM_SHA512_T_KEY_GEN**, is a key
10136 generation mechanism for NIST's SHA512/t-HMAC.

10137 It does not have a parameter.

10138 The mechanism generates SHA512/t-HMAC keys with a particular length in bytes, as specified in the
10139 **CKA_VALUE_LEN** attribute of the template for the key.

10140 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
10141 key. Other attributes supported by the SHA512/t-HMAC key type (specifically, the flags indicating which
10142 functions the key supports) may be specified in the template for the key, or else are assigned default
10143 initial values.

10144 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
10145 specify the supported range of **CKM_SHA512_T_HMAC** key sizes, in bytes.

10146

10147 **6.28 SHA3-224**

10148 *Table 157, SHA3-224 Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|-----------------------------|-------------------|---------------|----------------------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_SHA3_224 | | | | ✓ | | | |
| CKM_SHA3_224_HMAC | | ✓ | | | | | |
| CKM_SHA3_224_HMAC_GENERAL | | ✓ | | | | | |
| CKM_SHA3_224_KEY_DERIVATION | | | | | | | ✓ |
| CKM_SHA3_224_KEY_GEN | | | | | ✓ | | |

10149 **6.28.1 Definitions**

10150 Mechanisms:

10151 CKM_SHA3_224

10152 CKM_SHA3_224_HMAC

10153 CKM_SHA3_224_HMAC_GENERAL

10154 CKM_SHA3_224_KEY_DERIVATION

10155 CKM_SHA3_224_KEY_GEN

10156

10157 CKK_SHA3_224_HMAC

6.28.2 SHA3-224 digest

The SHA3-224 mechanism, denoted **CKM_SHA3_224**, is a mechanism for message digesting, following the Secure Hash 3 Algorithm with a 224-bit message digest defined in FIPS Pub 202.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 158, SHA3-224: Data Length

| Function | Input length | Digest length |
|----------|--------------|---------------|
| C_Digest | any | 28 |

6.28.3 General-length SHA3-224-HMAC

The general-length SHA3-224-HMAC mechanism, denoted **CKM_SHA3_224_HMAC_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in section 6.20.4 except that it uses the HMAC construction based on the SHA3-224 hash function and length of the output should be in the range 1-28. The keys it uses are generic secret keys and CKK_SHA3_224_HMAC. FIPS-198 compliant tokens may require the key length to be at least 14 bytes; that is, half the size of the SHA3-224 hash output.

It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired output. This length should be in the range 1-28 (the output size of SHA3-224 is 28 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 14 (half the maximum length). Signatures (MACs) produced by this mechanism shall be taken from the start of the full 28-byte HMAC output.

Table 159, General-length SHA3-224-HMAC: Key And Data Length

| Function | Key type | Data length | Signature length |
|----------|-------------------------------------|-------------|-------------------------------|
| C_Sign | generic secret or CKK_SHA3_224_HMAC | Any | 1-28, depending on parameters |
| C_Verify | generic secret or CKK_SHA3_224_HMAC | Any | 1-28, depending on parameters |

6.28.4 SHA3-224-HMAC

The SHA3-224-HMAC mechanism, denoted **CKM_SHA3_224_HMAC**, is a special case of the general-length SHA3-224-HMAC mechanism.

It has no parameter, and always produces an output of length 28.

6.28.5 SHA3-224 key derivation

SHA-224 key derivation, denoted **CKM_SHA3_224_KEY_DERIVATION**, is the same as the SHA-1 key derivation mechanism in Section 6.20.5 except that it uses the SHA3-224 hash function and the relevant length is 28 bytes.

6.28.6 SHA3-224 HMAC key generation

The SHA3-224-HMAC key generation mechanism, denoted **CKM_SHA3_224_KEY_GEN**, is a key generation mechanism for NIST's SHA3-224-HMAC.

It does not have a parameter.

The mechanism generates SHA3-224-HMAC keys with a particular length in bytes, as specified in the **CKA_VALUE_LEN** attribute of the template for the key.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the SHA3-224-HMAC key type (specifically, the flags indicating which

10193 functions the key supports) may be specified in the template for the key, or else are assigned default
 10194 initial values.
 10195 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 10196 specify the supported range of **CKM_SHA3_224_HMAC** key sizes, in bytes.

10197 6.29 SHA3-256

10198 *Table 160, SHA3-256 Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|-----------------------------|-------------------|---------------|----------------------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_SHA3_256 | | | | ✓ | | | |
| CKM_SHA3_256_HMAC_GENERAL | | ✓ | | | | | |
| CKM_SHA3_256_HMAC | | ✓ | | | | | |
| CKM_SHA3_256_KEY_DERIVATION | | | | | | | ✓ |
| CKM_SHA3_256_KEY_GEN | | | | | ✓ | | |

10199 6.29.1 Definitions

10200 Mechanisms:
 10201 CKM_SHA3_256
 10202 CKM_SHA3_256_HMAC
 10203 CKM_SHA3_256_HMAC_GENERAL
 10204 CKM_SHA3_256_KEY_DERIVATION
 10205 CKM_SHA3_256_KEY_GEN
 10206
 10207 CKK_SHA3_256_HMAC

10208 6.29.2 SHA3-256 digest

10209 The SHA3-256 mechanism, denoted **CKM_SHA3_256**, is a mechanism for message digesting, following
 10210 the Secure Hash 3 Algorithm with a 256-bit message digest defined in FIPS PUB 202.
 10211 It does not have a parameter.
 10212 Constraints on the length of input and output data are summarized in the following table. For single-part
 10213 digesting, the data and the digest may begin at the same location in memory.

10214 *Table 161, SHA3-256: Data Length*

| Function | Input length | Digest length |
|----------|--------------|---------------|
| C_Digest | any | 32 |

10215 6.29.3 General-length SHA3-256-HMAC

10216 The general-length SHA3-256-HMAC mechanism, denoted **CKM_SHA3_256_HMAC_GENERAL**, is the
 10217 same as the general-length SHA-1-HMAC mechanism in Section 6.20.4, except that it uses the HMAC
 10218 construction based on the SHA3-256 hash function and length of the output should be in the range 1-32.
 10219 The keys it uses are generic secret keys and CKK_SHA3_256_HMAC. FIPS-198 compliant tokens may
 10220 require the key length to be at least 16 bytes; that is, half the size of the SHA3-256 hash output.

10221 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired
 10222 output. This length should be in the range 1-32 (the output size of SHA3-256 is 32 bytes). FIPS-198
 10223 compliant tokens may constrain the output length to be at least 4 or 16 (half the maximum length).
 10224 Signatures (MACs) produced by this mechanism shall be taken from the start of the full 32-byte HMAC
 10225 output.

10226 *Table 162, General-length SHA3-256-HMAC: Key And Data Length*

| Function | Key type | Data length | Signature length |
|-----------------|--|--------------------|-------------------------------|
| C_Sign | generic secret or CKK_SHA3_256_HMAC | Any | 1-32, depending on parameters |
| C_Verify | generic secret or CKK_SHA3_256_HMAC | Any | 1-32, depending on parameters |

10227 **6.29.4 SHA3-256-HMAC**

10228 The SHA-256-HMAC mechanism, denoted **CKM_SHA3_256_HMAC**, is a special case of the general-
 10229 length SHA-256-HMAC mechanism.

10230 It has no parameter, and always produces an output of length 32.

10231 **6.29.5 SHA3-256 key derivation**

10232 SHA-256 key derivation, denoted **CKM_SHA3_256_KEY_DERIVATION**, is the same as the SHA-1 key
 10233 derivation mechanism in Section 6.20.5, except that it uses the SHA3-256 hash function and the relevant
 10234 length is 32 bytes.

10235 **6.29.6 SHA3-256 HMAC key generation**

10236 The SHA3-256-HMAC key generation mechanism, denoted **CKM_SHA3_256_KEY_GEN**, is a key
 10237 generation mechanism for NIST's SHA3-256-HMAC.

10238 It does not have a parameter.

10239 The mechanism generates SHA3-256-HMAC keys with a particular length in bytes, as specified in the
 10240 **CKA_VALUE_LEN** attribute of the template for the key.

10241 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
 10242 key. Other attributes supported by the SHA3-256-HMAC key type (specifically, the flags indicating which
 10243 functions the key supports) may be specified in the template for the key, or else are assigned default
 10244 initial values.

10245 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 10246 specify the supported range of **CKM_SHA3_256_HMAC** key sizes, in bytes.

10247

10248 **6.30 SHA3-384**

10249 *Table 163, SHA3-384 Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|-----------------------------|------------------------------|--------------------------|--------------------------------|---------------|--------------------------|--------------------------|---------------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR¹ | Digest | Gen. Key/Key Pair | Wrap & Unwrap | Derive |
| CKM_SHA3_384 | | | | ✓ | | | |
| CKM_SHA3_384_HMAC_GENERAL | | ✓ | | | | | |
| CKM_SHA3_384_HMAC | | ✓ | | | | | |
| CKM_SHA3_384_KEY_DERIVATION | | | | | | | ✓ |

| Mechanism | Functions | | | | | | |
|----------------------|-------------------|---------------|----------------------|--------|-------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/Key Pair | Wrap & Unwrap | Derive |
| CKM_SHA3_384_KEY_GEN | | | | ✓ | | | |

6.30.1 Definitions

10250 CKM_SHA3_384
 10251 CKM_SHA3_384_HMAC
 10252 CKM_SHA3_384_HMAC_GENERAL
 10253 CKM_SHA3_384_KEY_DERIVATION
 10254 CKM_SHA3_384_KEY_GEN
 10255 CKK_SHA3_384_HMAC
 10256
 10257 CKK_SHA3_384_HMAC

6.30.2 SHA3-384 digest

10259 The SHA3-384 mechanism, denoted **CKM_SHA3_384**, is a mechanism for message digesting, following
 10260 the Secure Hash 3 Algorithm with a 384-bit message digest defined in FIPS PUB 202.

10261 It does not have a parameter.

10262 Constraints on the length of input and output data are summarized in the following table. For single-part
 10263 digesting, the data and the digest may begin at the same location in memory.

10264 *Table 164, SHA3-384: Data Length*

| Function | Input length | Digest length |
|----------|--------------|---------------|
| C_Digest | any | 48 |

6.30.3 General-length SHA3-384-HMAC

10266 The general-length SHA3-384-HMAC mechanism, denoted **CKM_SHA3_384_HMAC_GENERAL**, is the
 10267 same as the general-length SHA-1-HMAC mechanism in Section 6.20.4, except that it uses the HMAC
 10268 construction based on the SHA-384 hash function and length of the output should be in the range 1-
 10269 48. The keys it uses are generic secret keys and CKK_SHA3_384_HMAC. FIPS-198 compliant tokens
 10270 may require the key length to be at least 24 bytes; that is, half the size of the SHA3-384 hash output.

10271

10272 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired
 10273 output. This length should be in the range 1-48 (the output size of SHA3-384 is 48 bytes). FIPS-198
 10274 compliant tokens may constrain the output length to be at least 4 or 24 (half the maximum length).
 10275 Signatures (MACs) produced by this mechanism shall be taken from the start of the full 48-byte HMAC
 10276 output.

10277 *Table 165, General-length SHA3-384-HMAC: Key And Data Length*

| Function | Key type | Data length | Signature length |
|----------|-------------------------------------|-------------|-------------------------------|
| C_Sign | generic secret or CKK_SHA3_384_HMAC | Any | 1-48, depending on parameters |
| C_Verify | generic secret or CKK_SHA3_384_HMAC | Any | 1-48, depending on parameters |

10278

6.30.4 SHA3-384-HMAC

The SHA3-384-HMAC mechanism, denoted **CKM_SHA3_384_HMAC**, is a special case of the general-length SHA3-384-HMAC mechanism.

It has no parameter, and always produces an output of length 48.

6.30.5 SHA3-384 key derivation

SHA3-384 key derivation, denoted **CKM_SHA3_384_KEY_DERIVATION**, is the same as the SHA-1 key derivation mechanism in Section 6.20.5, except that it uses the SHA-384 hash function and the relevant length is 48 bytes.

6.30.6 SHA3-384 HMAC key generation

The SHA3-384-HMAC key generation mechanism, denoted **CKM_SHA3_384_KEY_GEN**, is a key generation mechanism for NIST's SHA3-384-HMAC.

It does not have a parameter.

The mechanism generates SHA3-384-HMAC keys with a particular length in bytes, as specified in the **CKA_VALUE_LEN** attribute of the template for the key.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the SHA3-384-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of **CKM_SHA3_384_HMAC** key sizes, in bytes.

6.31 SHA3-512

Table 166, SHA-512 Mechanisms vs. Functions

| Mechanism | Functions | | | | | | |
|-----------------------------|-------------------|---------------|----------------------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_SHA3_512 | | | | ✓ | | | |
| CKM_SHA3_512_HMAC_GENERAL | | ✓ | | | | | |
| CKM_SHA3_512_HMAC | | ✓ | | | | | |
| CKM_SHA3_512_KEY_DERIVATION | | | | | | | ✓ |
| CKM_SHA3_512_KEY_GEN | | | | ✓ | | | |

6.31.1 Definitions

CKM_SHA3_512

CKM_SHA3_512_HMAC

CKM_SHA3_512_HMAC_GENERAL

CKM_SHA3_512_KEY_DERIVATION

CKM_SHA3_512_KEY_GEN

CKK_SHA3_512_HMAC

6.31.2 SHA3-512 digest

The SHA3-512 mechanism, denoted **CKM_SHA3_512**, is a mechanism for message digesting, following the Secure Hash 3 Algorithm with a 512-bit message digest defined in FIPS PUB 202.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 167, SHA3-512: Data Length

| Function | Input length | Digest length |
|----------|--------------|---------------|
| C_Digest | any | 64 |

6.31.3 General-length SHA3-512-HMAC

The general-length SHA3-512-HMAC mechanism, denoted **CKM_SHA3_512_HMAC_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in Section 6.20.4, except that it uses the HMAC construction based on the SHA3-512 hash function and length of the output should be in the range 1-64. The keys it uses are generic secret keys and CKK_SHA3_512_HMAC. FIPS-198 compliant tokens may require the key length to be at least 32 bytes; that is, half the size of the SHA3-512 hash output.

It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired output. This length should be in the range 1-64 (the output size of SHA3-512 is 64 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 32 (half the maximum length). Signatures (MACs) produced by this mechanism shall be taken from the start of the full 64-byte HMAC output.

Table 168, General-length SHA3-512-HMAC: Key And Data Length

| Function | Key type | Data length | Signature length |
|----------|-------------------------------------|-------------|-------------------------------|
| C_Sign | generic secret or CKK_SHA3_512_HMAC | Any | 1-64, depending on parameters |
| C_Verify | generic secret or CKK_SHA3_512_HMAC | Any | 1-64, depending on parameters |

6.31.4 SHA3-512-HMAC

The SHA3-512-HMAC mechanism, denoted **CKM_SHA3_512_HMAC**, is a special case of the general-length SHA3-512-HMAC mechanism.

It has no parameter, and always produces an output of length 64.

6.31.5 SHA3-512 key derivation

SHA3-512 key derivation, denoted **CKM_SHA3_512_KEY_DERIVATION**, is the same as the SHA-1 key derivation mechanism in Section 6.20.5, except that it uses the SHA-512 hash function and the relevant length is 64 bytes.

6.31.6 SHA3-512 HMAC key generation

The SHA3-512-HMAC key generation mechanism, denoted **CKM_SHA3_512_KEY_GEN**, is a key generation mechanism for NIST's SHA3-512-HMAC.

It does not have a parameter.

The mechanism generates SHA3-512-HMAC keys with a particular length in bytes, as specified in the **CKA_VALUE_LEN** attribute of the template for the key.

10344 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
10345 key. Other attributes supported by the SHA3-512-HMAC key type (specifically, the flags indicating which
10346 functions the key supports) may be specified in the template for the key, or else are assigned default
10347 initial values.

10348 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
10349 specify the supported range of **CKM_SHA3_512_HMAC** key sizes, in bytes.

10350 **6.32 SHAKE**

10351 *Table 169, SHA-512 Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|------------------------------|-------------------|---------------|----------------------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_SHAKE_128_KEY_DERIVATION | | | | | | | ✓ |
| CKM_SHAKE_256_KEY_DERIVATION | | | | | | | ✓ |

10352 **6.32.1 Definitions**

10353 CKM_SHAKE_128_KEY_DERIVATION
10354 CKM_SHAKE_256_KEY_DERIVATION

10355 **6.32.2 SHAKE Key Derivation**

10356 SHAKE-128 and SHAKE-256 key derivation, denoted **CKM_SHAKE_128_KEY_DERIVATION** and
10357 **CKM_SHAKE_256_KEY_DERIVATION**, implements the SHAKE expansion function defined in FIPS 202
10358 on the input key.

- 10359 • If no length or key type is provided in the template a **CKR_TEMPLATE_INCOMPLETE** error is
10360 generated.
- 10361 • If no key type is provided in the template, but a length is, then the key produced by this mechanism
10362 shall be a generic secret key of the specified length.
- 10363 • If no length was provided in the template, but a key type is, then that key type must have a well-
10364 defined length. If it does, then the key produced by this mechanism shall be of the type specified in
10365 the template. If it doesn't, an error shall be returned.
- 10366 • If both a key type and a length are provided in the template, the length must be compatible with that
10367 key type. The key produced by this mechanism shall be of the specified type and length.

10368 If a DES, DES2, or CDMF key is derived with this mechanism, the parity bits of the key shall be set
10369 properly.

10370 This mechanism has the following rules about key sensitivity and extractability:

- 10371 • The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both
10372 be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some
10373 default value.
- 10374 • If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key
10375 shall as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the
10376 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
10377 **CKA_SENSITIVE** attribute.
 - 10378 • Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then
10379 the derived key shall, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
10380 CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
10381 value from its **CKA_EXTRACTABLE** attribute.

10382 **6.33 BLAKE2B-160**

10383 *Table 170, BLAKE2B-160 Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|------------------------------|-------------------|---------------|----------------------|--------|-------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/Key Pair | Wrap & Unwrap | Derive |
| CKM_BLAKE2B_160 | | | | ✓ | | | |
| CKM_BLAKE2B_160_HMAC | | ✓ | | | | | |
| CKM_BLAKE2B_160_HMAC_GENERAL | | ✓ | | | | | |
| CKM_BLAKE2B_160_KEY_DERIVE | | | | | | | ✓ |
| CKM_BLAKE2B_160_KEY_GEN | | | | | ✓ | | |

10384 **6.33.1 Definitions**

10385 Mechanisms:

10386 CKM_BLAKE2B_160

10387 CKM_BLAKE2B_160_HMAC

10388 CKM_BLAKE2B_160_HMAC_GENERAL

10389 CKM_BLAKE2B_160_KEY_DERIVE

10390 CKM_BLAKE2B_160_KEY_GEN

10391 CKK_BLAKE2B_160_HMAC

10392 **6.33.2 BLAKE2B-160 digest**

10393 The BLAKE2B-160 mechanism, denoted **CKM_BLAKE2B_160**, is a mechanism for message digesting, following the Blake2b Algorithm with a 160-bit message digest without a key as defined in [RFC 7693](#).

10395 It does not have a parameter.

10396 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.

10398 *Table 171, BLAKE2B-160: Data Length*

| Function | Input length | Digest length |
|----------|--------------|---------------|
| C_Digest | any | 20 |

10399 **6.33.3 General-length BLAKE2B-160-HMAC**

10400 The general-length BLAKE2B-160-HMAC mechanism, denoted
 10401 **CKM_BLAKE2B_160_HMAC_GENERAL**, is the keyed variant of BLAKE2b-160 and length of the output
 10402 should be in the range 1-20. The keys it uses are generic secret keys and CKK_BLAKE2B_160_HMAC.

10403 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired
 10404 output. This length should be in the range 1-20 (the output size of BLAKE2B-160 is 20 bytes). Signatures
 10405 (MACs) produced by this mechanism shall be taken from the start of the full 20-byte HMAC output.

10406 *Table 172, General-length BLAKE2B-160-HMAC: Key And Data Length*

| Function | Key type | Data length | Signature length |
|----------|---|-------------|-------------------------------|
| C_Sign | generic secret or CKK_BLAKE2B_160_H MAC | Any | 1-20, depending on parameters |
| C_Verify | generic secret or CKK_BLAKE2B_160_H MAC | Any | 1-20, depending on parameters |

10407 **6.33.4 BLAKE2B-160-HMAC**

10408 The BLAKE2B-160-HMAC mechanism, denoted **CKM_BLAKE2B_160_HMAC**, is a special case of the
10409 general-length BLAKE2B-160-HMAC mechanism.

10410 It has no parameter, and always produces an output of length 20.

10411 **6.33.5 BLAKE2B-160 key derivation**

10412 BLAKE2B-160 key derivation, denoted **CKM_BLAKE2B_160_KEY_DERIVE**, is the same as the SHA-1
10413 key derivation mechanism in Section 6.20.5 except that it uses the BLAKE2B-160 hash function and the
10414 relevant length is 20 bytes.

10415 **6.33.6 BLAKE2B-160 HMAC key generation**

10416 The BLAKE2B-160-HMAC key generation mechanism, denoted **CKM_BLAKE2B_160_KEY_GEN**, is a
10417 key generation mechanism for BLAKE2B-160-HMAC.

10418 It does not have a parameter.

10419 The mechanism generates BLAKE2B-160-HMAC keys with a particular length in bytes, as specified in the
10420 **CKA_VALUE_LEN** attribute of the template for the key.

10421 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
10422 key. Other attributes supported by the BLAKE2B-160-HMAC key type (specifically, the flags indicating
10423 which functions the key supports) may be specified in the template for the key, or else are assigned
10424 default initial values.

10425 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
10426 specify the supported range of **CKM_BLAKE2B_160_HMAC** key sizes, in bytes.

10427 **6.34 BLAKE2B-256**

10428 *Table 173, BLAKE2B-256 Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|------------------------------|-------------------|---------------|----------------------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_BLAKE2B_256 | | | | ✓ | | | |
| CKM_BLAKE2B_256_HMAC_GENERAL | | ✓ | | | | | |
| CKM_BLAKE2B_256_HMAC | | ✓ | | | | | |
| CKM_BLAKE2B_256_KEY_DERIVE | | | | | | | ✓ |
| CKM_BLAKE2B_256_KEY_GEN | | | | | ✓ | | |

6.34.1 Definitions

Mechanisms:

- 10431 CKM_BLAKE2B_256
- 10432 CKM_BLAKE2B_256_HMAC
- 10433 CKM_BLAKE2B_256_HMAC_GENERAL
- 10434 CKM_BLAKE2B_256_KEY_DERIVE
- 10435 CKM_BLAKE2B_256_KEY_GEN
- 10436 CKK_BLAKE2B_256_HMAC

6.34.2 BLAKE2B-256 digest

10438 The BLAKE2B-256 mechanism, denoted **CKM_BLAKE2B_256**, is a mechanism for message digesting, following the Blake2b Algorithm with a 256-bit message digest without a key as defined in RFC 7693.

10440 It does not have a parameter.

10441 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.

10443 *Table 174, BLAKE2B-256: Data Length*

| Function | Input length | Digest length |
|----------|--------------|---------------|
| C_Digest | any | 32 |

6.34.3 General-length BLAKE2B-256-HMAC

10445 The general-length BLAKE2B-256-HMAC mechanism, denoted **CKM_BLAKE2B_256_HMAC_GENERAL**, is the keyed variant of Blake2b-256 and length of the output should be in the range 1-32. The keys it uses are generic secret keys and CKK_BLAKE2B_256_HMAC.

10448 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired output. This length should be in the range 1-32 (the output size of BLAKE2B-256 is 32 bytes). Signatures (MACs) produced by this mechanism shall be taken from the start of the full 32-byte HMAC output.

10451 *Table 175, General-length BLAKE2B-256-HMAC: Key And Data Length*

| Function | Key type | Data length | Signature length |
|----------|--|-------------|-------------------------------|
| C_Sign | generic secret or CKK_BLAKE2B_256_HMAC | Any | 1-32, depending on parameters |
| C_Verify | generic secret or CKK_BLAKE2B_256_HMAC | Any | 1-32, depending on parameters |

6.34.4 BLAKE2B-256-HMAC

The BLAKE2B-256-HMAC mechanism, denoted **CKM_BLAKE2B_256_HMAC**, is a special case of the general-length BLAKE2B-256-HMAC mechanism in Section 6.34.3.

It has no parameter, and always produces an output of length 32.

6.34.5 BLAKE2B-256 key derivation

BLAKE2B-256 key derivation, denoted **CKM_BLAKE2B_256_KEY_DERIVE**, is the same as the SHA-1 key derivation mechanism in Section 6.20.5, except that it uses the BLAKE2B-256 hash function and the relevant length is 32 bytes.

6.34.6 BLAKE2B-256 HMAC key generation

The BLAKE2B-256-HMAC key generation mechanism, denoted **CKM_BLAKE2B_256_KEY_GEN**, is a key generation mechanism for BLAKE2B-256-HMAC.

It does not have a parameter.

The mechanism generates BLAKE2B-256-HMAC keys with a particular length in bytes, as specified in the **CKA_VALUE_LEN** attribute of the template for the key.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the BLAKE2B-256-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of **CKM_BLAKE2B_256_HMAC** key sizes, in bytes.

6.35 BLAKE2B-384

Table 176, BLAKE2B-384 Mechanisms vs. Functions

| Mechanism | Functions | | | | | | |
|------------------------------|-------------------|---------------|----------------------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_BLAKE2B_384 | | | | ✓ | | | |
| CKM_BLAKE2B_384_HMAC_GENERAL | | ✓ | | | | | |
| CKM_BLAKE2B_384_HMAC | | ✓ | | | | | |
| CKM_BLAKE2B_384_KEY_DERIVE | | | | | | | ✓ |
| CKM_BLAKE2B_384_KEY_GEN | | | | ✓ | | | |

10474 6.35.1 Definitions

10475 CKM_BLAKE2B_384
 10476 CKM_BLAKE2B_384_HMAC
 10477 CKM_BLAKE2B_384_HMAC_GENERAL
 10478 CKM_BLAKE2B_384_KEY_DERIVE
 10479 CKM_BLAKE2B_384_KEY_GEN
 10480 CKK_BLAKE2B_384_HMAC

10481 6.35.2 BLAKE2B-384 digest

10482 The BLAKE2B-384 mechanism, denoted **CKM_BLAKE2B_384**, is a mechanism for message digesting,
 10483 following the Blake2b Algorithm with a 384-bit message digest without a key as defined in RFC 7693.

10484 It does not have a parameter.

10485 Constraints on the length of input and output data are summarized in the following table. For single-part
 10486 digesting, the data and the digest may begin at the same location in memory.

10487 *Table 177, BLAKE2B-384: Data Length*

| Function | Input length | Digest length |
|----------|--------------|---------------|
| C_Digest | any | 48 |

10488 6.35.3 General-length BLAKE2B-384-HMAC

10489 The general-length BLAKE2B-384-HMAC mechanism, denoted
 10490 **CKM_BLAKE2B_384_HMAC_GENERAL**, is the keyed variant of the BLAKE2B-384 hash function and
 10491 length of the output should be in the range 1-48. The keys it uses are generic secret keys and
 10492 CKK_BLAKE2B_384_HMAC.

10493

10494 It has a parameter, a CK_MAC_GENERAL_PARAMS, which holds the length in bytes of the desired output.
 10495 This length should be in the range 1-48 (the output size of BLAKE2B-384 is 48 bytes). Signatures
 10496 (MACs) produced by this mechanism shall be taken from the start of the full 48-byte HMAC output.

10497 *Table 178, General-length BLAKE2B-384-HMAC: Key And Data Length*

| Function | Key type | Data length | Signature length |
|-----------------|---|--------------------|-------------------------------|
| C_Sign | generic secret or CKK_BLAKE2B_384_H MAC | Any | 1-48, depending on parameters |
| C_Verify | generic secret or CKK_BLAKE2B_384_H MAC | Any | 1-48, depending on parameters |

10498

10499 **6.35.4 BLAKE2B-384-HMAC**

10500 The BLAKE2B-384-HMAC mechanism, denoted **CKM_BLAKE2B_384_HMAC**, is a special case of the
10501 general-length BLAKE2B-384-HMAC mechanism.

10502 It has no parameter, and always produces an output of length 48.

10503 **6.35.5 BLAKE2B-384 key derivation**

10504 BLAKE2B-384 key derivation, denoted **CKM_BLAKE2B_384_KEY_DERIVE**, is the same as the SHA-1
10505 key derivation mechanism in Section 6.20.5, except that it uses the BLAKE2B-384 hash function and the
10506 relevant length is 48 bytes.

10507 **6.35.6 BLAKE2B-384 HMAC key generation**

10508 The BLAKE2B-384-HMAC key generation mechanism, denoted **CKM_BLAKE2B_384_KEY_GEN**, is a
10509 key generation mechanism for NIST's BLAKE2B-384-HMAC.

10510 It does not have a parameter.

10511 The mechanism generates BLAKE2B-384-HMAC keys with a particular length in bytes, as specified in the
10512 **CKA_VALUE_LEN** attribute of the template for the key.

10513 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
10514 key. Other attributes supported by the BLAKE2B-384-HMAC key type (specifically, the flags indicating
10515 which functions the key supports) may be specified in the template for the key, or else are assigned
10516 default initial values.

10517 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
10518 specify the supported range of **CKM_BLAKE2B_384_HMAC** key sizes, in bytes.

10519 **6.36 BLAKE2B-512**

10520 *Table 179, SHA-512 Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|-------------------------------|-------------------|---------------|----------------------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ₁ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_BLAKE2B_512 | | | | ✓ | | | |
| CKM_BLAKE2B_512_HMAC_GENE_RAL | | ✓ | | | | | |
| CKM_BLAKE2B_512_HMAC | | ✓ | | | | | |
| CKM_BLAKE2B_512_KEY_DERIVE | | | | | | | ✓ |
| CKM_BLAKE2B_512_KEY_GEN | | | | ✓ | | | |

10521 **6.36.1 Definitions**

10522 CKM_BLAKE2B_512
 10523 CKM_BLAKE2B_512_HMAC
 10524 CKM_BLAKE2B_512_HMAC_GENERAL
 10525 CKM_BLAKE2B_512_KEY_DERIVE
 10526 CKM_BLAKE2B_512_KEY_GEN
 10527 CKK_BLAKE2B_512_HMAC

10528 **6.36.2 BLAKE2B-512 digest**

10529 The BLAKE2B-512 mechanism, denoted **CKM_BLAKE2B_512**, is a mechanism for message digesting,
 10530 following the Blake2b Algorithm with a 512-bit message digest defined in RFC 7693.

10531 It does not have a parameter.

10532 Constraints on the length of input and output data are summarized in the following table. For single-part
 10533 digesting, the data and the digest may begin at the same location in memory.

10534 *Table 180, BLAKE2B-512: Data Length*

| Function | Input length | Digest length |
|----------|--------------|---------------|
| C_Digest | any | 64 |

10535 **6.36.3 General-length BLAKE2B-512-HMAC**

10536 The general-length BLAKE2B-512-HMAC mechanism, denoted
 10537 **CKM_BLAKE2B_512_HMAC_GENERAL**, is the keyed variant of the BLAKE2B-512 hash function and
 10538 length of the output should be in the range 1-64. The keys it uses are generic secret keys and
 10539 CKK_BLAKE2B_512_HMAC.

10540

10541 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired
 10542 output. This length should be in the range 1-64 (the output size of BLAKE2B-512 is 64 bytes). Signatures
 10543 (MACs) produced by this mechanism shall be taken from the start of the full 64-byte HMAC output.

10544 *Table 181, General-length BLAKE2B-512-HMAC: Key And Data Length*

| Function | Key type | Data length | Signature length |
|----------|--|-------------|-------------------------------|
| C_Sign | generic secret or CKK_BLAKE2B_512_HMAC | Any | 1-64, depending on parameters |
| C_Verify | generic secret or CKK_BLAKE2B_512_HMAC | Any | 1-64, depending on parameters |

10545

6.36.4 BLAKE2B-512-HMAC

10546 The BLAKE2B-512-HMAC mechanism, denoted **CKM_BLAKE2B_512_HMAC**, is a special case of the general-length BLAKE2B-512-HMAC mechanism.

10547 It has no parameter, and always produces an output of length 64.

6.36.5 BLAKE2B-512 key derivation

10548 BLAKE2B-512 key derivation, denoted **CKM_BLAKE2B_512_KEY_DERIVE**, is the same as the SHA-1 key derivation mechanism in Section 6.20.5, except that it uses the BLAKE2B-512 hash function and the relevant length is 64 bytes.

6.36.6 BLAKE2B-512 HMAC key generation

10549 The BLAKE2B-512-HMAC key generation mechanism, denoted **CKM_BLAKE2B_512_KEY_GEN**, is a key generation mechanism for NIST's BLAKE2B-512-HMAC.

10550 It does not have a parameter.

10551 The mechanism generates BLAKE2B-512-HMAC keys with a particular length in bytes, as specified in the **CKA_VALUE_LEN** attribute of the template for the key.

10552 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the BLAKE2B-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.

10553 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of **CKM_BLAKE2B_512_HMAC** key sizes, in bytes.

10554

6.37 PKCS #5 and PKCS #5-style password-based encryption (PBE)

10555 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].

10556 *Table 182, PKCS 5 Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|-----------------------------|-------------------|---------------|----------------------|--------|---------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen . Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_PBE_SHA1_DES3_EDE_CBC | | | | | ✓ | | |
| CKM_PBE_SHA1_DES2_EDE_CBC | | | | | ✓ | | |
| CKM_PBA_SHA1_WITH_SHA1_HMAC | | | | | ✓ | | |

| Mechanism | Functions | | | | | | |
|-----------------|-------------------|---------------|---------|--------|---------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR | Digest | Gen . Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_PKCS5_PBKD2 | | | | | ✓ | | |

10571 6.37.1 Definitions

10572 Mechanisms:

10573 CKM_PBE_SHA1_DES3_EDE_CBC
 10574 CKM_PBE_SHA1_DES2_EDE_CBC
 10575 CKM_PKCS5_PBKD2
 10576 CKM_PBA_SHA1_WITH_SHA1_HMAC

10577 6.37.2 Password-based encryption/authentication mechanism parameters

10578 ♦ CK_PBE_PARAMS; CK_PBE_PARAMS_PTR

10579 **CK_PBE_PARAMS** is a structure which provides all of the necessary information required by the
 10580 CKM_PBE mechanisms (see [PKCS #5] and [PKCS #12] for information on the PBE generation
 10581 mechanisms) and the CKM_PBA_SHA1_WITH_SHA1_HMAC mechanism. It is defined as follows:

```
10582     typedef struct CK_PBE_PARAMS {
10583         CK_BYTE_PTR          pInitVector;
10584         CK_UTF8CHAR_PTR      pPassword;
10585         CK ULONG              ulPasswordLen;
10586         CK_BYTE_PTR          pSalt;
10587         CK ULONG              ulSaltLen;
10588         CK ULONG              ulIteration;
10589     } CK_PBE_PARAMS;
```

10590
 10591 The fields of the structure have the following meanings:

| | |
|---------------------|--|
| 10592 pInitVector | pointer to the location that receives the 8-byte initialization vector (IV), if an IV is required; |
| 10594 pPassword | points to the password to be used in the PBE key generation; |
| 10595 ulPasswordLen | length in bytes of the password information; |
| 10596 pSalt | points to the salt to be used in the PBE key generation; |
| 10597 ulSaltLen | length in bytes of the salt information; |
| 10598 ullIteration | number of iterations required for the generation. |

10599 **CK_PBE_PARAMS_PTR** is a pointer to a **CK_PBE_PARAMS**.

10600 6.37.3 PKCS #5 PBKDF2 key generation mechanism parameters

- 10601 ◆ **CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE;**
 10602 **CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE_PTR**

10603 **CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE** is used to indicate the Pseudo-Random
 10604 Function (PRF) used to generate key bits using PKCS #5 PBKDF2. It is defined as follows:

10605 `typedef CK ULONG CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE;`

10606

10607 The following PRFs are defined in PKCS #5 v2.1. The following table lists the defined functions.

10608 *Table 183, PKCS #5 PBKDF2 Key Generation: Pseudo-random functions*

| PRF Identifier | Value | Parameter Type |
|---------------------------------|--------------|---|
| CKP_PKCS5_PBKD2_HMAC_SHA1 | 0x00000001UL | No Parameter. <i>pPrfData</i> must be NULL and <i>ulPrfDataLen</i> must be zero. |
| CKP_PKCS5_PBKD2_HMAC_GOSTR3411 | 0x00000002UL | This PRF uses GOST R34.11-94 hash to produce secret key value. <i>pPrfData</i> should point to DER-encoded OID, indicating GOSTR34.11-94 parameters. <i>ulPrfDataLen</i> holds encoded OID length in bytes. If <i>pPrfData</i> is set to NULL_PTR, then <i>id-GostR3411-94-CryptoProParamSet</i> parameters will be used (RFC 4357, 11.2), and <i>ulPrfDataLen</i> must be 0. |
| CKP_PKCS5_PBKD2_HMAC_SHA224 | 0x00000003UL | No Parameter. <i>pPrfData</i> must be NULL and <i>ulPrfDataLen</i> must be zero. |
| CKP_PKCS5_PBKD2_HMAC_SHA256 | 0x00000004UL | No Parameter. <i>pPrfData</i> must be NULL and <i>ulPrfDataLen</i> must be zero. |
| CKP_PKCS5_PBKD2_HMAC_SHA384 | 0x00000005UL | No Parameter. <i>pPrfData</i> must be NULL and <i>ulPrfDataLen</i> must be zero. |
| CKP_PKCS5_PBKD2_HMAC_SHA512 | 0x00000006UL | No Parameter. <i>pPrfData</i> must be NULL and <i>ulPrfDataLen</i> must be zero. |
| CKP_PKCS5_PBKD2_HMAC_SHA512_224 | 0x00000007UL | No Parameter. <i>pPrfData</i> must be NULL and <i>ulPrfDataLen</i> must be zero. |
| CKP_PKCS5_PBKD2_HMAC_SHA512_256 | 0x00000008UL | No Parameter. <i>pPrfData</i> must be NULL and <i>ulPrfDataLen</i> must be zero. |

10609 **CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE_PTR** is a pointer to a
 10610 **CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE**.

10611

- 10612 ◆ **CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE;**
 10613 **CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE_PTR**

10614 **CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE** is used to indicate the source of the salt value when
 10615 deriving a key using PKCS #5 PBKDF2. It is defined as follows:

10616 `typedef CK ULONG CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE;`

10617
 10618 The following salt value sources are defined in PKCS #5 v2.1. The following table lists the defined
 10619 sources along with the corresponding data type for the *pSaltSourceData* field in the
 10620 **CK_PKCS5_PBKD2_PARAMS2** structure defined below.

10621 *Table 184, PKCS #5 PBKDF2 Key Generation: Salt sources*

| Source Identifier | Value | Data Type |
|--------------------|------------|--|
| CKZ_SALT_SPECIFIED | 0x00000001 | Array of CK_BYTE containing the value of the salt value. |

10622 **CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE_PTR** is a pointer to a
 10623 **CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE**.

- 10624 ◆ **CK_PKCS5_PBKD2_PARAMS2; CK_PKCS5_PBKD2_PARAMS2_PTR**

10625 **CK_PKCS5_PBKD2_PARAMS2** is a structure that provides the parameters to the
 10626 **CKM_PKCS5_PBKD2** mechanism. The structure is defined as follows:

```
10627     typedef struct CK_PKCS5_PBKD2_PARAMS2 {
10628         CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE      saltSource;
10629         CK_VOID_PTR                           pSaltSourceData;
10630         CK ULONG                             ulSaltSourceDataLen;
10631         CK ULONG                             iterations;
10632         CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE prf;
10633         CK_VOID_PTR                           pPrfData;
10634         CK ULONG                             ulPrfDataLen;
10635         CK_UTF8CHAR_PTR                      pPassword;
10636         CK ULONG                             ulPasswordLen;
10637     } CK_PKCS5_PBKD2_PARAMS2;
```

10639 The fields of the structure have the following meanings:

| | | |
|-------|---------------------|---|
| 10640 | saltSource | source of the salt value |
| 10641 | pSaltSourceData | data used as the input for the salt source |
| 10642 | ulSaltSourceDataLen | length of the salt source input |
| 10643 | iterations | number of iterations to perform when generating each block of random data |
| 10644 | prf | pseudo-random function used to generate the key |
| 10645 | pPrfData | data used as the input for PRF in addition to the salt value |
| 10646 | ulPrfDataLen | length of the input data for the PRF |
| 10647 | pPassword | points to the password to be used in the PBE key generation |
| 10648 | ulPasswordLen | length in bytes of the password information |

10649 **CK_PKCS5_PBKD2_PARAMS2_PTR** is a pointer to a **CK_PKCS5_PBKD2_PARAMS2**.

10651 6.37.4 PKCS #5 PBKD2 key generation

10652 PKCS #5 PBKDF2 key generation, denoted **CKM_PKCS5_PBKD2**, is a mechanism used for generating
10653 a secret key from a password and a salt value. This functionality is defined in PKCS#5 as PBKDF2.

10654 It has a parameter, a **CK_PKCS5_PBKD2_PARAMS2** structure. The parameter specifies the salt value
10655 source, pseudo-random function, and iteration count used to generate the new key.

10656 Since this mechanism can be used to generate any type of secret key, new key templates must contain
10657 the **CKA_KEY_TYPE** and **CKA_VALUE_LEN** attributes. If the key type has a fixed length the
10658 **CKA_VALUE_LEN** attribute may be omitted.

10659 6.38 PKCS #12 password-based encryption/authentication 10660 mechanisms

10661 The mechanisms in this section are for generating keys and IVs for performing password-based
10662 encryption or authentication. The method used to generate keys and IVs is based on a method that was
10663 specified in [PKCS #12].

10664 We specify here a general method for producing various types of pseudo-random bits from a password,
10665 p ; a string of salt bits, s ; and an iteration count, c . The “type” of pseudo-random bits to be produced is
10666 identified by an identification byte, ID , the meaning of which will be discussed later.

10667 Let H be a hash function built around a compression function $f: \mathbb{Z}_2^u \times \mathbb{Z}_2^v \rightarrow \mathbb{Z}_2^u$ (that is, H has a chaining
10668 variable and output of length u bits, and the message input to the compression function of H is v bits).
10669 For MD2 and MD5, $u=128$ and $v=512$; for SHA-1, $u=160$ and $v=512$.

10670 We assume here that u and v are both multiples of 8, as are the lengths in bits of the password and salt
10671 strings and the number n of pseudo-random bits required. In addition, u and v are of course nonzero.

- 10672 1. Construct a string, D (the “diversifier”), by concatenating $v/8$ copies of ID .
- 10673 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
10674 salt may be truncated to create S). Note that if the salt is the empty string, then so is S .
- 10675 3. Concatenate copies of the password together to create a string P of length $v\lceil p/v \rceil$ bits (the final copy
10676 of the password may be truncated to create P). Note that if the password is the empty string, then so
10677 is P .
- 10678 4. Set $I=S||P$ to be the concatenation of S and P .
- 10679 5. Set $j=\lceil n/u \rceil$.
- 10680 6. For $i=1, 2, \dots, j$, do the following:
 - 10681 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
10682 that hash; etc.; continue in this fashion until a total of c hashes have been computed, each on
10683 the result of the previous hash.
 - 10684 b. Concatenate copies of A_i to create a string B of length v bits (the final copy of A_i may be
10685 truncated to create B).
 - 10686 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
10687 setting $I_j=(I_j+B+1) \bmod 2^v$ for each j . To perform this addition, treat each v -bit block as a
10688 binary number represented most-significant bit first.
- 10689 7. Concatenate A_1, A_2, \dots, A_j together to form a pseudo-random bit string, A .
- 10690 8. Use the first n bits of A as the output of this entire process.

10691 When the password-based encryption mechanisms presented in this section are used to generate a key
10692 and IV (if needed) from a password, salt, and an iteration count, the above algorithm is used. To
10693 generate a key, the identifier byte ID is set to the value 1; to generate an IV, the identifier byte ID is set to
10694 the value 2.

10695 When the password based authentication mechanism presented in this section is used to generate a key
10696 from a password, salt, and an iteration count, the above algorithm is used. The identifier byte ID is set to
10697 the value 3.

6.38.1 SHA-1-PBE for 3-key triple-DES-CBC

SHA-1-PBE for 3-key triple-DES-CBC, denoted **CKM_PBE_SHA1_DES3_EDE_CBC**, is a mechanism used for generating a 3-key triple-DES secret key and IV from a password and a salt value by using the SHA-1 digest algorithm and an iteration count. The method used to generate the key and IV is described above. Each byte of the key produced will have its low-order bit adjusted, if necessary, so that a valid 3-key triple-DES key with proper parity bits is obtained.

It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the key generation process and the location of the application-supplied buffer which will receive the 8-byte IV generated by the mechanism.

The key and IV produced by this mechanism will typically be used for performing password-based encryption.

6.38.2 SHA-1-PBE for 2-key triple-DES-CBC

SHA-1-PBE for 2-key triple-DES-CBC, denoted **CKM_PBE_SHA1_DES2_EDE_CBC**, is a mechanism used for generating a 2-key triple-DES secret key and IV from a password and a salt value by using the SHA-1 digest algorithm and an iteration count. The method used to generate the key and IV is described above. Each byte of the key produced will have its low-order bit adjusted, if necessary, so that a valid 2-key triple-DES key with proper parity bits is obtained.

It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the key generation process and the location of the application-supplied buffer which will receive the 8-byte IV generated by the mechanism.

The key and IV produced by this mechanism will typically be used for performing password-based encryption.

6.38.3 SHA-1-PBA for SHA-1-HMAC

SHA-1-PBA for SHA-1-HMAC, denoted **CKM_PBA_SHA1_WITH_SHA1_HMAC**, is a mechanism used for generating a 160-bit generic secret key from a password and a salt value by using the SHA-1 digest algorithm and an iteration count. The method used to generate the key is described above.

It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the key generation process. The parameter also has a field to hold the location of an application-supplied buffer which will receive an IV; for this mechanism, the contents of this field are ignored, since authentication with SHA-1-HMAC does not require an IV.

The key generated by this mechanism will typically be used for computing a SHA-1 HMAC to perform password-based authentication (not *password-based encryption*). At the time of this writing, this is primarily done to ensure the integrity of a PKCS #12 PDU.

6.39 SSL

Table 185,SSL Mechanisms vs. Functions

| Mechanism | Functions | | | | | | |
|-----------------------------|-------------------|---------------|----------------------|--------|----------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen - Key / Key Pair | Wrap & Unwrap | Derive |
| CKM_SSL3_PRE_MASTER_KEY_GEN | | | | | ✓ | | |
| CKM_TLS_PRE_MASTER_KEY_GEN | | | | | ✓ | | |

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

10733 6.39.1 Definitions

10734 Mechanisms:

10735 CKM_SSL3_PRE_MASTER_KEY_GEN
 10736 CKM_TLS_PRE_MASTER_KEY_GEN
 10737 CKM_SSL3_MASTER_KEY_DERIVE
 10738 CKM_SSL3_KEY_AND_MAC_DERIVE
 10739 CKM_SSL3_MASTER_KEY_DERIVE_DH
 10740 CKM_SSL3_MD5_MAC
 10741 CKM_SSL3_SHA1_MAC

10742 6.39.2 SSL mechanism parameters

10743 ♦ CK_SSL3_RANDOM_DATA

10744 CK_SSL3_RANDOM_DATA is a structure which provides information about the random data of a client
 10745 and a server in an SSL context. This structure is used by both the CKM_SSL3_MASTER_KEY_DERIVE
 10746 and the CKM_SSL3_KEY_AND_MAC_DERIVE mechanisms. It is defined as follows:

```
10747     typedef struct CK_SSL3_RANDOM_DATA {
10748         CK_BYTE_PTR pClientRandom;
10749         CK ULONG ulClientRandomLen;
10750         CK_BYTE_PTR pServerRandom;
10751         CK ULONG ulServerRandomLen;
10752     } CK_SSL3_RANDOM_DATA;
```

10753

10754 The fields of the structure have the following meanings:

| | |
|-------------------------|---|
| 10755 pClientRandom | pointer to the client's random data |
| 10756 ulClientRandomLen | length in bytes of the client's random data |
| 10757 pServerRandom | pointer to the server's random data |
| 10758 ulServerRandomLen | length in bytes of the server's random data |

10759 ◆ **CK_SSL3_MASTER_KEY_DERIVE_PARAMS;**
10760 **CK_SSL3_MASTER_KEY_DERIVE_PARAMS_PTR**

10761 **CK_SSL3_MASTER_KEY_DERIVE_PARAMS** is a structure that provides the parameters to the
10762 **CKM_SSL3_MASTER_KEY_DERIVE** mechanism. It is defined as follows:

```
10763       typedef struct CK_SSL3_MASTER_KEY_DERIVE_PARAMS {  
10764           CK_SSL3_RANDOM_DATA RandomInfo;  
10765           CK_VERSION_PTR pVersion;  
10766       } CK_SSL3_MASTER_KEY_DERIVE_PARAMS;
```

10767

10768 The fields of the structure have the following meanings:

10769 RandomInfo client's and server's random data information.

10770 pVersion pointer to a **CK_VERSION** structure which receives the SSL
10771 protocol version information

10772 **CK_SSL3_MASTER_KEY_DERIVE_PARAMS_PTR** is a pointer to a
10773 **CK_SSL3_MASTER_KEY_DERIVE_PARAMS**.

10774 ◆ **CK_SSL3_KEY_MAT_OUT; CK_SSL3_KEY_MAT_OUT_PTR**

10775 **CK_SSL3_KEY_MAT_OUT** is a structure that contains the resulting key handles and initialization vectors
10776 after performing a C_DeriveKey function with the **CKM_SSL3_KEY_AND_MAC_DERIVE** mechanism. It
10777 is defined as follows:

```
10778       typedef struct CK_SSL3_KEY_MAT_OUT {  
10779           CK_OBJECT_HANDLE hClientMacSecret;  
10780           CK_OBJECT_HANDLE hServerMacSecret;  
10781           CK_OBJECT_HANDLE hClientKey;  
10782           CK_OBJECT_HANDLE hServerKey;  
10783           CK_BYTE_PTR pIVClient;  
10784           CK_BYTE_PTR pIVServer;  
10785       } CK_SSL3_KEY_MAT_OUT;
```

10786

10787 The fields of the structure have the following meanings:

10788 hClientMacSecret key handle for the resulting Client MAC Secret key

10789 hServerMacSecret key handle for the resulting Server MAC Secret key

10790 hClientKey key handle for the resulting Client Secret key

10791 hServerKey key handle for the resulting Server Secret key

10792 pIVClient pointer to a location which receives the initialization vector (IV)
10793 created for the client (if any)

10794 pIVServer pointer to a location which receives the initialization vector (IV)
10795 created for the server (if any)

10796 **CK_SSL3_KEY_MAT_OUT_PTR** is a pointer to a **CK_SSL3_KEY_MAT_OUT**.

10797 ◆ **CK_SSL3_KEY_MAT_PARAMS; CK_SSL3_KEY_MAT_PARAMS_PTR**

10798 **CK_SSL3_KEY_MAT_PARAMS** is a structure that provides the parameters to the
10799 **CKM_SSL3_KEY_AND_MAC_DERIVE** mechanism. It is defined as follows:

```

10800     typedef struct CK_SSL3_KEY_MAT_PARAMS {
10801         CK ULONG             ulMacSizeInBits;
10802         CK ULONG             ulKeySizeInBits;
10803         CK ULONG             ulIVSizeInBits;
10804         CK_BBOOL            bIsExport;
10805         CK_SSL3_RANDOM_DATA RandomInfo;
10806         CK_SSL3_KEY_MAT_OUT_PTR pReturnedKeyMaterial;
10807     } CK_SSL3_KEY_MAT_PARAMS;

```

10808

10809 The fields of the structure have the following meanings:

| | | |
|-------|----------------------|---|
| 10810 | ulMacSizeInBits | the length (in bits) of the MACing keys agreed upon during the protocol handshake phase |
| 10812 | ulKeySizeInBits | the length (in bits) of the secret keys agreed upon during the protocol handshake phase |
| 10814 | ulIVSizeInBits | the length (in bits) of the IV agreed upon during the protocol handshake phase. If no IV is required, the length should be set to 0 |
| 10816 | bIsExport | a Boolean value which indicates whether the keys have to be derived for an export version of the protocol |
| 10818 | RandomInfo | client's and server's random data information. |
| 10819 | pReturnedKeyMaterial | points to a CK_SSL3_KEY_MAT_OUT structures which receives the handles for the keys generated and the IVs |

10821 **CK_SSL3_KEY_MAT_PARAMS_PTR** is a pointer to a **CK_SSL3_KEY_MAT_PARAMS**.

10822 6.39.3 Pre-master key generation

10823 Pre-master key generation in SSL 3.0, denoted **CKM_SSL3_PRE_MASTER_KEY_GEN**, is a mechanism
 10824 which generates a 48-byte generic secret key. It is used to produce the "pre_master" key used in SSL
 10825 version 3.0 for RSA-like cipher suites.

10826 It has one parameter, a **CK_VERSION** structure, which provides the client's SSL version number.

10827 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
 10828 key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may
 10829 be specified in the template, or else are assigned default values.

10830 The template sent along with this mechanism during a **C_GenerateKey** call may indicate that the object
 10831 class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN**
 10832 attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to
 10833 specify any of them.

10834 For this mechanism, the **ulMinKeySize** and **ulMaxKeySize** fields of the **CK_MECHANISM_INFO** structure
 10835 both indicate 48 bytes.

10836 **CKM_TLS_PRE_MASTER_KEY_GEN** has identical functionality as
 10837 **CKM_SSL3_PRE_MASTER_KEY_GEN**. It exists only for historical reasons, please use
 10838 **CKM_SSL3_PRE_MASTER_KEY_GEN** instead.

10839 6.39.4 Master key derivation

10840 Master key derivation in SSL 3.0, denoted **CKM_SSL3_MASTER_KEY_DERIVE**, is a mechanism used
 10841 to derive one 48-byte generic secret key from another 48-byte generic secret key. It is used to produce
 10842 the "master_secret" key used in the SSL protocol from the "pre_master" key. This mechanism returns the
 10843 value of the client version, which is built into the "pre_master" key as well as a handle to the derived
 10844 "master_secret" key.

10845 It has a parameter, a **CK_SSL3_MASTER_KEY_DERIVE_PARAMS** structure, which allows for the
10846 passing of random data to the token as well as the returning of the protocol version number which is part
10847 of the pre-master key. This structure is defined in Section 6.39.

10848 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
10849 key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may
10850 be specified in the template; otherwise they are assigned default values.

10851 The template sent along with this mechanism during a **C_DeriveKey** call may indicate that the object
10852 class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN**
10853 attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to
10854 specify any of them.

10855 This mechanism has the following rules about key sensitivity and extractability:

- 10856 • The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both
10857 be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some
10858 default value.
- 10859 • If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key
10860 will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the
10861 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
10862 **CKA_SENSITIVE** attribute.
- 10863 • Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
10864 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
10865 CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
10866 value from its **CKA_EXTRACTABLE** attribute.

10867 For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK_MECHANISM_INFO** structure
10868 both indicate 48 bytes.

10869 Note that the **CK_VERSION** structure pointed to by the **CK_SSL3_MASTER_KEY_DERIVE_PARAMS**
10870 structure's *pVersion* field will be modified by the **C_DeriveKey** call. In particular, when the call returns,
10871 this structure will hold the SSL version associated with the supplied pre_master key.

10872 Note that this mechanism is only useable for cipher suites that use a 48-byte "pre_master" secret with an
10873 embedded version number. This includes the RSA cipher suites, but excludes the Diffie-Hellman cipher
10874 suites.

10875 6.39.5 Master key derivation for Diffie-Hellman

10876 Master key derivation for Diffie-Hellman in SSL 3.0, denoted **CKM_SSL3_MASTER_KEY_DERIVE_DH**,
10877 is a mechanism used to derive one 48-byte generic secret key from another arbitrary length generic
10878 secret key. It is used to produce the "master_secret" key used in the SSL protocol from the "pre_master"
10879 key.

10880 It has a parameter, a **CK_SSL3_MASTER_KEY_DERIVE_PARAMS** structure, which allows for the
10881 passing of random data to the token. This structure is defined in Section 6.39. The *pVersion* field of the
10882 structure must be set to NULL_PTR since the version number is not embedded in the "pre_master" key
10883 as it is for RSA-like cipher suites.

10884 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
10885 key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may
10886 be specified in the template, or else are assigned default values.

10887 The template sent along with this mechanism during a **C_DeriveKey** call may indicate that the object
10888 class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN**
10889 attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to
10890 specify any of them.

10891 This mechanism has the following rules about key sensitivity and extractability:

- 10892 • The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both
10893 be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some
10894 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 *ullVSizeInBits* 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 *ullVSizeInBits* 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 **NULPTR**.

10945 If a call to **C_DeriveKey** with this mechanism fails, then *none* of the four keys will be created on the
10946 token.

10947 6.39.7 MD5 MACing in SSL 3.0

10948 MD5 MACing in SSL3.0, denoted **CKM_SSL3_MD5_MAC**, is a mechanism for single- and multiple-part
10949 signatures (data authentication) and verification using MD5, based on the SSL 3.0 protocol. This
10950 technique is very similar to the HMAC technique.

10951 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which specifies the length in bytes of the
10952 signatures produced by this mechanism.

10953 Constraints on key types and the length of input and output data are summarized in the following table:

10954 *Table 186, MD5 MACing in SSL 3.0: Key And Data Length*

| Function | Key type | Data length | Signature length |
|----------|----------------|-------------|------------------------------|
| C_Sign | generic secret | any | 4-8, depending on parameters |
| C_Verify | generic secret | any | 4-8, depending on parameters |

10955 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
10956 specify the supported range of generic secret key sizes, in bits.

10957 6.39.8 SHA-1 MACing in SSL 3.0

10958 SHA-1 MACing in SSL3.0, denoted **CKM_SSL3_SHA1_MAC**, is a mechanism for single- and multiple-part
10959 signatures (data authentication) and verification using SHA-1, based on the SSL 3.0 protocol. This
10960 technique is very similar to the HMAC technique.

10961 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which specifies the length in bytes of the
10962 signatures produced by this mechanism.

10963 Constraints on key types and the length of input and output data are summarized in the following table:

10964 *Table 187, SHA-1 MACing in SSL 3.0: Key And Data Length*

| Function | Key type | Data length | Signature length |
|----------|----------------|-------------|------------------------------|
| C_Sign | generic secret | any | 4-8, depending on parameters |
| C_Verify | generic secret | any | 4-8, depending on parameters |

10965 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
10966 specify the supported range of generic secret key sizes, in bits.

10967 6.40 TLS 1.2 Mechanisms

10968 Details for TLS 1.2 and its key derivation and MAC mechanisms can be found in [TLS12]. TLS 1.2
10969 mechanisms differ from TLS 1.0 and 1.1 mechanisms in that the base hash used in the underlying TLS
10970 PRF (pseudo-random function) can be negotiated. Therefore each mechanism parameter for the TLS 1.2
10971 mechanisms contains a new value in the parameters structure to specify the hash function.

10972 This section also specifies **CKM_TLS12_MAC** which should be used in place of **CKM_TLS_PRF** to
10973 calculate the verify_data in the TLS "finished" message.

10974 This section also specifies **CKM_TLS_KDF** that can be used in place of **CKM_TLS_PRF** to implement
10975 key material exporters.

10976

10977 *Table 188, TLS 1.2 Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|--------------------------------|-------------------|---------------|----------------------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_TLS12_MASTER_KEY_DERIVE | | | | | | | ✓ |
| CKM_TLS12_MASTER_KEY_DERIVE_DH | | | | | | | ✓ |
| CKM_TLS12_KEY_AND_MAC_DERIVE | | | | | | | ✓ |
| CKM_TLS12_KEY_SAFE_DERIVE | | | | | | | ✓ |
| CKM_TLS_KDF | | | | | | | ✓ |
| CKM_TLS12_MAC | | ✓ | | | | | |
| CKM_TLS12_KDF | | | | | | | ✓ |

6.40.1 Definitions

Mechanisms:

```
CKM_TLS12_MASTER_KEY_DERIVE
CKM_TLS12_MASTER_KEY_DERIVE_DH
CKM_TLS12_KEY_AND_MAC_DERIVE
CKM_TLS12_KEY_SAFE_DERIVE
CKM_TLS_KDF
CKM_TLS12_MAC
CKM_TLS12_KDF
```

6.40.2 TLS 1.2 mechanism parameters

- ◆ **CK_TLS12_MASTER_KEY_DERIVE_PARAMS;**
- CK_TLS12_MASTER_KEY_DERIVE_PARAMS_PTR**

CK_TLS12_MASTER_KEY_DERIVE_PARAMS is a structure that provides the parameters to the **CKM_TLS12_MASTER_KEY_DERIVE** mechanism. It is defined as follows:

```
typedef struct CK_TLS12_MASTER_KEY_DERIVE_PARAMS {
    CK_SSL3_RANDOM_DATA RandomInfo;
    CK_VERSION_PTR pVersion;
    CK_MECHANISM_TYPE prfHashMechanism;
} CK_TLS12_MASTER_KEY_DERIVE_PARAMS;
```

The fields of the structure have the following meanings:

RandomInfo client's and server's random data information.

pVersion pointer to a **CK_VERSION** structure which receives the SSL protocol version information

prfHashMechanism base hash used in the underlying TLS1.2 PRF operation used to derive the master key.

CK_TLS12_MASTER_KEY_DERIVE_PARAMS_PTR is a pointer to a **CK_TLS12_MASTER_KEY_DERIVE_PARAMS**.

11007 ◆ **CK_TLS12_KEY_MAT_PARAMS; CK_TLS12_KEY_MAT_PARAMS_PTR**

11008 **CK_TLS12_KEY_MAT_PARAMS** is a structure that provides the parameters to the
11009 **CKM_TLS12_KEY_AND_MAC_DERIVE** mechanism. It is defined as follows:

```
11010     typedef struct CK_TLS12_KEY_MAT_PARAMS {  
11011         CK ULONG ulMacSizeInBits;  
11012         CK ULONG ulKeySizeInBits;  
11013         CK ULONG ullIVSizeInBits;  
11014         CK_BBOOL bIsExport;  
11015         CK_SSL3_RANDOM_DATA RandomInfo;  
11016         CK_SSL3_KEY_MAT_OUT_PTR pReturnedKeyMaterial;  
11017         CK_MECHANISM_TYPE prfHashMechanism;  
11018     } CK_TLS12_KEY_MAT_PARAMS;
```

11019 The fields of the structure have the following meanings:

| | | |
|-------|----------------------|--|
| 11021 | ulMacSizeInBits | the length (in bits) of the MACing keys agreed upon during the protocol handshake phase. If no MAC key is required, the length should be set to 0. |
| 11024 | ulKeySizeInBits | the length (in bits) of the secret keys agreed upon during the protocol handshake phase |
| 11026 | ullIVSizeInBits | the length (in bits) of the IV agreed upon during the protocol handshake phase. If no IV is required, the length should be set to 0 |
| 11028 | bIsExport | must be set to CK_FALSE because export cipher suites must not be used in TLS 1.1 and later. |
| 11030 | RandomInfo | client's and server's random data information. |
| 11031 | pReturnedKeyMaterial | points to a CK_SSL3_KEY_MAT_OUT structures which receives the handles for the keys generated and the IVs |
| 11033 | prfHashMechanism | base hash used in the underlying TLS1.2 PRF operation used to derive the master key. |

11035 **CK_TLS12_KEY_MAT_PARAMS_PTR** is a pointer to a **CK_TLS12_KEY_MAT_PARAMS**.

11036 ◆ **CK_TLS_KDF_PARAMS; CK_TLS_KDF_PARAMS_PTR**

11037 **CK_TLS_KDF_PARAMS** is a structure that provides the parameters to the CKM_TLS_KDF mechanism.
11038 It is defined as follows:

```
11039     typedef struct CK_TLS_KDF_PARAMS {  
11040         CK_MECHANISM_TYPE prfMechanism;  
11041         CK_BYTE_PTR pLabel;  
11042         CK ULONG ulLabelLength;  
11043         CK_SSL3_RANDOM_DATA RandomInfo;  
11044         CK_BYTE_PTR pContextData;  
11045         CK ULONG ulContextDataLength;  
11046     } CK_TLS_KDF_PARAMS;
```

11048 The fields of the structure have the following meanings:

| | | |
|-------|--------------|---|
| 11049 | prfMechanism | the hash mechanism used in the TLS1.2 PRF construct or 11050 CKM_TLS_PRF to use with the TLS1.0 and 1.1 PRF construct. |
|-------|--------------|---|

11051 pLabel a pointer to the label for this key derivation
11052 ulLabelLength length of the label in bytes
11053 RandomInfo the random data for the key derivation
11054 pContextData a pointer to the context data for this key derivation. NULL_PTR if not
11055 present
11056 ulContextDataLength length of the context data in bytes. 0 if not present.
11057 **CK_TLS_KDF_PARAMS_PTR** is a pointer to a **CK_TLS_KDF_PARAMS**.

11058 ◆ **CK_TLS_MAC_PARAMS; CK_TLS_MAC_PARAMS_PTR**

11059 **CK_TLS_MAC_PARAMS** is a structure that provides the parameters to the **CKM_TLS_MAC**
11060 mechanism. It is defined as follows:

```
11061        typedef struct CK_TLS_MAC_PARAMS {  
11062            CK_MECHANISM_TYPE prfHashMechanism;  
11063            CK ULONG ulMacLength;  
11064            CK ULONG ulServerOrClient;  
11065        } CK_TLS_MAC_PARAMS;  
11066
```

11067 The fields of the structure have the following meanings:

11068 prfHashMechanism the hash mechanism used in the TLS12 PRF construct or
11069 CKM_TLS_PRF to use with the TLS1.0 and 1.1 PRF construct.
11070 ulMacLength the length of the MAC tag required or offered. Always 12 octets in
11071 TLS 1.0 and 1.1. Generally 12 octets, but may be negotiated to a
11072 longer value in TLS1.2.
11073 ulServerOrClient 1 to use the label "server finished", 2 to use the label "client
11074 finished". All other values are invalid.

11075 **CK_TLS_MAC_PARAMS_PTR** is a pointer to a **CK_TLS_MAC_PARAMS**.

11076

11077 ◆ **CK_TLS_PRF_PARAMS; CK_TLS_PRF_PARAMS_PTR**

11078 **CK_TLS_PRF_PARAMS** is a structure, which provides the parameters to the **CKM_TLS_PRF**
11079 mechanism. It is defined as follows:

```
11080        typedef struct CK_TLS_PRF_PARAMS {  
11081            CK_BYTE_PTR      pSeed;  
11082            CK ULONG      ulSeedLen;  
11083            CK_BYTE_PTR      pLabel;  
11084            CK ULONG      ullLabelLen;  
11085            CK_BYTE_PTR      pOutput;  
11086            CK ULONG_PTR      pulOutputLen;  
11087        } CK_TLS_PRF_PARAMS;  
11088
```

11089 The fields of the structure have the following meanings:

11090 pSeed *pointer to the input seed*
11091 ulSeedLen *length in bytes of the input seed*
11092 pLabel *pointer to the identifying label*

11093 ulLabelLen *length in bytes of the identifying label*
 11094 pOutput *pointer receiving the output of the operation*
 11095 pulOutputLen *pointer to the length in bytes that the output to be created shall have, has to hold the desired length as input and will receive the calculated length as output*
 11097
 11098 CK_TLS_PRF_PARAMS_PTR is a pointer to a CK_TLS_PRF_PARAMS.

11099 6.40.3 TLS MAC

11100 The TLS MAC mechanism is used to generate integrity tags for the TLS "finished" message. It replaces
 11101 the use of the **CKM_TLS_PRF** function for TLS1.0 and 1.1 and that mechanism is deprecated.
 11102 **CKM_TLS_MAC** takes a parameter of CK_TLS_MAC_PARAMS. To use this mechanism with TLS1.0
 11103 and TLS1.1, use **CKM_TLS_PRF** as the value for *prfMechanism* in place of a hash mechanism. Note:
 11104 Although **CKM_TLS_PRF** is deprecated as a mechanism for C_DeriveKey, the manifest value is retained
 11105 for use with this mechanism to indicate the use of the TLS1.0/1.1 pseudo-random function.
 11106 In TLS1.0 and 1.1 the "finished" message verify_data (i.e. the output signature from the MAC mechanism)
 11107 is always 12 bytes. In TLS1.2 the "finished" message verify_data is a minimum of 12 bytes, defaults to 12
 11108 bytes, but may be negotiated to longer length.
 11109 Table 189, 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 |

11110

11111 6.40.4 Master key derivation

11112 Master key derivation in TLS 1.0, denoted **CKM_TLS_MASTER_KEY_DERIVE**, is a mechanism used to
 11113 derive one 48-byte generic secret key from another 48-byte generic secret key. It is used to produce the
 11114 "master_secret" key used in the TLS protocol from the "pre_master" key. This mechanism returns the
 11115 value of the client version, which is built into the "pre_master" key as well as a handle to the derived
 11116 "master_secret" key.

11117 It has a parameter, a **CK_SSL3_MASTER_KEY_DERIVE_PARAMS** structure, which allows for the
 11118 passing of random data to the token as well as the returning of the protocol version number which is part
 11119 of the pre-master key. This structure is defined in Section 6.39.

11120 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
 11121 key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may
 11122 be specified in the template, or else are assigned default values.

11123 The mechanism also contributes the **CKA_ALLOWED_MECHANISMS** attribute consisting only of
 11124 **CKM_TLS12_KEY_AND_MAC_DERIVE**, **CKM_TLS12_KEY_SAFE_DERIVE**, **CKM_TLS12_KDF** and
 11125 **CKM_TLS12_MAC**.

11126 The template sent along with this mechanism during a **C_DeriveKey** call may indicate that the object
 11127 class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN**
 11128 attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to
 11129 specify any of them.

11130 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

11136 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
11137 **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.

11142 For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK_MECHANISM_INFO** structure
11143 both indicate 48 bytes.

11144 Note that the **CK_VERSION** structure pointed to by the **CK_SSL3_MASTER_KEY_DERIVE_PARAMS**
11145 structure's *pVersion* field will be modified by the **C_DeriveKey** call. In particular, when the call returns,
11146 this structure will hold the SSL version associated with the supplied pre_master key.

11147 Note that this mechanism is only useable for cipher suites that use a 48-byte "pre_master" secret with an
11148 embedded version number. This includes the RSA cipher suites, but excludes the Diffie-Hellman cipher
11149 suites.

11150 6.40.5 Master key derivation for Diffie-Hellman

11151 Master key derivation for Diffie-Hellman in TLS 1.0, denoted **CKM_TLS_MASTER_KEY_DERIVE_DH**, is
11152 a mechanism used to derive one 48-byte generic secret key from another arbitrary length generic secret
11153 key. It is used to produce the "master_secret" key used in the TLS protocol from the "pre_master" key.

11154 It has a parameter, a **CK_SSL3_MASTER_KEY_DERIVE_PARAMS** structure, which allows for the
11155 passing of random data to the token. This structure is defined in Section 6.39. The *pVersion* field of the
11156 structure must be set to NULL_PTR since the version number is not embedded in the "pre_master" key
11157 as it is for RSA-like cipher suites.

11158 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
11159 key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may
11160 be specified in the template, or else are assigned default values.

11161 The mechanism also contributes the **CKA_ALLOWED_MECHANISMS** attribute consisting only of
11162 **CKM_TLS12_KEY_AND_MAC_DERIVE**, **CKM_TLS12_KEY_SAFE_DERIVE**, **CKM_TLS12_KDF** and
11163 **CKM_TLS12_MAC**.

11164 The template sent along with this mechanism during a **C_DeriveKey** call may indicate that the object
11165 class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN**
11166 attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to
11167 specify any of them.

11168 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
11169 be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some
11170 default value.
- If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key
11171 will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the
11172 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
11173 **CKA_SENSITIVE** attribute.
- Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the derived key
11174 will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_TRUE, then the derived key has its
11175 **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite* value from its **CKA_EXTRACTABLE** attribute.

11176 For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK_MECHANISM_INFO** structure
11177 both indicate 48 bytes.

11178 Note that this mechanism is only useable for cipher suites that do not use a fixed length 48-byte
11179 "pre_master" secret with an embedded version number. This includes the Diffie-Hellman cipher suites, but
11180 excludes the RSA cipher suites.

11185 **6.40.6 Key and MAC derivation**

11186 Key, MAC and IV derivation in TLS 1.0, denoted **CKM_TLS_KEY_AND_MAC_DERIVE**, is a mechanism
11187 used to derive the appropriate cryptographic keying material used by a "CipherSuite" from the
11188 "master_secret" key and random data. This mechanism returns the key handles for the keys generated in
11189 the process, as well as the IVs created.

11190 It has a parameter, a **CK_SSL3_KEY_MAT_PARAMS** structure, which allows for the passing of random
11191 data as well as the characteristic of the cryptographic material for the given CipherSuite and a pointer to a
11192 structure which receives the handles and IVs which were generated. This structure is defined in Section
11193 6.39.

11194 This mechanism contributes to the creation of four distinct keys on the token and returns two IVs (if IVs
11195 are requested by the caller) back to the caller. The keys are all given an object class of
11196 **CKO_SECRET_KEY**.

11197 The two MACing keys ("client_write_MAC_secret" and "server_write_MAC_secret") (if present) are
11198 always given a type of **CKK_GENERIC_SECRET**. They are flagged as valid for signing and verification.

11199 The other two keys ("client_write_key" and "server_write_key") are typed according to information found
11200 in the template sent along with this mechanism during a **C_DeriveKey** function call. By default, they are
11201 flagged as valid for encryption, decryption, and derivation operations.

11202 For **CKM_TLS12_KEY_AND_MAC_DERIVE**, IVs will be generated and returned if the *ullVSizeInBits*
11203 field of the **CK_SSL3_KEY_MAT_PARAMS** field has a nonzero value. If they are generated, their length
11204 in bits will agree with the value in the *ullVSizeInBits* field.

11205

11206 Note Well: CKM_TLS12_KEY_AND_MAC_DERIVE produces both private (key) and public (IV)
11207 data. It is possible to "leak" private data by the simple expedient of decreasing the length of
11208 private data requested. E.g. Setting *ulMacSizeInBits* and *ulKeySizeInBits* to 0 (or other lengths
11209 less than the key size) will result in the private key data being placed in the destination
11210 designated for the IV's. Repeated calls with the same master key and same RandomInfo but with
11211 differing lengths for the private key material will result in different data being leaked.<

11212

11213 All four keys inherit the values of the **CKA_SENSITIVE**, **CKA_ALWAYS_SENSITIVE**,
11214 **CKA_EXTRACTABLE**, and **CKA_NEVER_EXTRACTABLE** attributes from the base key. The template
11215 provided to **C_DeriveKey** may not specify values for any of these attributes which differ from those held
11216 by the base key.

11217 Note that the **CK_SSL3_KEY_MAT_OUT** structure pointed to by the **CK_SSL3_KEY_MAT_PARAMS**
11218 structure's *pReturnedKeyMaterial* field will be modified by the **C_DeriveKey** call. In particular, the four
11219 key handle fields in the **CK_SSL3_KEY_MAT_OUT** structure will be modified to hold handles to the
11220 newly-created keys; in addition, the buffers pointed to by the **CK_SSL3_KEY_MAT_OUT** structure's
11221 *pIVClient* and *pIVServer* fields will have IVs returned in them (if IVs are requested by the caller).
11222 Therefore, these two fields must point to buffers with sufficient space to hold any IVs that will be returned.

11223 This mechanism departs from the other key derivation mechanisms in Cryptoki in its returned information.
11224 For most key-derivation mechanisms, **C_DeriveKey** returns a single key handle as a result of a
11225 successful completion. However, since the **CKM_TLS12_KEY_AND_MAC_DERIVE** mechanism returns
11226 all of its key handles in the **CK_SSL3_KEY_MAT_OUT** structure pointed to by the
11227 **CK_SSL3_KEY_MAT_PARAMS** structure specified as the mechanism parameter, the parameter *phKey*
11228 passed to **C_DeriveKey** is unnecessary, and should be a **NULL_PTR**.

11229 If a call to **C_DeriveKey** with this mechanism fails, then *none* of the four keys will be created on the
11230 token.

11231 **6.40.7 CKM_TLS12_KEY_SAFE_DERIVE**

11232 **CKM_TLS12_KEY_SAFE_DERIVE** is identical to **CKM_TLS12_KEY_AND_MAC_DERIVE** except that it
11233 shall never produce IV data, and the *ullVSizeInBits* field of **CK_TLS12_KEY_MAT_PARAMS** is ignored

11234 and treated as 0. All of the other conditions and behavior described for
11235 CKM_TLS12_KEY_AND_MAC_DERIVE, with the exception of the black box warning, apply to this
11236 mechanism.

11237 CKM_TLS12_KEY_SAFE_DERIVE is provided as a separate mechanism to allow a client to control the
11238 export of IV material (and possible leaking of key material) through the use of the
11239 CKA_ALLOWED_MECHANISMS key attribute.

11240 6.40.8 Generic Key Derivation using the TLS PRF

11241 CKM_TLS_KDF is the mechanism defined in [RFC 5705]. It uses the TLS key material and TLS PRF
11242 function to produce additional key material for protocols that want to leverage the TLS key negotiation
11243 mechanism. CKM_TLS_KDF has a parameter of CK_TLS_KDF_PARAMS. If the protocol using this
11244 mechanism does not use context information, the pContextData field shall be set to NULL_PTR and the
11245 ulContextDataLength field shall be set to 0.

11246 To use this mechanism with TLS1.0 and TLS1.1, use CKM_TLS_PRF as the value for prfMechanism in
11247 place of a hash mechanism. Note: Although CKM_TLS_PRF is deprecated as a mechanism for
11248 C_DeriveKey, the manifest value is retained for use with this mechanism to indicate the use of the
11249 TLS1.0/1.1 Pseudo-random function.

11250 This mechanism can be used to derive multiple keys (e.g. similar to
11251 CKM_TLS12_KEY_AND_MAC_DERIVE) by first deriving the key stream as a CKK_GENERIC_SECRET
11252 of the necessary length and doing subsequent derives against that derived key using the
11253 CKM_EXTRACT_KEY_FROM_KEY mechanism to split the key stream into the actual operational keys.

11254 The mechanism should not be used with the labels defined for use with TLS, but the token does not
11255 enforce this behavior.

11256 This mechanism has the following rules about key sensitivity and extractability:

- 11257 • If the original key has its CKA_SENSITIVE attribute set to CK_TRUE, so does the derived key. If not,
11258 then the derived key's CKA_SENSITIVE attribute is set either from the supplied template or from the
11259 original key.
- 11260 • Similarly, if the original key has its CKA_EXTRACTABLE attribute set to CK_FALSE, so does the
11261 derived key. If not, then the derived key's CKA_EXTRACTABLE attribute is set either from the
11262 supplied template or from the original key.
- 11263 • The derived key's CKA_ALWAYS_SENSITIVE attribute is set to CK_TRUE if and only if the original
11264 key has its CKA_ALWAYS_SENSITIVE attribute set to CK_TRUE.
- 11265 • Similarly, the derived key's CKA_NEVER_EXTRACTABLE attribute is set to CK_TRUE if and only if
11266 the original key has its CKA_NEVER_EXTRACTABLE attribute set to CK_TRUE.

11267 6.40.9 Generic Key Derivation using the TLS12 PRF

11268 CKM_TLS12_KDF is the mechanism defined in [RFC 5705]. It uses the TLS key material and TLS PRF
11269 function to produce additional key material for protocols that want to leverage the TLS key negotiation
11270 mechanism. CKM_TLS12_KDF has a parameter of CK_TLS_KDF_PARAMS. If the protocol using this
11271 mechanism does not use context information, the pContextData field shall be set to NULL_PTR and the
11272 ulContextDataLength field shall be set to 0.

11273 To use this mechanism with TLS1.0 and TLS1.1, use CKM_TLS_PRF as the value for prfMechanism in
11274 place of a hash mechanism. Note: Although CKM_TLS_PRF is deprecated as a mechanism for
11275 C_DeriveKey, the manifest value is retained for use with this mechanism to indicate the use of the
11276 TLS1.0/1.1 Pseudo-random function.

11277 This mechanism can be used to derive multiple keys (e.g. similar to
11278 CKM_TLS12_KEY_AND_MAC_DERIVE) by first deriving the key stream as a CKK_GENERIC_SECRET
11279 of the necessary length and doing subsequent derives against that derived key stream using the
11280 CKM_EXTRACT_KEY_FROM_KEY mechanism to split the key stream into the actual operational keys.

11281 The mechanism should not be used with the labels defined for use with TLS, but the token does not
11282 enforce this behavior.

11283 This mechanism has the following rules about key sensitivity and extractability:

- 11284 • If the original key has its **CKA_SENSITIVE** attribute set to CK_TRUE, so does the derived key. If not,
11285 then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied template or from the
11286 original key.

- 11287 • Similarly, if the original key has its **CKA_EXTRACTABLE** attribute set to CK_FALSE, so does the
11288 derived key. If not, then the derived key's **CKA_EXTRACTABLE** attribute is set either from the
11289 supplied template or from the original key.

- 11290 • The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to CK_TRUE if and only if the original
11291 key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE.

- 11292 • Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to CK_TRUE if and only if
11293 the original key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_TRUE.

11294 6.41 WTLS

11295 Details can be found in [WTLS].

11296 When comparing the existing TLS mechanisms with these extensions to support WTLS one could argue
11297 that there would be no need to have distinct handling of the client and server side of the handshake.

11298 However, since in WTLS the server and client use different sequence numbers, there could be instances
11299 (e.g. when WTLS is used to protect asynchronous protocols) where sequence numbers on the client and
11300 server side differ, and hence this motivates the introduced split.

11301

11302 *Table 190, WTLS Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|------------------------------------|---------------------------------|-------------------------|-------------------------------|------------|-------------------------------------|-------------------------|------------|
| | Encry pt & Decry pt | Sign & Verif y | SR & VR ¹ | Dige st | Ge n. Key / Key Pair | Wrap & Unwra p | Deriv e |
| CKM_WTLS_PRE_MASTER_KEY_GEN | | | | | ✓ | | |
| CKM_WTLS_MASTER_KEY_DERIVE | | | | | | | ✓ |
| CKM_WTLS_MASTER_KEY_DERIVE_DH_ECC | | | | | | | ✓ |
| CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE | | | | | | | ✓ |
| CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE | | | | | | | ✓ |
| CKM_WTLS_PRF | | | | | | | ✓ |

11303 6.41.1 Definitions

11304 Mechanisms:

11305 CKM_WTLS_PRE_MASTER_KEY_GEN

11306 CKM_WTLS_MASTER_KEY_DERIVE

11307 CKM_WTLS_MASTER_KEY_DERIVE_DH_ECC

```
11308      CKM_WTLS_PRF  
11309      CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE  
11310      CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE
```

11311 6.41.2 WTLS mechanism parameters

11312 ♦ CK_WTLS_RANDOM_DATA; CK_WTLS_RANDOM_DATA_PTR

11313 **CK_WTLS_RANDOM_DATA** is a structure, which provides information about the random data of a client
11314 and a server in a WTLS context. This structure is used by the **CKM_WTLS_MASTER_KEY_DERIVE**
11315 mechanism. It is defined as follows:

```
11316      typedef struct CK_WTLS_RANDOM_DATA {  
11317          CK_BYTE_PTR pClientRandom;  
11318          CK ULONG ulClientRandomLen;  
11319          CK_BYTE_PTR pServerRandom;  
11320          CK ULONG ulServerRandomLen;  
11321      } CK_WTLS_RANDOM_DATA;
```

11322 The fields of the structure have the following meanings:

| | | |
|-------|-------------------|---|
| 11324 | pClientRandom | pointer to the client's random data |
| 11325 | pClientRandomLen | length in bytes of the client's random data |
| 11326 | pServerRaondom | pointer to the server's random data |
| 11327 | ulServerRandomLen | length in bytes of the server's random data |

11328 **CK_WTLS_RANDOM_DATA_PTR** is a pointer to a **CK_WTLS_RANDOM_DATA**.

11329 ♦ CK_WTLS_MASTER_KEY_DERIVE_PARAMS; 11330 CK_WTLS_MASTER_KEY_DERIVE_PARAMS_PTR

11331 **CK_WTLS_MASTER_KEY_DERIVE_PARAMS** is a structure, which provides the parameters to the
11332 **CKM_WTLS_MASTER_KEY_DERIVE** mechanism. It is defined as follows:

```
11333      typedef struct CK_WTLS_MASTER_KEY_DERIVE_PARAMS {  
11334          CK_MECHANISM_TYPE DigestMechanism;  
11335          CK_WTLS_RANDOM_DATA RandomInfo;  
11336          CK_BYTE_PTR pVersion;  
11337      } CK_WTLS_MASTER_KEY_DERIVE_PARAMS;
```

11338 The fields of the structure have the following meanings:

| | | |
|-------|-----------------|--|
| 11340 | DigestMechanism | the mechanism type of the digest mechanism to be used (possible 11341 types can be found in [WTLS]) |
| 11342 | RandomInfo | Client's and server's random data information |
| 11343 | pVersion | pointer to a CK_BYTE which receives the WTLS protocol version 11344 information |

11345 **CK_WTLS_MASTER_KEY_DERIVE_PARAMS_PTR** is a pointer to a
11346 **CK_WTLS_MASTER_KEY_DERIVE_PARAMS**.

11347 ◆ **CK_WTLS_PRF_PARAMS; CK_WTLS_PRF_PARAMS_PTR**

11348 **CK_WTLS_PRF_PARAMS** is a structure, which provides the parameters to the **CKM_WTLS_PRF**
11349 mechanism. It is defined as follows:

```
11350     typedef struct CK_WTLS_PRF_PARAMS {  
11351         CK_MECHANISM_TYPE DigestMechanism;  
11352         CK_BYTE_PTR pSeed;  
11353         CK ULONG ulSeedLen;  
11354         CK_BYTE_PTR pLabel;  
11355         CK ULONG ulLabelLen;  
11356         CK_BYTE_PTR pOutput;  
11357         CK ULONG_PTR pulOutputLen;  
11358     } CK_WTLS_PRF_PARAMS;
```

11359 The fields of the structure have the following meanings:

| | | |
|-------|------------------|---|
| 11361 | Digest Mechanism | the mechanism type of the digest mechanism to be used (possible 11362 types can be found in [WTLS]) |
| 11363 | pSeed | pointer to the input seed |
| 11364 | ulSeedLen | length in bytes of the input seed |
| 11365 | pLabel | pointer to the identifying label |
| 11366 | ulLabelLen | length in bytes of the identifying label |
| 11367 | pOutput | pointer receiving the output of the operation |
| 11368 | pulOutputLen | pointer to the length in bytes that the output to be created shall 11369 have, has to hold the desired length as input and will receive the 11370 calculated length as output |

11371 **CK_WTLS_PRF_PARAMS_PTR** is a pointer to a **CK_WTLS_PRF_PARAMS**.

11372 ◆ **CK_WTLS_KEY_MAT_OUT; CK_WTLS_KEY_MAT_OUT_PTR**

11373 **CK_WTLS_KEY_MAT_OUT** is a structure that contains the resulting key handles and initialization
11374 vectors after performing a `C_DeriveKey` function with the
11375 **CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE** or with the
11376 **CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE** mechanism. It is defined as follows:

```
11377     typedef struct CK_WTLS_KEY_MAT_OUT {  
11378         CK_OBJECT_HANDLE hMacSecret;  
11379         CK_OBJECT_HANDLE hKey;  
11380         CK_BYTE_PTR pIV;  
11381     } CK_WTLS_KEY_MAT_OUT;
```

11382

11383 The fields of the structure have the following meanings:

| | | |
|-------|------------|---|
| 11384 | hMacSecret | Key handle for the resulting MAC secret key |
| 11385 | hKey | Key handle for the resulting secret key |
| 11386 | pIV | Pointer to a location which receives the initialization vector (IV) 11387 created (if any) |

11388 **CK_WTLS_KEY_MAT_OUT_PTR** is a pointer to a **CK_WTLS_KEY_MAT_OUT**.

11389 ◆ **CK_WTLS_KEY_MAT_PARAMS; CK_WTLS_KEY_MAT_PARAMS_PTR**

11390 **CK_WTLS_KEY_MAT_PARAMS** is a structure that provides the parameters to the
11391 **CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE** and the
11392 **CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE** mechanisms. It is defined as follows:

```
11393     typedef struct CK_WTLS_KEY_MAT_PARAMS {  
11394         CK_MECHANISM_TYPE          DigestMechanism;  
11395         CK ULONG                  ulMacSizeInBits;  
11396         CK ULONG                  ulKeySizeInBits;  
11397         CK ULONG                  ulIVSizeInBits;  
11398         CK ULONG                  ulSequenceNumber;  
11399         CK_BBOOL                 bIsExport;  
11400         CK_WTLS_RANDOM_DATA      RandomInfo;  
11401         CK_WTLS_KEY_MAT_OUT_PTR pReturnedKeyMaterial;  
11402     } CK_WTLS_KEY_MAT_PARAMS;
```

11403

11404 The fields of the structure have the following meanings:

| | | |
|-------|----------------------|---|
| 11405 | Digest Mechanism | the mechanism type of the digest mechanism to be used (possible types can be found in [WTLS]) |
| 11407 | ulMaxSizeInBits | the length (in bits) of the MACing key agreed upon during the protocol handshake phase |
| 11409 | ulKeySizeInBits | the length (in bits) of the secret key agreed upon during the handshake phase |
| 11411 | ulIVSizeInBits | the length (in bits) of the IV agreed upon during the handshake phase. If no IV is required, the length should be set to 0. |
| 11413 | ulSequenceNumber | the current sequence number used for records sent by the client and server respectively |
| 11415 | bIsExport | a boolean value which indicates whether the keys have to be derived for an export version of the protocol. If this value is true (i.e., the keys are exportable) then ulKeySizeInBits is the length of the key in bits before expansion. The length of the key after expansion is determined by the information found in the template sent along with this mechanism during a C_DeriveKey function call (either the CKA_KEY_TYPE or the CKA_VALUE_LEN attribute). |
| 11422 | RandomInfo | client's and server's random data information |
| 11423 | pReturnedKeyMaterial | points to a CK_WTLS_KEY_MAT_OUT structure which receives the handles for the keys generated and the IV |

11425 **CK_WTLS_KEY_MAT_PARAMS_PTR** is a pointer to a **CK_WTLS_KEY_MAT_PARAMS**.

11426 **6.41.3 Pre master secret key generation for RSA key exchange suite**

11427 Pre master secret key generation for the RSA key exchange suite in WTLS denoted
11428 **CKM_WTLS_PRE_MASTER_KEY_GEN**, is a mechanism, which generates a variable length secret key.
11429 It is used to produce the pre master secret key for RSA key exchange suite used in WTLS. This
11430 mechanism returns a handle to the pre master secret key.

11431 It has one parameter, a **CK_BYTE**, which provides the client's WTLS version.

11432 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE** and **CKA_VALUE** attributes to the new
11433 key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may
11434 be specified in the template, or else are assigned default values.

11435 The template sent along with this mechanism during a **C_GenerateKey** call may indicate that the object
11436 class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN**
11437 attribute indicates the length of the pre master secret key.

11438 For this mechanism, the **ulMinKeySize** field of the **CK_MECHANISM_INFO** structure shall indicate 20
11439 bytes.

11440 **6.41.4 Master secret key derivation**

11441 Master secret derivation in WTLS, denoted **CKM_WTLS_MASTER_KEY_DERIVE**, is a mechanism used
11442 to derive a 20 byte generic secret key from variable length secret key. It is used to produce the master
11443 secret key used in WTLS from the pre master secret key. This mechanism returns the value of the client
11444 version, which is built into the pre master secret key as well as a handle to the derived master secret key.

11445 It has a parameter, a **CK_WTLS_MASTER_KEY_DERIVE_PARAMS** structure, which allows for passing
11446 the mechanism type of the digest mechanism to be used as well as the passing of random data to the
11447 token as well as the returning of the protocol version number which is part of the pre master secret key.

11448 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
11449 key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may
11450 be specified in the template, or else are assigned default values.

11451 The template sent along with this mechanism during a **C_DeriveKey** call may indicate that the object
11452 class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN**
11453 attribute has value 20. However, since these facts are all implicit in the mechanism, there is no need to
11454 specify any of them.

11455 This mechanism has the following rules about key sensitivity and extractability:

11456 The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both be
11457 specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some default
11458 value.

11459 If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key will
11460 as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the derived
11461 key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its **CKA_SENSITIVE** attribute.

11462 Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
11463 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_TRUE,
11464 then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the opposite value from its
11465 **CKA_EXTRACTABLE** attribute.

11466 For this mechanism, the **ulMinKeySize** and **ulMaxKeySize** fields of the **CK_MECHANISM_INFO** structure
11467 both indicate 20 bytes.

11468 Note that the **CK_BYTE** pointed to by the **CK_WTLS_MASTER_KEY_DERIVE_PARAMS** structure's
11469 *pVersion* field will be modified by the **C_DeriveKey** call. In particular, when the call returns, this byte will
11470 hold the WTLS version associated with the supplied pre master secret key.

11471 Note that this mechanism is only useable for key exchange suites that use a 20-byte pre master secret
11472 key with an embedded version number. This includes the RSA key exchange suites, but excludes the
11473 Diffie-Hellman and Elliptic Curve Cryptography key exchange suites.

11474 **6.41.5 Master secret key derivation for Diffie-Hellman and Elliptic Curve 11475 Cryptography**

11476 Master secret derivation for Diffie-Hellman and Elliptic Curve Cryptography in WTLS, denoted
11477 **CKM_WTLS_MASTER_KEY_DERIVE_DH_ECC**, is a mechanism used to derive a 20 byte generic
11478 secret key from variable length secret key. It is used to produce the master secret key used in WTLS from
11479 the pre master secret key. This mechanism returns a handle to the derived master secret key.

11480 It has a parameter, a **CK_WTLS_MASTER_KEY_DERIVE_PARAMS** structure, which allows for the
11481 passing of the mechanism type of the digest mechanism to be used as well as random data to the token.
11482 The *pVersion* field of the structure must be set to NULL_PTR since the version number is not embedded
11483 in the pre master secret key as it is for RSA-like key exchange suites.

11484 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
11485 key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may
11486 be specified in the template, or else are assigned default values.
11487 The template sent along with this mechanism during a **C_DeriveKey** call may indicate that the object
11488 class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN**
11489 attribute has value 20. However, since these facts are all implicit in the mechanism, there is no need to
11490 specify any of them.
11491 This mechanism has the following rules about key sensitivity and extractability:
11492 The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both be
11493 specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some default
11494 value.
11495 If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key will
11496 as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the derived
11497 key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its **CKA_SENSITIVE** attribute.
11498 Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
11499 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_TRUE,
11500 then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite* value from its
11501 **CKA_EXTRACTABLE** attribute.
11502 For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK_MECHANISM_INFO** structure
11503 both indicate 20 bytes.
11504 Note that this mechanism is only useable for key exchange suites that do not use a fixed length 20-byte
11505 pre master secret key with an embedded version number. This includes the Diffie-Hellman and Elliptic
11506 Curve Cryptography key exchange suites, but excludes the RSA key exchange suites.

11507 6.41.6 WTLS PRF (pseudorandom function)

11508 PRF (pseudo random function) in WTLS, denoted **CKM_WTLS_PRF**, is a mechanism used to produce a
11509 securely generated pseudo-random output of arbitrary length. The keys it uses are generic secret keys.
11510 It has a parameter, a **CK_WTLS_PRF_PARAMS** structure, which allows for passing the mechanism type
11511 of the digest mechanism to be used, the passing of the input seed and its length, the passing of an
11512 identifying label and its length and the passing of the length of the output to the token and for receiving
11513 the output.
11514 This mechanism produces securely generated pseudo-random output of the length specified in the
11515 parameter.
11516 This mechanism departs from the other key derivation mechanisms in Cryptoki in not using the template
11517 sent along with this mechanism during a **C_DeriveKey** function call, which means the template shall be a
11518 NULL_PTR. For most key-derivation mechanisms, **C_DeriveKey** returns a single key handle as a result
11519 of a successful completion. However, since the **CKM_WTLS_PRF** mechanism returns the requested
11520 number of output bytes in the **CK_WTLS_PRF_PARAMS** structure specified as the mechanism
11521 parameter, the parameter *phKey* passed to **C_DeriveKey** is unnecessary, and should be a NULL_PTR.
11522 If a call to **C_DeriveKey** with this mechanism fails, then no output will be generated.

11523 6.41.7 Server Key and MAC derivation

11524 Server key, MAC and IV derivation in WTLS, denoted
11525 **CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE**, is a mechanism used to derive the appropriate
11526 cryptographic keying material used by a cipher suite from the master secret key and random data. This
11527 mechanism returns the key handles for the keys generated in the process, as well as the IV created.
11528 It has a parameter, a **CK_WTLS_KEY_MAT_PARAMS** structure, which allows for the passing of the
11529 mechanism type of the digest mechanism to be used, random data, the characteristic of the cryptographic
11530 material for the given cipher suite, and a pointer to a structure which receives the handles and IV which
11531 were generated.

11532 This mechanism contributes to the creation of two distinct keys and returns one IV (if an IV is requested
11533 by the caller) back to the caller. The keys are all given an object class of **CKO_SECRET_KEY**.
11534 The MACing key (server write MAC secret) is always given a type of **CKK_GENERIC_SECRET**. It is
11535 flagged as valid for signing, verification and derivation operations.
11536 The other key (server write key) is typed according to information found in the template sent along with
11537 this mechanism during a **C_DeriveKey** function call. By default, it is flagged as valid for encryption,
11538 decryption, and derivation operations.
11539 An IV (server write IV) will be generated and returned if the *ullVSizeInBits* field of the
11540 **CK_WTLS_KEY_MAT_PARAMS** field has a nonzero value. If it is generated, its length in bits will agree
11541 with the value in the *ullVSizeInBits* field
11542 Both keys inherit the values of the **CKA_SENSITIVE**, **CKA_ALWAYS_SENSITIVE**,
11543 **CKA_EXTRACTABLE**, and **CKA_NEVER_EXTRACTABLE** attributes from the base key. The template
11544 provided to **C_DeriveKey** may not specify values for any of these attributes that differ from those held by
11545 the base key.
11546 Note that the **CK_WTLS_KEY_MAT_OUT** structure pointed to by the **CK_WTLS_KEY_MAT_PARAMS**
11547 structure's *pReturnedKeyMaterial* field will be modified by the **C_DeriveKey** call. In particular, the two key
11548 handle fields in the **CK_WTLS_KEY_MAT_OUT** structure will be modified to hold handles to the newly-
11549 created keys; in addition, the buffer pointed to by the **CK_WTLS_KEY_MAT_OUT** structure's *pIV* field will
11550 have the IV returned in them (if an IV is requested by the caller). Therefore, this field must point to a
11551 buffer with sufficient space to hold any IV that will be returned.
11552 This mechanism departs from the other key derivation mechanisms in Cryptoki in its returned information.
11553 For most key-derivation mechanisms, **C_DeriveKey** returns a single key handle as a result of a
11554 successful completion. However, since the **CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE**
11555 mechanism returns all of its key handles in the **CK_WTLS_KEY_MAT_OUT** structure pointed to by the
11556 **CK_WTLS_KEY_MAT_PARAMS** structure specified as the mechanism parameter, the parameter *phKey*
11557 passed to **C_DeriveKey** is unnecessary, and should be a **NONE_PTR**.
11558 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

11559 Client key, MAC and IV derivation in WTLS, denoted **CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE**,
11560 is a mechanism used to derive the appropriate cryptographic keying material used by a cipher suite from
11561 the master secret key and random data. This mechanism returns the key handles for the keys generated
11562 in the process, as well as the IV created.
11563 It has a parameter, a **CK_WTLS_KEY_MAT_PARAMS** structure, which allows for the passing of the
11564 mechanism type of the digest mechanism to be used, random data, the characteristic of the cryptographic
11565 material for the given cipher suite, and a pointer to a structure which receives the handles and IV which
11566 were generated.
11567 This mechanism contributes to the creation of two distinct keys and returns one IV (if an IV is requested
11568 by the caller) back to the caller. The keys are all given an object class of **CKO_SECRET_KEY**.
11569 The MACing key (client write MAC secret) is always given a type of **CKK_GENERIC_SECRET**. It is
11570 flagged as valid for signing, verification and derivation operations.
11571 The other key (client write key) is typed according to information found in the template sent along with this
11572 mechanism during a **C_DeriveKey** function call. By default, it is flagged as valid for encryption,
11573 decryption, and derivation operations.
11574 An IV (client write IV) will be generated and returned if the *ullVSizeInBits* field of the
11575 **CK_WTLS_KEY_MAT_PARAMS** field has a nonzero value. If it is generated, its length in bits will agree
11576 with the value in the *ullVSizeInBits* field
11577 Both keys inherit the values of the **CKA_SENSITIVE**, **CKA_ALWAYS_SENSITIVE**,
11578 **CKA_EXTRACTABLE**, and **CKA_NEVER_EXTRACTABLE** attributes from the base key. The template
11579 provided to **C_DeriveKey** may not specify values for any of these attributes that differ from those held by
11580 the base key.
11581

11582 Note that the **CK_WTLS_KEY_MAT_OUT** structure pointed to by the **CK_WTLS_KEY_MAT_PARAMS**
 11583 structure's *pReturnedKeyMaterial* field will be modified by the **C_DeriveKey** call. In particular, the two key
 11584 handle fields in the **CK_WTLS_KEY_MAT_OUT** structure will be modified to hold handles to the newly-
 11585 created keys; in addition, the buffer pointed to by the **CK_WTLS_KEY_MAT_OUT** structure's *pIV* field will
 11586 have the IV returned in them (if an IV is requested by the caller). Therefore, this field must point to a
 11587 buffer with sufficient space to hold any IV that will be returned.
 11588 This mechanism departs from the other key derivation mechanisms in Cryptoki in its returned information.
 11589 For most key-derivation mechanisms, **C_DeriveKey** returns a single key handle as a result of a
 11590 successful completion. However, since the **CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE** mechanism
 11591 returns all of its key handles in the **CK_WTLS_KEY_MAT_OUT** structure pointed to by the
 11592 **CK_WTLS_KEY_MAT_PARAMS** structure specified as the mechanism parameter, the parameter *phKey*
 11593 passed to **C_DeriveKey** is unnecessary, and should be a **NONE_PTR**.
 11594 If a call to **C_DeriveKey** with this mechanism fails, then *none* of the two keys will be created.

11595 6.42 SP 800-108 Key Derivation

11596 NIST SP800-108 defines three types of key derivation functions (KDF); a Counter Mode KDF, a
 11597 Feedback Mode KDF and a Double Pipeline Mode KDF.

11598 This section defines a unique mechanism for each type of KDF. These mechanisms can be used to
 11599 derive one or more symmetric keys from a single base symmetric key.

11600 The KDFs defined in SP800-108 are all built upon pseudo random functions (PRF). In general terms, the
 11601 PRFs accept two pieces of input; a base key and some input data. The base key is taken from the
 11602 *hBaseKey* parameter to **C_Derive**. The input data is constructed from an iteration variable (internally
 11603 defined by the KDF/PRF) and the data provided in the **CK_PRF_DATA_PARAM** array that is part of the
 11604 mechanism parameter.

11605 *Table 191, SP800-108 Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|-----------------------------------|-------------------|---------------|---------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_SP800_108_COUNTER_KDF | | | | | | | ✓ |
| CKM_SP800_108_FEEDBACK_KDF | | | | | | | ✓ |
| CKM_SP800_108_DOUBLE_PIPELINE_KDF | | | | | | | ✓ |

11606
 11607 For these mechanisms, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO**
 11608 structure specify the minimum and maximum supported base key size in bits. Note, these mechanisms
 11609 support multiple PRF types and key types; as such the values reported by *ulMinKeySize* and
 11610 *ulMaxKeySize* specify the minimum and maximum supported base key size when all PRF and keys types
 11611 are considered. For example, a Cryptoki implementation may support CKK_GENERIC_SECRET keys
 11612 that can be as small as 8-bits in length and therefore *ulMinKeySize* could report 8-bits. However, for an
 11613 AES-CMAC PRF the base key must be of type CKK_AES and must be either 16-bytes, 24-bytes or 32-
 11614 bytes in lengths and therefore the value reported by *ulMinKeySize* could be misleading. Depending on
 11615 the PRF type selected, additional key size restrictions may apply.

11616 6.42.1 Definitions

11617 Mechanisms:

11618 CKM_SP800_108_COUNTER_KDF
 11619 CKM_SP800_108_FEEDBACK_KDF

```
11620      CKM_SP800_108_DOUBLE_PIPELINE_KDF
11621
11622 Data Field Types:
11623     CK_SP800_108_ITERATION_VARIABLE
11624     CK_SP800_108_COUNTER
11625     CK_SP800_108_DKM_LENGTH
11626     CK_SP800_108_BYTE_ARRAY
11627
11628 DKM Length Methods:
11629     CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS
11630     CK_SP800_108_DKM_LENGTH_SUM_OF_SEGMENTS
```

11631 6.42.2 Mechanism Parameters

11632 ♦ CK_SP800_108_PRF_TYPE

11633 The **CK_SP800_108_PRF_TYPE** field of the mechanism parameter is used to specify the type of PRF
11634 that is to be used. It is defined as follows:

```
11635     typedef CK_MECHANISM_TYPE CK_SP800_108_PRF_TYPE;
```

11636 The **CK_SP800_108_PRF_TYPE** field reuses the existing mechanisms definitions. The following table
11637 lists the supported PRF types:

11638 *Table 192, SP800-108 Pseudo Random Functions*

| Pseudo Random Function Identifiers |
|------------------------------------|
| CKM_SHA_1_HMAC |
| CKM_SHA224_HMAC |
| CKM_SHA256_HMAC |
| CKM_SHA384_HMAC |
| CKM_SHA512_HMAC |
| CKM_SHA3_224_HMAC |
| CKM_SHA3_256_HMAC |
| CKM_SHA3_384_HMAC |
| CKM_SHA3_512_HMAC |
| CKM_DES3_CMAC |
| CKM_AES_CMAC |

11639

11640 ♦ CK_PRF_DATA_TYPE

11641 Each mechanism parameter contains an array of **CK_PRF_DATA_PARAM** structures. The
11642 **CK_PRF_DATA_PARAM** structure contains **CK_PRF_DATA_TYPE** field. The **CK_PRF_DATA_TYPE**
11643 field is used to identify the type of data identified by each **CK_PRF_DATA_PARAM** element in the array.
11644 Depending on the type of KDF used, some data field types are mandatory, some data field types are
11645 optional and some data field types are not allowed. These requirements are defined on a per-mechanism
11646 basis in the sections below. The **CK_PRF_DATA_TYPE** is defined as follows:

```
11647     typedef CK ULONG CK_PRF_DATA_TYPE;
```

11648 The following table lists all of the supported data field types:

11649 *Table 193, SP800-108 PRF Data Field Types*

| Data Field Identifier | Description |
|---------------------------------|---|
| CK_SP800_108_ITERATION_VARIABLE | Identifies the iteration variable defined internally by the KDF. |
| CK_SP800_108_COUNTER | Identifies an optional counter value represented as a binary string. Exact formatting of the counter value is defined by the CK_SP800_108_COUNTER_FORMAT structure. The value of the counter is defined by the KDF's internal loop counter. |
| CK_SP800_108_DKM_LENGTH | Identifies the length in bits of the derived keying material (DKM) represented as a binary string. Exact formatting of the length value is defined by the CK_SP800_108_DKM_LENGTH_FORMAT structure. |
| CK_SP800_108_BYTE_ARRAY | Identifies a generic byte array of data. This data type can be used to provide "context", "label", "separator bytes" as well as any other type of encoding information required by the higher level protocol. |

11650

11651 ◆ **CK_PRF_DATA_PARAM**

11652 **CK_PRF_DATA_PARAM** is used to define a segment of input for the PRF. Each mechanism parameter
11653 supports an array of **CK_PRF_DATA_PARAM** structures. The **CK_PRF_DATA_PARAM** is defined as
11654 follows:

```
11655     typedef struct CK_PRF_DATA_PARAM
11656     {
11657         CK_PRF_DATA_TYPE      type;
11658         CK_VOID_PTR           pValue;
11659         CK ULONG               ulValueLen;
11660     } CK_PRF_DATA_PARAM;
11661
11662     typedef CK_PRF_DATA_PARAM CK_PTR CK_PRF_DATA_PARAM_PTR
11663
```

11664 The fields of the **CK_PRF_DATA_PARAM** structure have the following meaning:

11665 type defines the type of data pointed to by pValue

11666 pValue pointer to the data defined by type

11667 ulValueLen size of the data pointed to by pValue

11668 If the *type* field of the **CK_PRF_DATA_PARAM** structure is set to

11669 CK_SP800_108_ITERATION_VARIABLE, then *pValue* must be set the appropriate value for the KDF's
11670 iteration variable type. For the Counter Mode KDF, *pValue* must be assigned a valid
11671 CK_SP800_108_COUNTER_FORMAT_PTR and *ulValueLen* must be set to
11672 sizeof(CK_SP800_108_COUNTER_FORMAT). For all other KDF types, *pValue* must be set to
11673 NULL_PTR and *ulValueLen* must be set to 0.

11674

11675 If the *type* field of the **CK_PRF_DATA_PARAM** structure is set to CK_SP800_108_COUNTER, then
11676 *pValue* must be assigned a valid CK_SP800_108_COUNTER_FORMAT_PTR and *ulValueLen* must be
11677 set to sizeof(CK_SP800_108_COUNTER_FORMAT).

11678

11679 If the *type* field of the **CK_PRF_DATA_PARAM** structure is set to CK_SP800_108_DKM_LENGTH then
 11680 *pValue* must be assigned a valid CK_SP800_108_DKM_LENGTH_FORMAT_PTR and *ulValueLen* must
 11681 be set to sizeof(CK_SP800_108_DKM_LENGTH_FORMAT).

11682

11683 If the *type* field of the **CK_PRF_DATA_PARAM** structure is set to CK_SP800_108_BYTE_ARRAY, then
 11684 *pValue* must be assigned a valid CK_BYTE_PTR value and *ulValueLen* must be set to a non-zero length.

11685 ◆ **CK_SP800_108_COUNTER_FORMAT**

11686 **CK_SP800_108_COUNTER_FORMAT** is used to define the encoding format for a counter value. The
 11687 **CK_SP800_108_COUNTER_FORMAT** is defined as follows:

```
11688     typedef struct CK_SP800_108_COUNTER_FORMAT
11689     {
11690         CK_BBOOL      bLittleEndian;
11691         CK ULONG      ulWidthInBits;
11692     } CK_SP800_108_COUNTER_FORMAT;
11693
11694     typedef CK_SP800_108_COUNTER_FORMAT CK_PTR
11695     CK_SP800_108_COUNTER_FORMAT_PTR
```

11696

11697 The fields of the CK_SP800_108_COUNTER_FORMAT structure have the following meaning:
 11698 **bLittleEndian** defines if the counter should be represented in Big Endian or Little
 11699 Endian format
 11700 **ulWidthInBits** defines the number of bits used to represent the counter value

11701 ◆ **CK_SP800_108_DKM_LENGTH_METHOD**

11702 **CK_SP800_108_DKM_LENGTH_METHOD** is used to define how the DKM length value is calculated.
 11703 The **CK_SP800_108_DKM_LENGTH_METHOD** type is defined as follows:

```
11704     typedef CK ULONG CK_SP800_108_DKM_LENGTH_METHOD;
```

11705 The following table lists all of the supported DKM Length Methods:

11706 *Table 194, SP800-108 DKM Length Methods*

| DKM Length Method Identifier | Description |
|---|---|
| CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS | Specifies that the DKM length should be set to the sum of the length of all keys derived by this invocation of the KDF. |
| CK_SP800_108_DKM_LENGTH_SUM_OF_SEGMENTS | Specifies that the DKM length should be set to the sum of the length of all segments of output produced by the PRF by this invocation of the KDF. |

11707

11708 ◆ **CK_SP800_108_DKM_LENGTH_FORMAT**

11709 **CK_SP800_108_DKM_LENGTH_FORMAT** is used to define the encoding format for the DKM length
 11710 value. The **CK_SP800_108_DKM_LENGTH_FORMAT** is defined as follows:

```
11711     typedef struct CK_SP800_108_DKM_LENGTH_FORMAT
```

```

11712     {
11713         CK_SP800_108_DKM_LENGTH_METHOD    dkmLengthMethod;
11714         CK_BBOOL                         bLittleEndian;
11715         CK ULONG                          ulWidthInBits;
11716     } CK_SP800_108_DKM_LENGTH_FORMAT;
11717
11718     typedef CK_SP800_108_DKM_LENGTH_FORMAT CK_PTR
11719     CK_SP800_108_DKM_LENGTH_FORMAT_PTR
11720
11721 The fields of the CK_SP800_108_DKM_LENGTH_FORMAT structure have the following meaning:
11722     dkmLengthMethod      defines the method used to calculate the DKM length value
11723     bLittleEndian        defines if the DKM length value should be represented in Big
11724                          Endian or Little Endian format
11725     ulWidthInBits       defines the number of bits used to represent the DKM length value

```

◆ CK_DERIVED_KEY

11727 **CK_DERIVED_KEY** is used to define an additional key to be derived as well as provide a
 11728 **CK_OBJECT_HANDLE_PTR** to receive the handle for the derived keys. The **CK_DERIVED_KEY** is
 11729 defined as follows:

```

11730     typedef struct CK_DERIVED_KEY
11731     {
11732         CK_ATTRIBUTE_PTR      pTemplate;
11733         CK ULONG             ulAttributeCount;
11734         CK_OBJECT_HANDLE_PTR phKey;
11735     } CK_DERIVED_KEY;
11736
11737     typedef CK_DERIVED_KEY CK_PTR CK_DERIVED_KEY_PTR
11738

```

11739 The fields of the **CK_DERIVED_KEY** structure have the following meaning:
 11740 pTemplate pointer to a template that defines a key to derive
 11741 ulAttributeCount number of attributes in the template pointed to by pTemplate
 11742 phKey pointer to receive the handle for a derived key

◆ CK_SP800_108_KDF_PARAMS, CK_SP800_108_KDF_PARAMS_PTR

11744 **CK_SP800_108_KDF_PARAMS** is a structure that provides the parameters for the
 11745 **CKM_SP800_108_COUNTER_KDF** and **CKM_SP800_108_DOUBLE_PIPELINE_KDF** mechanisms.

```

11746
11747     typedef struct CK_SP800_108_KDF_PARAMS
11748     {
11749         CK_SP800_108_PRF_TYPE    prfType;
11750         CK ULONG                 ulNumberOfDataParams;
11751         CK_PRF_DATA_PARAM_PTR  pDataParams;
11752         CK ULONG                 ulAdditionalDerivedKeys;
11753         CK_DERIVED_KEY_PTR      pAdditionalDerivedKeys;
11754     } CK_SP800_108_KDF_PARAMS;

```

```

11755
11756     typedef CK_SP800_108_KDF_PARAMS CK_PTR
11757     CK_SP800_108_KDF_PARAMS_PTR;
11758
11759 The fields of the CK_SP800_108_KDF_PARAMS structure have the following meaning:
11760     prfType      type of PRF
11761     ulNumberOfDataParams   number of elements in the array pointed to by pDataParams
11762             pDataParams    an array of CK_PRF_DATA_PARAM structures. The array defines
11763             input parameters that are used to construct the "data" input to the
11764             PRF.
11765     ulAdditionalDerivedKeys  number of additional keys that will be derived and the number of
11766             elements in the array pointed to by pAdditionalDerivedKeys. If
11767             pAdditionalDerivedKeys is set to NULL_PTR, this parameter must
11768             be set to 0.
11769     pAdditionalDerivedKeys  an array of CK_DERIVED_KEY structures. If
11770             ulAdditionalDerivedKeys is set to 0, this parameter must be set to
11771             NULL_PTR

11772 ◆ CK_SP800_108_FEEDBACK_KDF_PARAMS,
11773 CK_SP800_108_FEEDBACK_KDF_PARAMS_PTR

11774 The CK_SP800_108_FEEDBACK_KDF_PARAMS structure provides the parameters for the
11775 CKM_SP800_108_FEEDBACK_KDF mechanism. It is defined as follows:
11776     typedef struct CK_SP800_108_FEEDBACK_KDF_PARAMS
11777     {
11778         CK_SP800_108_PRF_TYPE      prfType;
11779         CK ULONG                  ulNumberOfDataParams;
11780         CK_PRF_DATA_PARAM_PTR    pDataParams;
11781         CK ULONG                  ulIVLen;
11782         CK_BYTE_PTR              pIV;
11783         CK ULONG                  ulAdditionalDerivedKeys;
11784         CK_DERIVED_KEY_PTR       pAdditionalDerivedKeys;
11785     } CK_SP800_108_FEEDBACK_KDF_PARAMS;
11786
11787     typedef CK_SP800_108_FEEDBACK_KDF_PARAMS CK_PTR
11788     CK_SP800_108_FEEDBACK_KDF_PARAMS_PTR;
11789
11790 The fields of the CK_SP800_108_FEEDBACK_KDF_PARAMS structure have the following meaning:
11791     prfType      type of PRF
11792     ulNumberOfDataParams   number of elements in the array pointed to by pDataParams
11793             pDataParams    an array of CK_PRF_DATA_PARAM structures. The array defines
11794             input parameters that are used to construct the "data" input to the
11795             PRF.
11796             ullIVLen    the length in bytes of the IV. If pIV is set to NULL_PTR, this
11797             parameter must be set to 0.
11798             pIV        an array of bytes to be used as the IV for the feedback mode KDF.
11799             This parameter is optional and can be set to NULL_PTR. If ullIVLen
11800             is set to 0, this parameter must be set to NULL_PTR.

```

| | | |
|-------|-------------------------|---|
| 11801 | ulAdditionalDerivedKeys | number of additional keys that will be derived and the number of elements in the array pointed to by pAdditionalDerivedKeys. If pAdditionalDerivedKeys is set to NULL_PTR, this parameter must be set to 0. |
| 11805 | pAdditionalDerivedKeys | an array of CK_DERIVED_KEY structures. If ulAdditionalDerivedKeys is set to 0, this parameter must be set to NULL_PTR. |

6.42.3 Counter Mode KDF

The SP800-108 Counter Mode KDF mechanism, denoted **CKM_SP800_108_COUNTER_KDF**, represents the KDF defined SP800-108 section 5.1. **CKM_SP800_108_COUNTER_KDF** is a mechanism for deriving one or more symmetric keys from a symmetric base key.

It has a parameter, a **CK_SP800_108_KDF_PARAMS** structure.

The following table lists the data field types that are supported for this KDF type and their meaning:

Table 195, Counter 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 for this KDF type is a counter.</p> <p>Exact formatting of the counter value is defined by the CK_SP800_108_COUNTER_FORMAT structure.</p> |
| CK_SP800_108_COUNTER | This data field type is invalid for this KDF type. |
| CK_SP800_108_DKM_LENGTH | <p>This data field type is optional.</p> <p>This data field type identifies the location of the DKM length in the constructed PRF input data.</p> <p>Exact formatting of the DKM length is defined by the CK_SP800_108_DKM_LENGTH_FORMAT structure.</p> <p>If specified, only one instance of this type may be specified.</p> |
| CK_SP800_108_BYTE_ARRAY | <p>This data field type is optional.</p> <p>This data field type identifies the location and value of a byte array of data in the constructed PRF input data.</p> <p>This standard does not restrict the number of instances of this data type.</p> |

SP800-108 limits the amount of derived keying material that can be produced by a Counter Mode KDF by limiting the internal loop counter to $(2^r - 1)$, where "r" is the number of bits used to represent the counter. Therefore the maximum number of bits that can be produced is $(2^r - 1)h$, where "h" is the length in bits of the output of the selected PRF.

6.42.4 Feedback Mode KDF

The SP800-108 Feedback Mode KDF mechanism, denoted **CKM_SP800_108_FEEDBACK_KDF**, represents the KDF defined SP800-108 section 5.2. **CKM_SP800_108_FEEDBACK_KDF** is a mechanism for deriving one or more symmetric keys from a symmetric base key.

It has a parameter, a **CK_SP800_108_FEEDBACK_KDF_PARAMS** structure.

The following table lists the data field types that are supported for this KDF type and their meaning:

Table 196, 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 SP800-108.</p> <p>The size, format and value of this data input is defined by the internal KDF structure and PRF output.</p> <p>Exact formatting of the counter value is defined by the CK_SP800_108_COUNTER_FORMAT structure.</p> |
| CK_SP800_108_COUNTER | <p>This data field type is optional.</p> <p>This data field type identifies the location of the counter in the constructed PRF input data.</p> <p>Exact formatting of the counter value is defined by the CK_SP800_108_COUNTER_FORMAT structure.</p> <p>If specified, only one instance of this type may be specified.</p> |
| CK_SP800_108_DKM_LENGTH | <p>This data field type is optional.</p> <p>This data field type identifies the location of the DKM length in the constructed PRF input data.</p> <p>Exact formatting of the DKM length is defined by the CK_SP800_108_DKM_LENGTH_FORMAT structure.</p> <p>If specified, only one instance of this type may be specified.</p> |
| CK_SP800_108_BYTE_ARRAY | <p>This data field type is optional.</p> <p>This data field type identifies the location and value of a byte array of data in the constructed PRF input data.</p> <p>This standard does not restrict the number of instances of this data type.</p> |

11827

11828 SP800-108 limits the amount of derived keying material that can be produced by a Feedback Mode KDF
 11829 by limiting the internal loop counter to $(2^{32}-1)$. Therefore the maximum number of bits that can be
 11830 produced is $(2^{32}-1)h$, where "h" is the length in bits of the output of the selected PRF.

11831 6.42.5 Double Pipeline Mode KDF

11832 The SP800-108 Double Pipeline Mode KDF mechanism, denoted
 11833 **CKM_SP800_108_DOUBLE_PIPELINE_KDF**, represents the KDF defined SP800-108 section 5.3.
 11834 **CKM_SP800_108_DOUBLE_PIPELINE_KDF** is a mechanism for deriving one or more symmetric keys
 11835 from a symmetric base key.

11836 It has a parameter, a CK_SP800_108_KDF_PARAMS structure.

11837 The following table lists the data field types that are supported for this KDF type and their meaning:

11838 *Table 197, Double Pipeline Mode data field requirements*

| Data Field Identifier | Description |
|---------------------------------|--|
| CK_SP800_108_ITERATION_VARIABLE | <p>This data field type is mandatory.</p> <p>This data field type identifies the location of the iteration variable in the constructed PRF input data.</p> <p>The iteration variable is defined as A(i) in section 5.3 of SP800-108.</p> |

| | |
|-------------------------|---|
| | <p>The size, format and value of this data input is defined by the internal KDF structure and PRF output.</p> <p>Exact formatting of the counter value is defined by the CK_SP800_108_COUNTER_FORMAT structure.</p> |
| CK_SP800_108_COUNTER | <p>This data field type is optional.</p> <p>This data field type identifies the location of the counter in the constructed PRF input data.</p> <p>Exact formatting of the counter value is defined by the CK_SP800_108_COUNTER_FORMAT structure.</p> <p>If specified, only one instance of this type may be specified.</p> |
| CK_SP800_108_DKM_LENGTH | <p>This data field type is optional.</p> <p>This data field type identifies the location of the DKM length in the constructed PRF input data.</p> <p>Exact formatting of the DKM length is defined by the CK_SP800_108_DKM_LENGTH_FORMAT structure.</p> <p>If specified, only one instance of this type may be specified.</p> |
| CK_SP800_108_BYTE_ARRAY | <p>This data field type is optional.</p> <p>This data field type identifies the location and value of a byte array of data in the constructed PRF input data.</p> <p>This standard does not restrict the number of instances of this data type.</p> |

11839

11840 SP800-108 limits the amount of derived keying material that can be produced by a Double-Pipeline Mode
 11841 KDF by limiting the internal loop counter to $(2^{32}-1)$. Therefore the maximum number of bits that can be
 11842 produced is $(2^{32}-1)h$, where "h" is the length in bits of the output of the selected PRF.

11843 The Double Pipeline KDF requires an internal IV value. The IV is constructed using the same method
 11844 used to construct the PRF input data; the data/values identified by the array of CK_PRF_DATA_PARAM
 11845 structures are concatenated in to a byte array that is used as the IV. As shown in SP800-108 section 5.3,
 11846 the CK_SP800_108_ITERATION_VARIABLE and CK_SP800_108_COUNTER data field types are not
 11847 included in IV construction process. All other data field types are included in the construction process.

11848 6.42.6 Deriving Additional Keys

11849 The KDFs defined in this section can be used to derive more than one symmetric key from the base key.
 11850 The **C_Derive** function accepts one CK_ATTRIBUTE_PTR to define a single derived key and one
 11851 CK_OBJECT_HANDLE_PTR to receive the handle for the derived key.

11852 To derive additional keys, the mechanism parameter structure can be filled in with one or more
 11853 CK_DERIVED_KEY structures. Each structure contains a CK_ATTRIBUTE_PTR to define a derived key
 11854 and a CK_OBJECT_HANDLE_PTR to receive the handle for the additional derived keys. The key
 11855 defined by the **C_Derive** function parameters is always derived before the keys defined by the
 11856 CK_DERIVED_KEY array that is part of the mechanism parameter. The additional keys that are defined
 11857 by the CK_DERIVED_KEY array are derived in the order they are defined in the array. That is to say that
 11858 the derived keying material produced by the KDF is processed from left to right, and bytes are assigned
 11859 first to the key defined by the **C_Derive** function parameters, and then bytes are assigned to the keys that
 11860 are defined by the CK_DERIVED_KEY array in the order they are defined in the array.

11861 Each internal iteration of a KDF produces a unique segment of PRF output. Sometimes, a single iteration
 11862 will produce enough keying material for the key being derived. Other times, additional internal iterations
 11863 are performed to produce multiple segments which are concatenated together to produce enough keying
 11864 material for the derived key(s).

11865 When deriving multiple keys, no key can be created using part of a segment that was used for another
 11866 key. All keys must be created from disjoint segments. For example, if the parameters are defined such

11867 that a 48-byte key (defined by the **C_Derive** function parameters) and a 16-byte key (defined by the
11868 content of CK_DERIVED_KEY) are to be derived using **CKM_SHA256_HMAC** as a PRF, three internal
11869 iterations of the KDF will be performed and three segments of PRF output will be produced. The first
11870 segment and half of the second segment will be used to create the 48-byte key and the third segment will
11871 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 |

11872
11873 In the above example, if the CK_SP800_108_DKM_LENGTH data field type is specified with method
11874 CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS, then the DKM length value will be 512 bits. If the
11875 CK_SP800_108_DKM_LENGTH data field type is specified with method
11876 CK_SP800_108_DKM_LENGTH_SUM_OF_SEGMENTS, then the DKM length value will be 768 bits.

11877 When deriving multiple keys, if any of the keys cannot be derived for any reason, none of the keys shall
11878 be derived. If the failure was caused by the content of a specific key's template (ie the template defined
11879 by the content of *pTemplate*), the corresponding *phKey* value will be set to CK_INVALID_HANDLE to
11880 identify the offending template.

11881 **6.42.7 Key Derivation Attribute Rules**

11882 The **CKM_SP800_108_COUNTER_KDF**, **CKM_SP800_108_FEEDBACK_KDF** and
11883 **CKM_SP800_108_DOUBLE_PIPELINE_KDF** mechanisms have the following rules about key sensitivity
11884 and extractability:

- 11885 • The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key(s) can
11886 both be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on
11887 some default value.
- 11888 • If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key
11889 will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the
11890 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
11891 **CKA_SENSITIVE** attribute.
- 11892 • Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
11893 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
11894 CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
11895 value from its **CKA_EXTRACTABLE** attribute.

11896 **6.42.8 Constructing PRF Input Data**

11897 SP800-108 defines the PRF input data for each KDF at a high level using terms like "label", "context",
11898 "separator", "counter"...etc. The value, formatting and order of the input data is not strictly defined by
11899 SP800-108, instead it is described as being defined by the "encoding scheme".

11900 To support any encoding scheme, these mechanisms construct the PRF input data from the array of
11901 CK_PRF_DATA_PARAM structures in the mechanism parameter. All of the values defined by the
11902 CK_PRF_DATA_PARAM array are concatenated in the order they are defined and passed in to the PRF
11903 as the data parameter.

11904 **6.42.8.1 Sample Counter Mode KDF**

11905 SP800-108 section 5.1 outlines a sample Counter Mode KDF which defines the following PRF input:

11906 $\text{PRF}(K, [i]_2 \parallel \text{Label} \parallel 0x00 \parallel \text{Context} \parallel [L]_2)$

11907 Section 5.1 does not define the number of bits used to represent the counter (the "r" value) or the DKM
11908 length (the "L" value), so 16-bits is assumed for both cases. The following sample code shows how to
11909 define this PRF input data using an array of CK_PRF_DATA_PARAM structures.

```

11910 #define DIM(a) (sizeof((a))/sizeof((a)[0]))
11911
11912 CK_OBJECT_HANDLE hBaseKey;
11913 CK_OBJECT_HANDLE hDerivedKey;
11914 CK_ATTRIBUTE derivedKeyTemplate = { ... };
11915
11916 CK_BYTE baLabel[] = {0xde, 0xad, 0xbe , 0xef};
11917 CK ULONG ulLabelLen = sizeof(baLabel);
11918 CK_BYTE baContext[] = {0xfe, 0xed, 0xbe , 0xef};
11919 CK ULONG ulContextLen = sizeof(baContext);
11920
11921 CK_SP800_108_COUNTER_FORMAT counterFormat = {0, 16};
11922 CK_SP800_108_DKM_LENGTH_FORMAT dkmFormat
11923     = {CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS, 0, 16};
11924
11925 CK_PRF_DATA_PARAM dataParams[] =
11926 {
11927     { CK_SP800_108_ITERATION_VARIABLE,
11928         &counterFormat, sizeof(counterFormat) },
11929     { CK_SP800_108_BYTE_ARRAY, baLabel, ulLabelLen },
11930     { CK_SP800_108_BYTE_ARRAY, {0x00}, 1 },
11931     { CK_SP800_108_BYTE_ARRAY, baContext, ulContextLen },
11932     { CK_SP800_108_DKM_LENGTH, dkmFormat, sizeof(dkmFormat) }
11933 };
11934
11935 CK_SP800_108_KDF_PARAMS kdfParams =
11936 {
11937     CKM_AES_CMAC,
11938     DIM(dataParams),
11939     &dataParams,
11940     0, /* no addition derived keys */
11941     NULL /* no addition derived keys */
11942 };
11943
11944 CK_MECHANISM = mechanism
11945 {
11946     CKM_SP800_108_COUNTER_KDF,
11947     &kdfParams,
11948     sizeof(kdfParams)
11949 };
11950
11951 hBaseKey = GetBaseKeyHandle(.....);
11952
11953 rv = C_DeriveKey(
11954     hSession,
11955     &mechanism,
11956     hBaseKey,
11957     &derivedKeyTemplate,
11958     DIM(derivedKeyTemplate),
11959     &hDerivedKey);

```

6.42.8.2 Sample SCP03 Counter Mode KDF

11961 The SCP03 standard defines a variation of a counter mode KDF which defines the following PRF input:
11962 $\text{PRF}(K_1, \text{Label} \parallel 0x00 \parallel [L]_2 \parallel [i]_2 \parallel \text{Context})$

```

11963 SCP03 defines the number of bits used to represent the counter (the "r" value) and number of bits used to
11964 represent the DKM length (the "L" value) as 16-bits. The following sample code shows how to define this
11965 PRF input data using an array of CK_PRF_DATA_PARAM structures.
11966 #define DIM(a) (sizeof((a))/sizeof((a)[0]))
11967
11968     CK_OBJECT_HANDLE hBaseKey;
11969     CK_OBJECT_HANDLE hDerivedKey;
11970     CK_ATTRIBUTE derivedKeyTemplate = { ... };
11971
11972     CK_BYTE baLabel[] = {0xde, 0xad, 0xbe , 0xef};
11973     CK ULONG ulLabelLen = sizeof(baLabel);
11974     CK_BYTE baContext[] = {0xfe, 0xed, 0xbe , 0xef};
11975     CK ULONG ulContextLen = sizeof(baContext);
11976
11977     CK_SP800_108_COUNTER_FORMAT counterFormat = {0, 16};
11978     CK_SP800_108_DKM_LENGTH_FORMAT dkmFormat
11979         = {CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS, 0, 16};
11980
11981     CK_PRF_DATA_PARAM dataParams[] =
11982     {
11983         { CK_SP800_108_BYTE_ARRAY, baLabel, ulLabelLen },
11984         { CK_SP800_108_BYTE_ARRAY, {0x00}, 1 },
11985         { CK_SP800_108_DKM_LENGTH, dkmFormat, sizeof(dkmFormat) },
11986         { CK_SP800_108_ITERATION_VARIABLE,
11987             &counterFormat, sizeof(counterFormat) },
11988         { CK_SP800_108_BYTE_ARRAY, baContext, ulContextLen }
11989     };
11990
11991     CK_SP800_108_KDF_PARAMS kdfParams =
11992     {
11993         CKM_AES_CMAC,
11994         DIM(dataParams),
11995         &dataParams,
11996         0, /* no addition derived keys */
11997         NULL /* no addition derived keys */
11998     };
11999
12000     CK_MECHANISM = mechanism
12001     {
12002         CKM_SP800_108_COUNTER_KDF,
12003         &kdfParams,
12004         sizeof(kdfParams)
12005     };
12006
12007     hBaseKey = GetBaseKeyHandle(.....);
12008
12009     rv = C_DeriveKey(
12010         hSession,
12011         &mechanism,
12012         hBaseKey,
12013         &derivedKeyTemplate,
12014         DIM(derivedKeyTemplate),
12015         &hDerivedKey);

```

6.42.8.3 Sample Feedback Mode KDF

SP800-108 section 5.2 outlines a sample Feedback Mode KDF which defines the following PRF input:

12018 PRF ($K_I, K(i-1) \{|| [i]_2 \}|| Label || 0x00 || Context || [L]_2$)

12019 Section 5.2 does not define the number of bits used to represent the counter (the “r” value) or the DKM
12020 length (the “L” value), so 16-bits is assumed for both cases. The counter is defined as being optional and
12021 is included in this example. The following sample code shows how to define this PRF input data using an
12022 array of CK_PRF_DATA_PARAM structures.

```
12023 #define DIM(a) (sizeof((a))/sizeof((a)[0]))  
12024  
12025     CK_OBJECT_HANDLE hBaseKey;  
12026     CK_OBJECT_HANDLE hDerivedKey;  
12027     CK_ATTRIBUTE derivedKeyTemplate = { ... };  
12028  
12029     CK_BYTE baFeedbackIV[] = {0x01, 0x02, 0x03, 0x04};  
12030     CK ULONG ulFeedbackIVLen = sizeof(baFeedbackIV);  
12031     CK_BYTE baLabel[] = {0xde, 0xad, 0xbe, 0xef};  
12032     CK ULONG ulLabelLen = sizeof(baLabel);  
12033     CK_BYTE baContext[] = {0xfe, 0xed, 0xbe, 0xef};  
12034     CK ULONG ulContextLen = sizeof(baContext);  
12035  
12036     CK_SP800_108_COUNTER_FORMAT counterFormat = {0, 16};  
12037     CK_SP800_108_DKM_LENGTH_FORMAT dkmFormat  
12038         = {CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS, 0, 16};  
12039  
12040     CK_PRF_DATA_PARAM dataParams[] =  
12041     {  
12042         { CK_SP800_108_ITERATION_VARIABLE,  
12043             &counterFormat, sizeof(counterFormat) },  
12044         { CK_SP800_108_BYTE_ARRAY, baLabel, ulLabelLen },  
12045         { CK_SP800_108_BYTE_ARRAY, {0x00}, 1 },  
12046         { CK_SP800_108_BYTE_ARRAY, baContext, ulContextLen },  
12047         { CK_SP800_108_DKM_LENGTH, dkmFormat, sizeof(dkmFormat) }  
12048     };  
12049  
12050     CK_SP800_108_FEEDBACK_KDF_PARAMS kdfParams =  
12051     {  
12052         CKM_AES_CMAC,  
12053         DIM(dataParams),  
12054         &dataParams,  
12055         ulFeedbackIVLen,  
12056         baFeedbackIV,  
12057         0, /* no addition derived keys */  
12058         NULL /* no addition derived keys */  
12059     };  
12060  
12061     CK_MECHANISM = mechanism  
12062     {  
12063         CKM_SP800_108_FEEDBACK_KDF,  
12064         &kdfParams,  
12065         sizeof(kdfParams)  
12066     };  
12067  
12068     hBaseKey = GetBaseKeyHandle(. . . . .);  
12069  
12070     rv = C_DeriveKey(  
12071         hSession,  
12072         &mechanism,  
12073         hBaseKey,
```

```
12074     &derivedKeyTemplate,  
12075     DIM(derivedKeyTemplate),  
12076     &hDerivedKey);
```

6.42.8.4 Sample Double-Pipeline Mode KDF

12078 SP800-108 section 5.3 outlines a sample Double-Pipeline Mode KDF which defines the two following
12079 PRF inputs:

```
12080     PRF (KI, A(i-1))  
12081     PRF (KI, K(i-1) {|| [i]2 }|| Label || 0x00 || Context || [L]2)
```

12082 Section 5.3 does not define the number of bits used to represent the counter (the “r” value) or the DKM
12083 length (the “L” value), so 16-bits is assumed for both cases. The counter is defined as being optional so it
12084 is left out in this example. The following sample code shows how to define this PRF input data using an
12085 array of CK_PRF_DATA_PARAM structures.

```
12086 #define DIM(a) (sizeof((a))/sizeof((a)[0]))  
12087  
12088     CK_OBJECT_HANDLE hBaseKey;  
12089     CK_OBJECT_HANDLE hDerivedKey;  
12090     CK_ATTRIBUTE derivedKeyTemplate = { ... };  
12091  
12092     CK_BYTE baLabel[] = {0xde, 0xad, 0xbe, 0xef};  
12093     CK_ULONG ulLabelLen = sizeof(baLabel);  
12094     CK_BYTE baContext[] = {0xfe, 0xed, 0xbe, 0xef};  
12095     CK_ULONG ulContextLen = sizeof(baContext);  
12096  
12097     CK_SP800_108_DKM_LENGTH_FORMAT dkmFormat  
12098         = {CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS, 0, 16};  
12099  
12100     CK_PRF_DATA_PARAM dataParams[] =  
12101     {  
12102         { CK_SP800_108_BYTE_ARRAY, baLabel, ulLabelLen },  
12103         { CK_SP800_108_BYTE_ARRAY, {0x00}, 1 },  
12104         { CK_SP800_108_BYTE_ARRAY, baContext, ulContextLen },  
12105         { CK_SP800_108_DKM_LENGTH, dkmFormat, sizeof(dkmFormat) }  
12106     };  
12107  
12108     CK_SP800_108_KDF_PARAMS kdfParams =  
12109     {  
12110         CKM_AES_CMAC,  
12111         DIM(dataParams),  
12112         &dataParams,  
12113         0, /* no addition derived keys */  
12114         NULL /* no addition derived keys */  
12115     };  
12116  
12117     CK_MECHANISM = mechanism  
12118     {  
12119         CKM_SP800_108_DOUBLE_PIPELINE_KDF,  
12120         &kdfParams,  
12121         sizeof(kdfParams)  
12122     };  
12123  
12124     hBaseKey = GetBaseKeyHandle(. . . . .);  
12125  
12126     rv = C_DeriveKey(  
12127         hSession,
```

```

12128     &mechanism,
12129     hBaseKey,
12130     &derivedKeyTemplate,
12131     DIM(derivedKeyTemplate),
12132     &hDerivedKey);

```

6.43 Miscellaneous simple key derivation mechanisms

Table 198, *Miscellaneous simple key derivation Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|-------------------------------|-------------------|---------------|----------------------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_CONCATENATE_BASE_AND_KEY | | | | | | | ✓ |
| CKM_CONCATENATE_BASE_AND_DATA | | | | | | | ✓ |
| CKM_CONCATENATE_DATA_AND_BASE | | | | | | | ✓ |
| CKM_XOR_BASE_AND_DATA | | | | | | | ✓ |
| CKM_EXTRACT_KEY_FROM_KEY | | | | | | | ✓ |

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

```

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**.

12158 ◆ **CK_EXTRACT_PARAMS; CK_EXTRACT_PARAMS_PTR**

12159 **CK_EXTRACT_PARAMS** provides the parameter to the **CKM_EXTRACT_KEY_FROM_KEY**
12160 mechanism. It specifies which bit of the base key should be used as the first bit of the derived key. It is
12161 defined as follows:

12162 **typedef CK ULONG CK_EXTRACT_PARAMS;**

12164 **CK_EXTRACT_PARAMS_PTR** is a pointer to a **CK_EXTRACT_PARAMS**.

12165 **6.43.3 Concatenation of a base key and another key**

12166 This mechanism, denoted **CKM_CONCATENATE_BASE_AND_KEY**, derives a secret key from the
12167 concatenation of two existing secret keys. The two keys are specified by handles; the values of the keys
12168 specified are concatenated together in a buffer.

12169 This mechanism takes a parameter, a **CK_OBJECT_HANDLE**. This handle produces the key value
12170 information which is appended to the end of the base key's value information (the base key is the key
12171 whose handle is supplied as an argument to **C_DeriveKey**).

12172 For example, if the value of the base key is 0x01234567, and the value of the other key is 0x89ABCDEF,
12173 then the value of the derived key will be taken from a buffer containing the string 0x0123456789ABCDEF.

- 12174 • If no length or key type is provided in the template, then the key produced by this mechanism will be a
12175 generic secret key. Its length will be equal to the sum of the lengths of the values of the two original
12176 keys.
- 12177 • If no key type is provided in the template, but a length is, then the key produced by this mechanism
12178 will be a generic secret key of the specified length.
- 12179 • If no length is provided in the template, but a key type is, then that key type must have a well-defined
12180 length. If it does, then the key produced by this mechanism will be of the type specified in the
12181 template. If it doesn't, an error will be returned.
- 12182 • If both a key type and a length are provided in the template, the length must be compatible with that
12183 key type. The key produced by this mechanism will be of the specified type and length.

12184 If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set
12185 properly.

12186 If the requested type of key requires more bytes than are available by concatenating the two original keys'
12187 values, an error is generated.

12188 This mechanism has the following rules about key sensitivity and extractability:

- 12189 • If either of the two original keys has its **CKA_SENSITIVE** attribute set to CK_TRUE, so does the
12190 derived key. If not, then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied
12191 template or from a default value.
- 12192 • Similarly, if either of the two original keys has its **CKA_EXTRACTABLE** attribute set to CK_FALSE,
12193 so does the derived key. If not, then the derived key's **CKA_EXTRACTABLE** attribute is set either
12194 from the supplied template or from a default value.
- 12195 • The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to CK_TRUE if and only if both of the
12196 original keys have their **CKA_ALWAYS_SENSITIVE** attributes set to CK_TRUE.
- 12197 • Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to CK_TRUE if and only if
12198 both of the original keys have their **CKA_NEVER_EXTRACTABLE** attributes set to CK_TRUE.

12199 **6.43.4 Concatenation of a base key and data**

12200 This mechanism, denoted **CKM_CONCATENATE_BASE_AND_DATA**, derives a secret key by
12201 concatenating data onto the end of a specified secret key.

12202 This mechanism takes a parameter, a **CK_KEY_DERIVATION_STRING_DATA** structure, which
12203 specifies the length and value of the data which will be appended to the base key to derive another key.
12204 For example, if the value of the base key is 0x01234567, and the value of the data is 0x89ABCDEF, then
12205 the value of the derived key will be taken from a buffer containing the string 0x0123456789ABCDEF.
12206 • If no length or key type is provided in the template, then the key produced by this mechanism will be a
12207 generic secret key. Its length will be equal to the sum of the lengths of the value of the original key
12208 and the data.
12209 • If no key type is provided in the template, but a length is, then the key produced by this mechanism
12210 will be a generic secret key of the specified length.
12211 • If no length is provided in the template, but a key type is, then that key type must have a well-defined
12212 length. If it does, then the key produced by this mechanism will be of the type specified in the
12213 template. If it doesn't, an error will be returned.
12214 • If both a key type and a length are provided in the template, the length must be compatible with that
12215 key type. The key produced by this mechanism will be of the specified type and length.
12216 If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set
12217 properly.
12218 If the requested type of key requires more bytes than are available by concatenating the original key's
12219 value and the data, an error is generated.
12220 This mechanism has the following rules about key sensitivity and extractability:
12221 • If the base key has its **CKA_SENSITIVE** attribute set to CK_TRUE, so does the derived key. If not,
12222 then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied template or from a
12223 default value.
12224 • Similarly, if the base key has its **CKA_EXTRACTABLE** attribute set to CK_FALSE, so does the
12225 derived key. If not, then the derived key's **CKA_EXTRACTABLE** attribute is set either from the
12226 supplied template or from a default value.
12227 • The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to CK_TRUE if and only if the base
12228 key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE.
12229 • Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to CK_TRUE if and only if
12230 the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_TRUE.

6.43.5 Concatenation of data and a base key

12231 This mechanism, denoted **CKM_CONCATENATE_DATA_AND_BASE**, derives a secret key by
12232 prepending data to the start of a specified secret key.
12233 This mechanism takes a parameter, a **CK_KEY_DERIVATION_STRING_DATA** structure, which
12234 specifies the length and value of the data which will be prepended to the base key to derive another key.
12235 For example, if the value of the base key is 0x01234567, and the value of the data is 0x89ABCDEF, then
12236 the value of the derived key will be taken from a buffer containing the string 0x89ABCDEF01234567.
12237 • If no length or key type is provided in the template, then the key produced by this mechanism will be a
12238 generic secret key. Its length will be equal to the sum of the lengths of the data and the value of the
12239 original key.
12240 • If no key type is provided in the template, but a length is, then the key produced by this mechanism
12241 will be a generic secret key of the specified length.
12242 • If no length is provided in the template, but a key type is, then that key type must have a well-defined
12243 length. If it does, then the key produced by this mechanism will be of the type specified in the
12244 template. If it doesn't, an error will be returned.
12245 • If both a key type and a length are provided in the template, the length must be compatible with that
12246 key type. The key produced by this mechanism will be of the specified type and length.
12247 If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set
12248 properly.

- 12250 If the requested type of key requires more bytes than are available by concatenating the data and the
12251 original key's value, an error is generated.
- 12252 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.

12263 **6.43.6 XORing of a key and data**

12264 XORing key derivation, denoted **CKM_XOR_BASE_AND_DATA**, is a mechanism which provides the
12265 capability of deriving a secret key by performing a bit XORing of a key pointed to by a base key handle
12266 and some data.

12267 This mechanism takes a parameter, a **CK_KEY_DERIVATION_STRING_DATA** structure, which
12268 specifies the data with which to XOR the original key's value.

12269 For example, if the value of the base key is 0x01234567, and the value of the data is 0x89ABCDEF, then
12270 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.

12271 If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set
12272 properly.

12273 If the requested type of key requires more bytes than are available by taking the shorter of the data and
12274 the original key's value, an error is generated.

12275 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.

12296 6.43.7 Extraction of one key from another key

12297 Extraction of one key from another key, denoted **CKM_EXTRACT_KEY_FROM_KEY**, is a mechanism
12298 which provides the capability of creating one secret key from the bits of another secret key.

12299 This mechanism has a parameter, a CK_EXTRACT_PARAMS, which specifies which bit of the original
12300 key should be used as the first bit of the newly-derived key.

12301 We give an example of how this mechanism works. Suppose a token has a secret key with the 4-byte
12302 value 0x329F84A9. We will derive a 2-byte secret key from this key, starting at bit position 21 (i.e., the
12303 value of the parameter to the CKM_EXTRACT_KEY_FROM_KEY mechanism is 21).

- 12304 1. We write the key's value in binary: 0011 0010 1001 1111 1000 0100 1010 1001. We regard this
12305 binary string as holding the 32 bits of the key, labeled as b0, b1, ..., b31.
- 12306 2. We then extract 16 consecutive bits (i.e., 2 bytes) from this binary string, starting at bit b21. We
12307 obtain the binary string 1001 0101 0010 0110.
- 12308 3. The value of the new key is thus 0x9526.

12309 Note that when constructing the value of the derived key, it is permissible to wrap around the end of the
12310 binary string representing the original key's value.

12311 If the original key used in this process is sensitive, then the derived key must also be sensitive for the
12312 derivation to succeed.

- 12313 • If no length or key type is provided in the template, then an error will be returned.
- 12314 • If no key type is provided in the template, but a length is, then the key produced by this mechanism
12315 will be a generic secret key of the specified length.
- 12316 • If no length is provided in the template, but a key type is, then that key type must have a well-defined
12317 length. If it does, then the key produced by this mechanism will be of the type specified in the
12318 template. If it doesn't, an error will be returned.
- 12319 • If both a key type and a length are provided in the template, the length must be compatible with that
12320 key type. The key produced by this mechanism will be of the specified type and length.

12321 If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set
12322 properly.

12323 If the requested type of key requires more bytes than the original key has, an error is generated.

12324 This mechanism has the following rules about key sensitivity and extractability:

- 12325 • If the base key has its **CKA_SENSITIVE** attribute set to CK_TRUE, so does the derived key. If not,
12326 then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied template or from a
12327 default value.
- 12328 • Similarly, if the base key has its **CKA_EXTRACTABLE** attribute set to CK_FALSE, so does the
12329 derived key. If not, then the derived key's **CKA_EXTRACTABLE** attribute is set either from the
12330 supplied template or from a default value.
- 12331 • The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to CK_TRUE if and only if the base
12332 key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE.
- 12333 • Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to CK_TRUE if and only if
12334 the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_TRUE.

12335 6.44 CMS

12336 *Table 199, CMS Mechanisms vs. Functions*

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

12337 **6.44.1 Definitions**

12338 Mechanisms:

12339 CKM_CMS_SIG

12340 **6.44.2 CMS Signature Mechanism Objects**

12341 These objects provide information relating to the CKM_CMS_SIG mechanism. CKM_CMS_SIG
 12342 mechanism object attributes represent information about supported CMS signature attributes in the token.
 12343 They are only present on tokens supporting the **CKM_CMS_SIG** mechanism, but must be present on
 12344 those tokens.

12345 *Table 200, CMS Signature Mechanism Object Attributes*

| Attribute | Data type | Meaning |
|------------------------------|------------|---|
| CKA_REQUIRED_CMS_ATTRIBUTES | Byte array | Attributes the token always will include in the set of CMS signed attributes |
| CKA_DEFAULT_CMS_ATTRIBUTES | Byte array | Attributes the token will include in the set of CMS signed attributes in the absence of any attributes specified by the application |
| CKA_SUPPORTED_CMS_ATTRIBUTES | Byte array | Attributes the token may include in the set of CMS signed attributes upon request by the application |

12346 The contents of each byte array will be a DER-encoded list of CMS **Attributes** with optional accompanying
 12347 values. Any attributes in the list shall be identified with its object identifier, and any values shall be DER-
 12348 encoded. The list of attributes is defined in ASN.1 as:

```
12349     Attributes ::= SET SIZE (1..MAX) OF Attribute
12350     Attribute ::= SEQUENCE {
12351         attrType   OBJECT IDENTIFIER,
12352         attrValues SET OF ANY DEFINED BY OBJECT IDENTIFIER
12353             OPTIONAL
12354     }
```

12355 The client may not set any of the attributes.

12356 **6.44.3 CMS mechanism parameters**

- **CK_CMS_SIG_PARAMS, CK_CMS_SIG_PARAMS_PTR**

12357 **CK_CMS_SIG_PARAMS** is a structure that provides the parameters to the **CKM_CMS_SIG** mechanism.
 12359 It is defined as follows:

```
12360     typedef struct CK_CMS_SIG_PARAMS {
12361         CK_OBJECT_HANDLE      certificateHandle;
12362         CK_MECHANISM_PTR      pSigningMechanism;
12363         CK_MECHANISM_PTR      pDigestMechanism;
```

```

12364     CK_UTF8CHAR_PTR           pContentType;
12365     CK_BYTE_PTR              pRequestedAttributes;
12366     CK ULONG                 ulRequestedAttributesLen;
12367     CK_BYTE_PTR              pRequiredAttributes;
12368     CK ULONG                 ulRequiredAttributesLen;
12369 } CK_CMS_SIG_PARAMS;
12370
12371 The fields of the structure have the following meanings:
12372     certificateHandle          Object handle for a certificate associated with the signing key. The token may use information from this certificate to identify the signer in the SignerInfo result value. CertificateHandle may be NULL_PTR if the certificate is not available as a PKCS #11 object or if the calling application leaves the choice of certificate completely to the token.
12373
12374     pSigningMechanism          Mechanism to use when signing a constructed CMS SignedAttributes value. E.g. CKM_SHA1_RSA_PKCS.
12375
12376     pDigestMechanism          Mechanism to use when digesting the data. Value shall be NULL_PTR when the digest mechanism to use follows from the pSigningMechanism parameter.
12377
12378     pContentType               NULL-terminated string indicating complete MIME Content-type of message to be signed; or the value NULL_PTR if the message is a MIME object (which the token can parse to determine its MIME Content-type if required). Use the value "application/octet-stream" if the MIME type for the message is unknown or undefined. Note that the pContentType string shall conform to the syntax specified in RFC 2045, i.e. any parameters needed for correct presentation of the content by the token (such as, for example, a non-default "charset") must be present. The token must follow rules and procedures defined in RFC 2045 when presenting the content.
12379
12380     pRequestedAttributes       Pointer to DER-encoded list of CMS Attributes the caller requests to be included in the signed attributes. Token may freely ignore this list or modify any supplied values.
12381
12382     ulRequestedAttributesLen   Length in bytes of the value pointed to by pRequestedAttributes
12383
12384     pRequiredAttributes        Pointer to DER-encoded list of CMS Attributes (with accompanying values) required to be included in the resulting signed attributes. Token must not modify any supplied values. If the token does not support one or more of the attributes, or does not accept provided values, the signature operation will fail. The token will use its own default attributes when signing if both the pRequestedAttributes and pRequiredAttributes field are set to NULL_PTR.
12385
12386     ulRequiredAttributesLen   Length in bytes, of the value pointed to by pRequiredAttributes.
12387
12388
12389
12390
12391
12392
12393
12394
12395
12396
12397
12398
12399
12400
12401
12402
12403
12404
12405
12406
12407
12408
12409
12410
12411

```

6.44.4 CMS signatures

The CMS mechanism, denoted **CKM_CMS_SIG**, is a multi-purpose mechanism based on the structures defined in [PKCS #7] and RFC 2630. It supports single- or multiple-part signatures with and without message recovery. The mechanism is intended for use with, e.g., PTDs (see MeT-PTD) or other capable tokens. The token will construct a CMS **SignedAttributes** value and compute a signature on this value. The content of the **SignedAttributes** value is decided by the token, however the caller can suggest some attributes in the parameter *pRequestedAttributes*. The caller can also require some attributes to be

12412 present through the parameters *pRequiredAttributes*. The signature is computed in accordance with the
12413 parameter *pSigningMechanism*.

12414 When this mechanism is used in successful calls to **C_Sign** or **C_SignFinal**, the *pSignature* return value
12415 will point to a DER-encoded value of type **SignerInfo**. **SignerInfo** is defined in ASN.1 as follows (for a
12416 complete definition of all fields and types, see RFC 2630):

```
12417     SignerInfo ::= SEQUENCE {
12418         version CMSVersion,
12419         sid SignerIdentifier,
12420         digestAlgorithm DigestAlgorithmIdentifier,
12421         signedAttrs [0] IMPLICIT SignedAttributes OPTIONAL,
12422         signatureAlgorithm SignatureAlgorithmIdentifier,
12423         signature SignatureValue,
12424         unsignedAttrs [1] IMPLICIT UnsignedAttributes
12425         OPTIONAL }
```

12426 The *certificateHandle* parameter, when set, helps the token populate the **sid** field of the **SignerInfo** value.
12427 If *certificateHandle* is NULL_PTR the choice of a suitable certificate reference in the **SignerInfo** result
12428 value is left to the token (the token could, e.g., interact with the user).

12429 This mechanism shall not be used in calls to **C_Verify** or **C_VerifyFinal** (use the *pSigningMechanism*
12430 mechanism instead).

12431 For the *pRequiredAttributes* field, the token may have to interact with the user to find out whether to
12432 accept a proposed value or not. The token should never accept any proposed attribute values without
12433 some kind of confirmation from its owner (but this could be through, e.g., configuration or policy settings
12434 and not direct interaction). If a user rejects proposed values, or the signature request as such, the value
12435 CKR_FUNCTION_REJECTED shall be returned.

12436 When possible, applications should use the **CKM_CMS_SIG** mechanism when generating CMS-
12437 compatible signatures rather than lower-level mechanisms such as **CKM_SHA1_RSA_PKCS**. This is
12438 especially true when the signatures are to be made on content that the token is able to present to a user.
12439 Exceptions may include those cases where the token does not support a particular signing attribute. Note
12440 however that the token may refuse usage of a particular signature key unless the content to be signed is
12441 known (i.e. the **CKM_CMS_SIG** mechanism is used).

12442 When a token does not have presentation capabilities, the PKCS #11-aware application may avoid
12443 sending the whole message to the token by electing to use a suitable signature mechanism (e.g.
12444 **CKM_RSA_PKCS**) as the *pSigningMechanism* value in the **CK_CMS_SIG_PARAMS** structure, and
12445 digesting the message itself before passing it to the token.

12446 PKCS #11-aware applications making use of tokens with presentation capabilities, should attempt to
12447 provide messages to be signed by the token in a format possible for the token to present to the user.
12448 Tokens that receive multipart MIME-messages for which only certain parts are possible to present may
12449 fail the signature operation with a return value of **CKR_DATA_INVALID**, but may also choose to add a
12450 signing attribute indicating which parts of the message were possible to present.

12451 **6.45 Blowfish**

12452 Blowfish, a secret-key block cipher. It is a Feistel network, iterating a simple encryption function 16 times.
12453 The block size is 64 bits, and the key can be any length up to 448 bits. Although there is a complex
12454 initialization phase required before any encryption can take place, the actual encryption of data is very
12455 efficient on large microprocessors. See [BLOWFISH] for details.

12456

12457 *Table 201, Blowfish Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|----------------------|------------------------------|--------------------------|--------------------------------|---------------|---------------------------|--------------------------|---------------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR¹ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_BLOWFISH_CBC | ✓ | | | | | ✓ | |
| CKM_BLOWFISH_CBC_PAD | ✓ | | | | | ✓ | |

12458 6.45.1 Definitions

12459 This section defines the key type "CKK_BLOWFISH" for type CK_KEY_TYPE as used in the
 12460 CKA_KEY_TYPE attribute of key objects.

12461 Mechanisms:

12462 CKM_BLOWFISH_KEY_GEN
 12463 CKM_BLOWFISH_CBC
 12464 CKM_BLOWFISH_CBC_PAD

12465 6.45.2 BLOWFISH secret key objects

12466 Blowfish secret key objects (object class CKO_SECRET_KEY, key type CKK_BLOWFISH) hold Blowfish
 12467 keys. The following table defines the Blowfish secret key object attributes, in addition to the common
 12468 attributes defined for this object class:

12469 *Table 202, BLOWFISH Secret Key Object*

| Attribute | Data type | Meaning |
|------------------------------|------------------|--|
| CKA_VALUE ^{1,4,6,7} | Byte array | Key value the key can be any length up to 448 bits. Bit length restricted to a byte array. |
| CKA_VALUE_LEN ^{2,3} | CK ULONG | Length in bytes of key value |

12470 ¹Refer to Table 11 for footnotes

12471 The following is a sample template for creating an Blowfish secret key object:

```
12472     CK_OBJECT_CLASS class = CKO_SECRET_KEY;
12473     CK_KEY_TYPE keyType = CKK_BLOWFISH;
12474     CK_UTF8CHAR label[] = "A blowfish secret key object";
12475     CK_BYTE value[16] = {...};
12476     CK_BBOOL true = CK_TRUE;
12477     CK_ATTRIBUTE template[] = {
12478         {CKA_CLASS, &class, sizeof(class)},
12479         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
12480         {CKA_TOKEN, &true, sizeof(true)},
12481         {CKA_LABEL, label, sizeof(label)-1},
12482         {CKA_ENCRYPT, &true, sizeof(true)},
12483         {CKA_VALUE, value, sizeof(value)}}
12484     };
```

12485 6.45.3 Blowfish key generation

12486 The Blowfish key generation mechanism, denoted **CKM_BLOWFISH_KEY_GEN**, is a key generation
12487 mechanism Blowfish.

12488 It does not have a parameter.

12489 The mechanism generates Blowfish keys with a particular length, as specified in the **CKA_VALUE_LEN**
12490 attribute of the template for the key.

12491 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
12492 key. Other attributes supported by the key type (specifically, the flags indicating which functions the key
12493 supports) may be specified in the template for the key, or else are assigned default initial values.

12494 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
12495 specify the supported range of key sizes in bytes.

12496 6.45.4 Blowfish-CBC

12497 Blowfish-CBC, denoted **CKM_BLOWFISH_CBC**, is a mechanism for single- and multiple-part encryption
12498 and decryption; key wrapping; and key unwrapping.

12499 It has a parameter, a 8-byte initialization vector.

12500 This mechanism can wrap and unwrap any secret key. For wrapping, the mechanism encrypts the value
12501 of the **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size
12502 minus one null bytes so that the resulting length is a multiple of the block size. The output data is the
12503 same length as the padded input data. It does not wrap the key type, key length, or any other information
12504 about the key; the application must convey these separately.

12505 For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the
12506 **CKA_KEY_TYPE** attribute of the template and, if it has one, and the key type supports it, the
12507 **CKA_VALUE_LEN** attribute of the template. The mechanism contributes the result as the **CKA_VALUE**
12508 attribute of the new key; other attributes required by the key type must be specified in the template.

12509 Constraints on key types and the length of data are summarized in the following table:

12510 *Table 203, BLOWFISH-CBC: Key and Data Length*

| Function | Key type | Input Length | Output Length |
|-------------|----------|------------------------|---|
| C_Encrypt | BLOWFISH | Multiple of block size | Same as input length |
| C_Decrypt | BLOWFISH | Multiple of block size | Same as input length |
| C_WrapKey | BLOWFISH | Any | Input length rounded up to multiple of the block size |
| C_UnwrapKey | BLOWFISH | Multiple of block size | Determined by type of key being unwrapped or CKA_VALUE_LEN |

12511 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
12512 specify the supported range of BLOWFISH key sizes, in bytes.

12513 6.45.5 Blowfish-CBC with PKCS padding

12514 Blowfish-CBC-PAD, denoted **CKM_BLOWFISH_CBC_PAD**, is a mechanism for single- and multiple-part
12515 encryption and decryption, key wrapping and key unwrapping, cipher-block chaining mode and the block
12516 cipher padding method detailed in [PKCS #7].

12517 It has a parameter, a 8-byte initialization vector.

12518 The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the
12519 ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified for
12520 the **CKA_VALUE_LEN** attribute.

12521 The entries in the table below for data length constraints when wrapping and unwrapping keys do not
12522 apply to wrapping and unwrapping private keys.

12523 Constraints on key types and the length of data are summarized in the following table:

12524

12525 *Table 204, BLOWFISH-CBC with PKCS Padding: Key and Data Length*

| Function | Key type | Input Length | Output Length |
|-------------|----------|------------------------|---|
| C_Encrypt | BLOWFISH | Any | Input length rounded up to multiple of the block size |
| C_Decrypt | BLOWFISH | Multiple of block size | Between 1 and block length block size bytes shorter than input length |
| C_WrapKey | BLOWFISH | Any | Input length rounded up to multiple of the block size |
| C_UnwrapKey | BLOWFISH | Multiple of block size | Between 1 and block length block size bytes shorter than input length |

12526 **6.46 Twofish**

12527 Twofish is a secret key block cipher. See [TWOFLASH] for details.

12528 **6.46.1 Definitions**

12529 This section defines the key type "CKK_TWOFISH" for type CK_KEY_TYPE as used in the
12530 CKA_KEY_TYPE attribute of key objects.

12531 Mechanisms:

12532 CKM_TWOFISH_KEY_GEN

12533 CKM_TWOFISH_CBC

12534 CKM_TWOFISH_CBC_PAD

12535

12536 **6.46.2 Twofish secret key objects**

12537 Twofish secret key objects (object class CKO_SECRET_KEY, key type CKK_TWOFISH) hold Twofish
12538 keys. The following table defines the Twofish secret key object attributes, in addition to the common
12539 attributes defined for this object class:

12540 *Table 205, Twofish Secret Key Object*

| Attribute | Data type | Meaning |
|------------------------------|------------|--------------------------------------|
| CKA_VALUE ^{1,4,6,7} | Byte array | Key value 128-, 192-, or 256-bit key |
| CKA_VALUE_LEN ^{2,3} | CK ULONG | Length in bytes of key value |

12541 ¹Refer to Table 11 for footnotes

12542 The following is a sample template for creating an TWOFLASH secret key object:

```
12543 CK_OBJECT_CLASS class = CKO_SECRET_KEY;
12544 CK_KEY_TYPE keyType = CKK_TWOFISH;
12545 CK_UTF8CHAR label[] = "A twofish secret key object";
12546 CK_BYTE value[16] = {...};
12547 CK_BBOOL true = CK_TRUE;
12548 CK_ATTRIBUTE template[] = {
12549     {CKA_CLASS, &class, sizeof(class)},
```

```

12550     { CKA_KEY_TYPE, &keyType, sizeof(keyType) },
12551     { CKA_TOKEN, &true, sizeof(true) },
12552     { CKA_LABEL, label, sizeof(label)-1 },
12553     { CKA_ENCRYPT, &true, sizeof(true) },
12554     { CKA_VALUE, value, sizeof(value) }
12555 } ;

```

6.46.3 Twofish key generation

12556 The Twofish key generation mechanism, denoted **CKM_TWOFISH_KEY_GEN**, is a key generation mechanism Twofish.
 12557 It does not have a parameter.
 12558 The mechanism generates Blowfish keys with a particular length, as specified in the **CKA_VALUE_LEN** attribute of the template for the key.
 12559 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.
 12560 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 12561 specify the supported range of key sizes, in bytes.

6.46.4 Twofish -CBC

12562 Twofish-CBC, denoted **CKM_TWOFISH_CBC**, is a mechanism for single- and multiple-part encryption
 12563 and decryption; key wrapping; and key unwrapping.
 12564 It has a parameter, a 16-byte initialization vector.

6.46.5 Twofish-CBC with PKCS padding

12565 Twofish-CBC-PAD, denoted **CKM_TWOFISH_CBC_PAD**, is a mechanism for single- and multiple-part
 12566 encryption and decryption, key wrapping and key unwrapping, cipher-block chaining mode and the block
 12567 cipher padding method detailed in [PKCS #7].
 12568 It has a parameter, a 16-byte initialization vector.
 12569 The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the
 12570 ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified for
 12571 the **CKA_VALUE_LEN** attribute.

6.47 CAMELLIA

12572 Camellia is a block cipher with 128-bit block size and 128-, 192-, and 256-bit keys, similar to AES.
 12573 Camellia is described e.g. in IETF RFC 3713.
 12574 *Table 206, Camellia Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|----------------------|-------------------|---------------|----------------------|--------|----------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen · Key / Key Pair | Wrap & Unwrap | Derive |
| CKM_CAMELLIA_KEY_GEN | | | | | ✓ | | |
| CKM_CAMELLIA_ECB | ✓ | | | | | ✓ | |

| Mechanism | Functions | | | | | | |
|-------------------------------|-------------------|---------------|----------------------|--------|----------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen . Key / Key Pair | Wrap & Unwrap | Derive |
| CKM_CAMELLIA_CBC | ✓ | | | | | ✓ | |
| CKM_CAMELLIA_CBC_PAD | ✓ | | | | | ✓ | |
| CKM_CAMELLIA_MAC_GENERAL | | ✓ | | | | | |
| CKM_CAMELLIA_MAC | | ✓ | | | | | |
| CKM_CAMELLIA_ECB_ENCRYPT_DATA | | | | | | | ✓ |
| CKM_CAMELLIA_CBC_ENCRYPT_DATA | | | | | | | ✓ |

6.47.1 Definitions

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

Mechanisms:

- CKM_CAMELLIA_KEY_GEN
- CKM_CAMELLIA_ECB
- CKM_CAMELLIA_CBC
- CKM_CAMELLIA_MAC
- CKM_CAMELLIA_MAC_GENERAL
- CKM_CAMELLIA_CBC_PAD

6.47.2 Camellia secret key objects

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

Table 207, Camellia Secret Key Object Attributes

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

Refer to Table 11 for footnotes

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

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_CAMELLIA;
CK_UTF8CHAR label[] = "A Camellia secret key object";
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
```

```
12606     { CKA_CLASS, &class, sizeof(class) },
12607     { CKA_KEY_TYPE, &keyType, sizeof(keyType) },
12608     { CKA_TOKEN, &true, sizeof(true) },
12609     { CKA_LABEL, label, sizeof(label)-1 },
12610     { CKA_ENCRYPT, &true, sizeof(true) },
12611     { CKA_VALUE, value, sizeof(value) }
12612 };
```

6.47.3 Camellia key generation

The Camellia key generation mechanism, denoted CKM_CAMELLIA_KEY_GEN, is a key generation mechanism for Camellia.

It does not have a parameter.

The mechanism generates Camellia 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 Camellia 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 Camellia key sizes, in bytes.

6.47.4 Camellia-ECB

Camellia-ECB, denoted **CKM_CAMELLIA_ECB**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on Camellia 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 208, Camellia-ECB: Key and Data Length

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

12640 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 12641 specify the supported range of Camellia key sizes, in bytes.

6.47.5 Camellia-CBC

12643 Camellia-CBC, denoted **CKM_CAMELLIA_CBC**, is a mechanism for single- and multiple-part encryption
 12644 and decryption; key wrapping; and key unwrapping, based on Camellia and cipher-block chaining mode.

12645 It has a parameter, a 16-byte initialization vector.

12646 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to
 12647 wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the
 12648 **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus
 12649 one null bytes so that the resulting length is a multiple of the block size. The output data is the same
 12650 length as the padded input data. It does not wrap the key type, key length, or any other information about
 12651 the key; the application must convey these separately.

12652 For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the
 12653 **CKA_KEY_TYPE** attribute of the template and, if it has one, and the key type supports it, the
 12654 **CKA_VALUE_LEN** attribute of the template. The mechanism contributes the result as the **CKA_VALUE**
 12655 attribute of the new key; other attributes required by the key type must be specified in the template.

12656 Constraints on key types and the length of data are summarized in the following table:

12657 *Table 209, Camellia-CBC: Key and Data Length*

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

12658 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 12659 specify the supported range of Camellia key sizes, in bytes.

6.47.6 Camellia-CBC with PKCS padding

12661 Camellia-CBC with PKCS padding, denoted **CKM_CAMELLIA_CBC_PAD**, is a mechanism for single-
 12662 and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on Camellia;
 12663 cipher-block chaining mode; and the block cipher padding method detailed in [PKCS #7].

- 12664 It has a parameter, a 16-byte initialization vector.
- 12665 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.
- 12668 In addition to being able to wrap and unwrap secret keys, this mechanism can wrap and unwrap RSA, Diffie-Hellman, X9.42 Diffie-Hellman, short Weierstrass EC and DSA private keys (see Section 6.7 for details). The entries in the table below for data length constraints when wrapping and unwrapping keys do not apply to wrapping and unwrapping private keys.
- 12672 Constraints on key types and the length of data are summarized in the following table:

12673 *Table 210, Camellia-CBC with PKCS Padding: Key and Data Length*

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

- 12674 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of Camellia key sizes, in bytes.

12676

12677 6.47.7 CAMELLIA with Counter mechanism parameters

12678 ♦ **CK_CAMELLIA_CTR_PARAMS; CK_CAMELLIA_CTR_PARAMS_PTR**

12679 **CK_CAMELLIA_CTR_PARAMS** is a structure that provides the parameters to the
12680 **CKM_CAMELLIA_CTR** mechanism. It is defined as follows:

```
12681     typedef struct CK_CAMELLIA_CTR_PARAMS {
12682         CK ULONG ulCounterBits;
12683         CK_BYTE cb[16];
12684     } CK_CAMELLIA_CTR_PARAMS;
```

12686 *ulCounterBits* specifies the number of bits in the counter block (*cb*) that shall be incremented. This
12687 number shall be such that $0 < \text{ulCounterBits} \leq 128$. For any values outside this range the mechanism
12688 shall return **CKR_MECHANISM_PARAM_INVALID**.

12689 It's up to the caller to initialize all of the bits in the counter block including the counter bits. The counter
12690 bits are the least significant bits of the counter block (*cb*). They are a big-endian value usually starting
12691 with 1. The rest of '*cb*' is for the nonce, and maybe an optional IV.

12692 E.g. as defined in [RFC 3686]:

```

12693      0           1           2           3
12694      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
12695      +-----+-----+-----+-----+-----+-----+
12696      |           Nonce          |
12697      +-----+-----+-----+-----+-----+-----+-----+
12698      |           Initialization Vector (IV)   |
12699      |
12700      +-----+-----+-----+-----+-----+-----+-----+
12701      |           Block Counter        |
12702      +-----+-----+-----+-----+-----+-----+-----+
12703

```

12704 This construction permits each packet to consist of up to $2^{32}-1$ blocks = 4,294,967,295 blocks =
 12705 68,719,476,720 octets.

12706 **CK_CAMELLIA_CTR_PARAMS_PTR** is a pointer to a **CK_CAMELLIA_CTR_PARAMS**.

12707

12708 6.47.8 General-length Camellia-MAC

12709 General-length Camellia -MAC, denoted CKM_CAMELLIA_MAC_GENERAL, is a mechanism for single-
 12710 and multiple-part signatures and verification, based on Camellia and data authentication as defined
 12711 in.[CAMELLIA]

12712 It has a parameter, a **CK_MAC_GENERAL_PARAMS** structure, which specifies the output length
 12713 desired from the mechanism.

12714 The output bytes from this mechanism are taken from the start of the final Camellia cipher block produced
 12715 in the MACing process.

12716 Constraints on key types and the length of data are summarized in the following table:

12717 *Table 211, General-length Camellia-MAC: Key and Data Length*

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

12718 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 12719 specify the supported range of Camellia key sizes, in bytes.

12720 6.47.9 Camellia-MAC

12721 Camellia-MAC, denoted by **CKM_CAMELLIA_MAC**, is a special case of the general-length Camellia-
 12722 MAC mechanism. Camellia-MAC always produces and verifies MACs that are half the block size in
 12723 length.

12724 It does not have a parameter.

12725 Constraints on key types and the length of data are summarized in the following table:

12726 *Table 212, Camellia-MAC: Key and Data Length*

| Function | Key type | Data length | Signature length |
|----------|--------------|-------------|------------------------------------|
| C_Sign | CKK_CAMELLIA | any | $\frac{1}{2}$ block size (8 bytes) |
| C_Verify | CKK_CAMELLIA | any | $\frac{1}{2}$ block size (8 bytes) |

12727 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 12728 specify the supported range of Camellia key sizes, in bytes.

12729 6.48 Key derivation by data encryption - Camellia

12730 These mechanisms allow derivation of keys using the result of an encryption operation as the key value.
12731 They are for use with the C_DeriveKey function.

12732 6.48.1 Definitions

12733 Mechanisms:

12734 CKM_CAMELLIA_ECB_ENCRYPT_DATA
12735 CKM_CAMELLIA_CBC_ENCRYPT_DATA

12736

```
12737     typedef struct CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS {  
12738         CK_BYTE          iv[16];  
12739         CK_BYTE_PTR      pData;  
12740         CK ULONG         length;  
12741     } CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS;  
12742  
12743     typedef CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS CK_PTR  
12744             CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS_PTR;
```

12745 6.48.2 Mechanism Parameters

12746 Uses CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS, and CK_KEY_DERIVATION_STRING_DATA.

12747 *Table 213, Mechanism Parameters for Camellia-based key derivation*

| | |
|-------------------------------|---|
| CKM_CAMELLIA_ECB_ENCRYPT_DATA | Uses CK_KEY_DERIVATION_STRING_DATA structure. Parameter is the data to be encrypted and must be a multiple of 16 long. |
| CKM_CAMELLIA_CBC_ENCRYPT_DATA | Uses CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS. Parameter is an 16 byte IV value followed by the data. The data value part must be a multiple of 16 bytes long. |

12748

12749 6.49 ARIA

12750 ARIA is a block cipher with 128-bit block size and 128-, 192-, and 256-bit keys, similar to AES. ARIA is
12751 described in NSRI "Specification of ARIA".

12752 *Table 214, ARIA Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|------------------|-------------------|---------------|----------------------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_ARIA_KEY_GEN | | | | | ✓ | | |
| CKM_ARIA_ECB | ✓ | | | | | ✓ | |
| CKM_ARIA_CBC | ✓ | | | | | ✓ | |
| CKM_ARIA_CBC_PAD | ✓ | | | | | ✓ | |

| Mechanism | Functions | | | | | | |
|---------------------------|-------------------|---------------|----------------------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_ARIA_MAC_GENERAL | | ✓ | | | | | |
| CKM_ARIA_MAC | | ✓ | | | | | |
| CKM_ARIA_ECB_ENCRYPT_DATA | | | | | | | ✓ |
| CKM_ARIA_CBC_ENCRYPT_DATA | | | | | | | ✓ |

6.49.1 Definitions

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

Mechanisms:

- CKM_ARIA_KEY_GEN
- CKM_ARIA_ECB
- CKM_ARIA_CBC
- CKM_ARIA_MAC
- CKM_ARIA_MAC_GENERAL
- CKM_ARIA_CBC_PAD

6.49.2 Aria secret key objects

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

Table 215, ARIA Secret Key Object Attributes

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

Refer to Table 11 for footnotes

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

```

CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_ARIA;
CK_UTF8CHAR label[] = "An ARIA secret key object";
CK_BYTE value[] = { ... };
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_ENCRYPT, &true, sizeof(true)},
```

```
12781     { CKA_VALUE, value, sizeof(value) }  
12782 } ;
```

6.49.3 ARIA key generation

12784 The ARIA key generation mechanism, denoted CKM_ARIA_KEY_GEN, is a key generation mechanism
12785 for Aria.

12786 It does not have a parameter.

12787 The mechanism generates ARIA keys with a particular length in bytes, as specified in the
12788 **CKA_VALUE_LEN** attribute of the template for the key.

12789 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
12790 key. Other attributes supported by the ARIA key type (specifically, the flags indicating which functions the
12791 key supports) may be specified in the template for the key, or else are assigned default initial values.

12792 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
12793 specify the supported range of ARIA key sizes, in bytes.

6.49.4 ARIA-ECB

12795 ARIA-ECB, denoted **CKM_ARIA_ECB**, is a mechanism for single- and multiple-part encryption and
12796 decryption; key wrapping; and key unwrapping, based on Aria and electronic codebook mode.

12797 It does not have a parameter.

12798 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to
12799 wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the
12800 **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus
12801 one null bytes so that the resulting length is a multiple of the block size. The output data is the same
12802 length as the padded input data. It does not wrap the key type, key length, or any other information about
12803 the key; the application must convey these separately.

12804 For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the
12805 **CKA_KEY_TYPE** attribute of the template and, if it has one, and the key type supports it, the
12806 **CKA_VALUE_LEN** attribute of the template. The mechanism contributes the result as the **CKA_VALUE**
12807 attribute of the new key; other attributes required by the key type must be specified in the template.

12808 Constraints on key types and the length of data are summarized in the following table:

12809 *Table 216, ARIA-ECB: Key and Data Length*

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

12810 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
12811 specify the supported range of ARIA key sizes, in bytes.

6.49.5 ARIA-CBC

12813 ARIA-CBC, denoted **CKM_ARIA_CBC**, is a mechanism for single- and multiple-part encryption and
12814 decryption; key wrapping; and key unwrapping, based on ARIA and cipher-block chaining mode.

12815 It has a parameter, a 16-byte initialization vector.

12816 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to
12817 wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the
12818 **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus
12819 one null bytes so that the resulting length is a multiple of the block size. The output data is the same
12820 length as the padded input data. It does not wrap the key type, key length, or any other information about
12821 the key; the application must convey these separately.

12822 For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the
12823 **CKA_KEY_TYPE** attribute of the template and, if it has one, and the key type supports it, the
12824 **CKA_VALUE_LEN** attribute of the template. The mechanism contributes the result as the **CKA_VALUE**
12825 attribute of the new key; other attributes required by the key type must be specified in the template.

12826 Constraints on key types and the length of data are summarized in the following table:

12827 *Table 217, ARIA-CBC: Key and Data Length*

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

12828 For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure
12829 specify the supported range of Aria key sizes, in bytes.

12830 **6.49.6 ARIA-CBC with PKCS padding**

12831 ARIA-CBC with PKCS padding, denoted **CKM_ARIA_CBC_PAD**, is a mechanism for single- and
12832 multiple-part encryption and decryption; key wrapping; and key unwrapping, based on ARIA; cipher-block
12833 chaining mode; and the block cipher padding method detailed in [PKCS #7].

12834 It has a parameter, a 16-byte initialization vector.

12835 The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the
12836 ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified
12837 for the **CKA_VALUE_LEN** attribute.

12838 In addition to being able to wrap and unwrap secret keys, this mechanism can wrap and unwrap RSA,
12839 Diffie-Hellman, X9.42 Diffie-Hellman, short Weierstrass EC and DSA private keys (see Section 6.7 for
12840 details). The entries in the table below for data length constraints when wrapping and unwrapping keys
12841 do not apply to wrapping and unwrapping private keys.

12842 Constraints on key types and the length of data are summarized in the following table:

12843 *Table 218, ARIA-CBC with PKCS Padding: Key and Data Length*

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

12844 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 12845 specify the supported range of ARIA key sizes, in bytes.

6.49.7 General-length ARIA-MAC

12847 General-length ARIA -MAC, denoted **CKM_ARIA_MAC_GENERAL**, is a mechanism for single- and
 12848 multiple-part signatures and verification, based on ARIA and data authentication as defined in [FIPS 113].

12849 It has a parameter, a **CK_MAC_GENERAL_PARAMS** structure, which specifies the output length
 12850 desired from the mechanism.

12851 The output bytes from this mechanism are taken from the start of the final ARIA cipher block produced in
 12852 the MACing process.

12853 Constraints on key types and the length of data are summarized in the following table:

12854 *Table 219, General-length ARIA-MAC: Key and Data Length*

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

12855 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 12856 specify the supported range of ARIA key sizes, in bytes.

6.49.8 ARIA-MAC

12858 ARIA-MAC, denoted by **CKM_ARIA_MAC**, is a special case of the general-length ARIA-MAC
 12859 mechanism. ARIA-MAC always produces and verifies MACs that are half the block size in length.

12860 It does not have a parameter.

12861 Constraints on key types and the length of data are summarized in the following table:

12862 *Table 220, ARIA-MAC: Key and Data Length*

| Function | Key type | Data length | Signature length |
|----------|----------|-------------|------------------------------------|
| C_Sign | CKK_ARIA | any | $\frac{1}{2}$ block size (8 bytes) |
| C_Verify | CKK_ARIA | any | $\frac{1}{2}$ block size (8 bytes) |

12863 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 12864 specify the supported range of ARIA key sizes, in bytes.

6.50 Key derivation by data encryption - ARIA

12866 These mechanisms allow derivation of keys using the result of an encryption operation as the key value.
 12867 They are for use with the C_DeriveKey function.

12868 **6.50.1 Definitions**

12869 Mechanisms:

12870 CKM_ARIA_ECB_ENCRYPT_DATA
12871 CKM_ARIA_CBC_ENCRYPT_DATA

12872

```
12873     typedef struct CK_ARIA_CBC_ENCRYPT_DATA_PARAMS {  
12874         CK_BYTE           iv[16];  
12875         CK_BYTE_PTR      pData;  
12876         CK_ULONG          length;  
12877     } CK_ARIA_CBC_ENCRYPT_DATA_PARAMS;  
12878  
12879     typedef CK_ARIA_CBC_ENCRYPT_DATA_PARAMS CK_PTR  
12880             CK_ARIA_CBC_ENCRYPT_DATA_PARAMS_PTR;
```

12881 **6.50.2 Mechanism Parameters**

12882 Uses CK_ARIA_CBC_ENCRYPT_DATA_PARAMS, and CK_KEY_DERIVATION_STRING_DATA.

12883 *Table 221, Mechanism Parameters for Aria-based key derivation*

| | |
|---------------------------|---|
| CKM_ARIA_ECB_ENCRYPT_DATA | Uses CK_KEY_DERIVATION_STRING_DATA structure. Parameter is the data to be encrypted and must be a multiple of 16 long. |
| CKM_ARIA_CBC_ENCRYPT_DATA | Uses CK_ARIA_CBC_ENCRYPT_DATA_PARAMS. Parameter is an 16 byte IV value followed by the data. The data value part must be a multiple of 16 bytes long. |

12884

12885 **6.51 SEED**

12886 SEED is a symmetric block cipher developed by the South Korean Information Security Agency (KISA). It
12887 has a 128-bit key size and a 128-bit block size.

12888 Its specification has been published as Internet [RFC 4269].

12889 RFCs have been published defining the use of SEED in

12890 TLS <ftp://ftp.rfc-editor.org/in-notes/rfc4162.txt>

12891 IPsec <ftp://ftp.rfc-editor.org/in-notes/rfc4196.txt>

12892 CMS <ftp://ftp.rfc-editor.org/in-notes/rfc4010.txt>

12893

12894 TLS cipher suites that use SEED include:

```
12895     CipherSuite TLS_RSA_WITH_SEED_CBC_SHA      = { 0x00,  
12896         0x96};  
12897     CipherSuite TLS_DH_DSS_WITH_SEED_CBC_SHA   = { 0x00,  
12898         0x97};  
12899     CipherSuite TLS_DH_RSA_WITH_SEED_CBC_SHA   = { 0x00,  
12900         0x98};  
12901     CipherSuite TLS_DHE_DSS_WITH_SEED_CBC_SHA = { 0x00,  
12902         0x99};
```

```

12903     CipherSuite TLS_DHE_RSA_WITH_SEED_CBC_SHA = { 0x00,
12904         0x9A};
12905     CipherSuite TLS_DH_anon_WITH_SEED_CBC_SHA = { 0x00,
12906         0x9B};

12907
12908 As with any block cipher, it can be used in the ECB, CBC, OFB and CFB modes of operation, as well as
12909 in a MAC algorithm such as HMAC.
12910 OIDs have been published for all these uses. A list may be seen at
12911 http://www.alvestrand.no/objectid/1.2.410.200004.1.html
12912
12913 Table 222, SEED Mechanisms vs. Functions

```

| Mechanism | Functions | | | | | | |
|---------------------------|-------------------|---------------|----------------------|--------|---------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen . Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_SEED_KEY_GEN | | | | | ✓ | | |
| CKM_SEED_ECB | | | ✓ | | | | |
| CKM_SEED_CBC | | | ✓ | | | | |
| CKM_SEED_CBC_PAD | ✓ | | | | | ✓ | |
| CKM_SEED_MAC_GENERAL | | | ✓ | | | | |
| CKM_SEED_MAC | | | | ✓ | | | |
| CKM_SEED_ECB_ENCRYPT_DATA | | | | | | | ✓ |
| CKM_SEED_CBC_ENCRYPT_DATA | | | | | | | ✓ |

6.51.1 Definitions

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

Mechanisms:

```

12918     CKM_SEED_KEY_GEN
12919     CKM_SEED_ECB
12920     CKM_SEED_CBC
12921     CKM_SEED_MAC
12922     CKM_SEED_MAC_GENERAL
12923     CKM_SEED_CBC_PAD
12924

```

For all of these mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** are always 16.

6.51.2 SEED secret key objects

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

12931 Table 223, SEED Secret Key Object Attributes

| Attribute | Data type | Meaning |
|------------------------------|------------|----------------------------------|
| CKA_VALUE ^{1,4,6,7} | Byte array | Key value (always 16 bytes long) |

12932 ^ Refer to Table 11 for footnotes

12933 The following is a sample template for creating a SEED secret key object:

```
12934 CK_OBJECT_CLASS class = CKO_SECRET_KEY;
12935 CK_KEY_TYPE keyType = CKK_SEED;
12936 CK_UTF8CHAR label[] = "A SEED secret key object";
12937 CK_BYTE value[] = {...};
12938 CK_BBOOL true = CK_TRUE;
12939 CK_ATTRIBUTE template[] = {
12940     {CKA_CLASS, &class, sizeof(class)},
12941     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
12942     {CKA_TOKEN, &true, sizeof(true)},
12943     {CKA_LABEL, label, sizeof(label)-1},
12944     {CKA_ENCRYPT, &true, sizeof(true)},
12945     {CKA_VALUE, value, sizeof(value)}}
12946 };
```

12947 6.51.3 SEED key generation

12948 The SEED key generation mechanism, denoted CKM_SEED_KEY_GEN, is a key generation mechanism
12949 for SEED.

12950 It does not have a parameter.

12951 The mechanism generates SEED keys.

12952 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
12953 key. Other attributes supported by the SEED key type (specifically, the flags indicating which functions
12954 the key supports) may be specified in the template for the key, or else are assigned default initial values.

12955 6.51.4 SEED-ECB

12956 SEED-ECB, denoted **CKM_SEED_ECB**, is a mechanism for single- and multiple-part encryption and
12957 decryption; key wrapping; and key unwrapping, based on SEED and electronic codebook mode.

12958 It does not have a parameter.

12959 6.51.5 SEED-CBC

12960 SEED-CBC, denoted **CKM_SEED_CBC**, is a mechanism for single- and multiple-part encryption and
12961 decryption; key wrapping; and key unwrapping, based on SEED and cipher-block chaining mode.

12962 It has a parameter, a 16-byte initialization vector.

12963 6.51.6 SEED-CBC with PKCS padding

12964 SEED-CBC with PKCS padding, denoted **CKM_SEED_CBC_PAD**, is a mechanism for single- and
12965 multiple-part encryption and decryption; key wrapping; and key unwrapping, based on SEED; cipher-
12966 block chaining mode; and the block cipher padding method detailed in [PKCS #7].

12967 It has a parameter, a 16-byte initialization vector.

12968 6.51.7 General-length SEED-MAC

12969 General-length SEED-MAC, denoted **CKM_SEED_MAC_GENERAL**, is a mechanism for single- and
12970 multiple-part signatures and verification, based on SEED and data authentication.

12971 It has a parameter, a **CK_MAC_GENERAL_PARAMS** structure, which specifies the output length
12972 desired from the mechanism.

12973 The output bytes from this mechanism are taken from the start of the final cipher block produced in the
12974 MACing process.

12975 6.51.8 SEED-MAC

12976 SEED-MAC, denoted by **CKM_SEED_MAC**, is a special case of the general-length SEED-MAC
12977 mechanism. SEED-MAC always produces and verifies MACs that are half the block size in length.

12978 It does not have a parameter.

12979 6.52 Key derivation by data encryption - SEED

12980 These mechanisms allow derivation of keys using the result of an encryption operation as the key value.
12981 They are for use with the C_DeriveKey function.

12982 6.52.1 Definitions

12983 Mechanisms:

```
12984     CKM_SEED_ECB_ENCRYPT_DATA  
12985     CKM_SEED_CBC_ENCRYPT_DATA
```

12986

```
12987     typedef struct CK_SEED_CBC_ENCRYPT_DATA_PARAMS {  
12988         CK_BYTE           iv[16];  
12989         CK_BYTE_PTR      pData;  
12990         CK_ULONG          length;  
12991     } CK_SEED_CBC_ENCRYPT_DATA_PARAMS;  
12992  
12993     typedef CK_SEED_CBC_ENCRYPT_DATA_PARAMS CK_PTR  
12994             CK_SEED_CBC_ENCRYPT_DATA_PARAMS_PTR;
```

12995 6.52.2 Mechanism Parameters

12996 *Table 224, Mechanism Parameters for SEED-based key derivation*

| | |
|---------------------------|---|
| CKM_SEED_ECB_ENCRYPT_DATA | Uses CK_KEY_DERIVATION_STRING_DATA structure. Parameter is the data to be encrypted and must be a multiple of 16 long. |
| CKM_SEED_CBC_ENCRYPT_DATA | Uses CK_SEED_CBC_ENCRYPT_DATA_PARAMS. Parameter is an 16 byte IV value followed by the data. The data value part must be a multiple of 16 bytes long. |

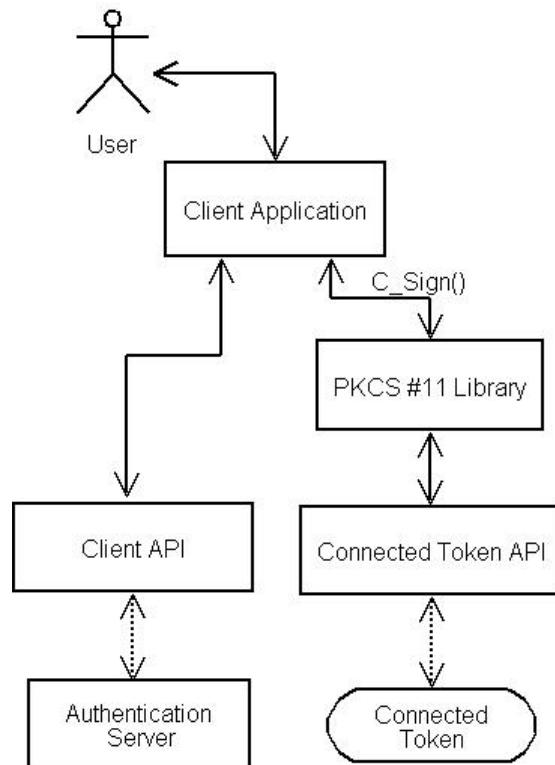
12997

12998 **6.53 OTP**

12999 **6.53.1 Usage overview**

13000 OTP tokens represented as PKCS #11 mechanisms may be used in a variety of ways. The usage cases
13001 can be categorized according to the type of sought functionality.

13002 **6.53.2 Case 1: Generation of OTP values**

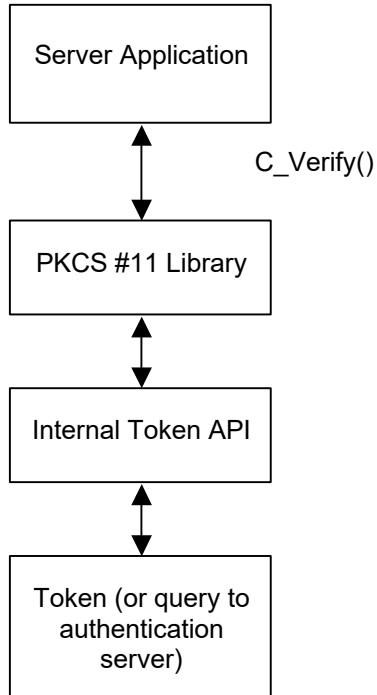


13003

13004 *Figure 2: Retrieving OTP values through C_Sign*

13005 Figure 2 shows an integration of PKCS #11 into an application that needs to authenticate users holding
13006 OTP tokens. In this particular example, a connected hardware token is used, but a software token is
13007 equally possible. The application invokes **C_Sign** to retrieve the OTP value from the token. In the
13008 example, the application then passes the retrieved OTP value to a client API that sends it via the network
13009 to an authentication server. The client API may implement a standard authentication protocol such as
13010 RADIUS [RFC 2865] or EAP [RFC 3748], or a proprietary protocol such as that used by RSA Security's
13011 ACE/Agent® software.

13012 **6.53.3 Case 2: Verification of provided OTP values**

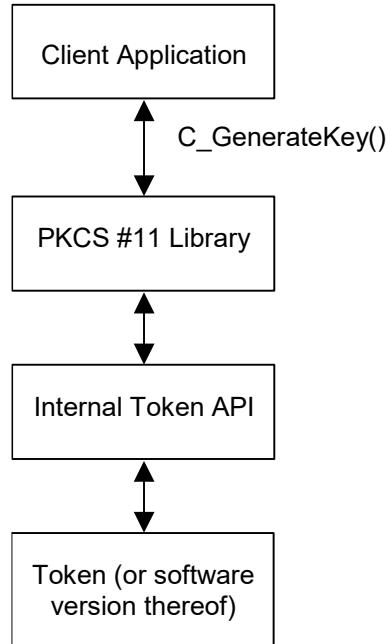


13013

13014 *Figure 3: Server-side verification of OTP values*

13015 Figure 3 illustrates the server-side equivalent of the scenario depicted in Figure 2. In this case, a server
13016 application invokes **C_Verify** with the received OTP value as the signature value to be verified.

13017 **6.53.4 Case 3: Generation of OTP keys**



13018

13019 *Figure 4: Generation of an OTP key*

13020 Figure 4 shows an integration of PKCS #11 into an application that generates OTP keys. The application
13021 invokes **C_GenerateKey** to generate an OTP key of a particular type on the token. The key may
13022 subsequently be used as a basis to generate OTP values.

13023 **6.53.5 OTP objects**

13024 **6.53.5.1 Key objects**

13025 OTP key objects (object class **CKO OTP KEY**) hold secret keys used by OTP tokens. The following
13026 table defines the attributes common to all OTP keys, in addition to the attributes defined for secret keys,
13027 all of which are inherited by this class:

13028 *Table 225: Common OTP key attributes*

| Attribute | Data type | Meaning |
|--|------------------|---|
| CKA OTP FORMAT | CK ULONG | Format of OTP values produced with this key: CK OTP FORMAT DECIMAL = Decimal (default) (UTF8-encoded) CK OTP FORMAT HEXADECIMAL = Hexadecimal (UTF8-encoded) CK OTP FORMAT ALPHANUMERIC = Alphanumeric (UTF8-encoded) CK OTP FORMAT BINARY = Only binary values. |
| CKA OTP LENGTH ⁹ | CK ULONG | Default length of OTP values (in the CKA OTP FORMAT) produced with this key. |
| CKA OTP USER FRIENDLY MODE ⁹ | CK BBOOL | Set to CK TRUE when the token is capable of returning OTPs suitable for human consumption. See the description of CKF USER FRIENDLY OTP below. |
| CKA OTP CHALLENGE REQUIREMENT ⁹ | CK ULONG | Parameter requirements when generating or verifying OTP values with this key: CK OTP PARAM MANDATORY = A challenge must be supplied. CK OTP PARAM OPTIONAL = A challenge may be supplied but need not be. CK OTP PARAM IGNORED = A challenge, if supplied, will be ignored. |
| CKA OTP TIME REQUIREMENT ⁹ | CK ULONG | Parameter requirements when generating or verifying OTP values with this key: CK OTP PARAM MANDATORY = A time value must be supplied. CK OTP PARAM OPTIONAL = A time value may be supplied but need not be. CK OTP PARAM IGNORED = A time value, if supplied, will be ignored. |

| | | |
|--|-----------------|---|
| CKA OTP COUNTER REQUIREMENT ⁹ | CK ULONG | Parameter requirements when generating or verifying OTP values with this key: CK OTP PARAM MANDATORY = A counter value must be supplied. CK OTP PARAM OPTIONAL = A counter value may be supplied but need not be. CK OTP PARAM IGNORED = A counter value, if supplied, will be ignored. |
| CKA OTP PIN REQUIREMENT ⁹ | CK ULONG | Parameter requirements when generating or verifying OTP values with this key: CK OTP PARAM MANDATORY = A PIN value must be supplied. CK OTP PARAM OPTIONAL = A PIN value may be supplied but need not be (if not supplied, then library will be responsible for collecting it) CK OTP PARAM IGNORED = A PIN value, if supplied, will be ignored. |
| CKA OTP COUNTER | Byte array | Value of the associated internal counter. Default value is empty (i.e. <i>ulValueLen</i> = 0). |
| CKA OTP TIME | RFC 2279 string | Value of the associated internal UTC time in the form YYYYMMDDhhmmss. Default value is empty (i.e. <i>ulValueLen</i> = 0). |
| CKA OTP USER IDENTIFIER | RFC 2279 string | Text string that identifies a user associated with the OTP key (may be used to enhance the user experience). Default value is empty (i.e. <i>ulValueLen</i> = 0). |
| CKA OTP SERVICE IDENTIFIER | RFC 2279 string | Text string that identifies a service that may validate OTPs generated by this key. Default value is empty (i.e. <i>ulValueLen</i> = 0). |
| CKA OTP SERVICE LOGO | Byte array | Logotype image that identifies a service that may validate OTPs generated by this key. Default value is empty (i.e. <i>ulValueLen</i> = 0). |
| CKA OTP SERVICE LOGO TYPE | RFC 2279 string | MIME type of the CKA OTP SERVICE LOGO attribute value. Default value is empty (i.e. <i>ulValueLen</i> = 0). |
| CKA VALUE ^{1, 4, 6, 7} | Byte array | Value of the key. |
| CKA VALUE LEN ^{2, 3} | CK ULONG | Length in bytes of key value. |

13029 Refer to Table 11 for footnotes

13030 Note: A Cryptoki library may support PIN-code caching in order to reduce user interactions. An OTP-
 13031 PKCS #11 application should therefore always consult the state of the CKA OTP_PIN_REQUIREMENT
 13032 attribute before each call to **C_SignInit**, as the value of this attribute may change dynamically.
 13033 For OTP tokens with multiple keys, the keys may be enumerated using **C_FindObjects**. The
 13034 **CKA OTP SERVICE IDENTIFIER** and/or the **CKA OTP SERVICE LOGO** attribute may be used to
 13035 distinguish between keys. The actual choice of key for a particular operation is however application-
 13036 specific and beyond the scope of this document.
 13037 For all OTP keys, the CKA_ALLOWED_MECHANISMS attribute should be set as required.

13038 **6.53.6 OTP-related notifications**

13039 This document extends the set of defined notifications as follows:

| | |
|------------------------------|---|
| 13040 CKN OTP CHANGED | Cryptoki is informing the application that the OTP for a key on a 13041 connected token just changed. This notification is particularly useful 13042 when applications wish to display the current OTP value for time- 13043 based mechanisms. |
|------------------------------|---|

13044 **6.53.7 OTP mechanisms**

13045 The following table shows, for the OTP mechanisms defined in this document, their support by different
 13046 cryptographic operations. For any particular token, of course, a particular operation may well support
 13047 only a subset of the mechanisms listed. There is also no guarantee that a token that supports one
 13048 mechanism for some operation supports any other mechanism for any other operation (or even supports
 13049 that same mechanism for any other operation).

13050 *Table 226: OTP mechanisms vs. applicable functions*

| Mechanism | Functions | | | | | | |
|---------------------|-------------------|---------------|----------------------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_SECURID_KEY_GEN | | | | | ✓ | | |
| CKM_SECURID | | ✓ | | | | | |
| CKM_HOTP_KEY_GEN | | | | | ✓ | | |
| CKM_HOTP | | ✓ | | | | | |
| CKM_ACTI_KEY_GEN | | | | | ✓ | | |
| CKM_ACTI | | ✓ | | | | | |

13051 The remainder of this section will present in detail the OTP mechanisms and the parameters that are
 13052 supplied to them.

13053 **6.53.7.1 OTP mechanism parameters**

13054 ◆ **CK OTP PARAM TYPE**

13055 **CK OTP PARAM TYPE** is a value that identifies an OTP parameter type. It is defined as follows:

```
13056     typedef CK ULONG CK OTP PARAM TYPE;
```

13057 The following **CK OTP PARAM TYPE** types are defined:

13058 *Table 227, OTP parameter types*

| Parameter | Data type | Meaning |
|----------------------|-----------------|---|
| CK OTP PIN | RFC 2279 string | A UTF8 string containing a PIN for use when computing or verifying PIN-based OTP values. |
| CK OTP CHALLENGE | Byte array | Challenge to use when computing or verifying challenge-based OTP values. |
| CK OTP TIME | RFC 2279 string | UTC time value in the form YYYYMMDDhhmmss to use when computing or verifying time-based OTP values. |
| CK OTP COUNTER | Byte array | Counter value to use when computing or verifying counter-based OTP values. |
| CK OTP FLAGS | CK_FLAGS | Bit flags indicating the characteristics of the sought OTP as defined below. |
| CK OTP OUTPUT LENGTH | CK ULONG | Desired output length (overrides any default value). A Cryptoki library will return CKR_MECHANISM_PARAM_INVALID if a provided length value is not supported. |
| CK OTP OUTPUT FORMAT | CK ULONG | Returned OTP format (allowed values are the same as for CKA OTP FORMAT). This parameter is only intended for C_Sign output, see paragraphs below. When not present, the returned OTP format will be the same as the value of the CKA OTP FORMAT attribute for the key in question. |
| CK OTP VALUE | Byte array | An actual OTP value. This parameter type is intended for C_Sign output, see paragraphs below. |

13059

13060 The following table defines the possible values for the CK OTP FLAGS type:

13061 *Table 228: OTP Mechanism Flags*

| Bit flag | Mask | Meaning |
|--------------|------------|--|
| CKF_NEXT OTP | 0x00000001 | True (i.e. set) if the OTP computation shall be for the next OTP, rather than the current one (current being interpreted in the context of the algorithm, e.g. for the current counter value or current time window). A Cryptoki library shall return CKR_MECHANISM_PARAM_INVALID if the CKF_NEXT OTP flag is set and the OTP mechanism in question does not support the concept of "next" OTP or the library is not capable of generating the next OTP ⁹ . |

⁹ Applications that may need to retrieve the next OTP should be prepared to handle this situation. For example, an application could store the OTP value returned by C_Sign so that, if a next OTP is required, it can compare it to the OTP value returned by subsequent calls to C_Sign should it turn out that the library does not support the CKF_NEXT OTP flag.

| Bit flag | Mask | Meaning |
|-----------------------|------------|---|
| CKF_EXCLUDE_TIME | 0x00000002 | True (i.e. set) if the OTP computation must not include a time value. Will have an effect only on mechanisms that do include a time value in the OTP computation and then only if the mechanism (and token) allows exclusion of this value. A Cryptoki library shall return CKR_MECHANISM_PARAM_INVALID if exclusion of the value is not allowed. |
| CKF_EXCLUDE_COUNTER | 0x00000004 | True (i.e. set) if the OTP computation must not include a counter value. Will have an effect only on mechanisms that do include a counter value in the OTP computation and then only if the mechanism (and token) allows exclusion of this value. A Cryptoki library shall return CKR_MECHANISM_PARAM_INVALID if exclusion of the value is not allowed. |
| CKF_EXCLUDE_CHALLENGE | 0x00000008 | True (i.e. set) if the OTP computation must not include a challenge. Will have an effect only on mechanisms that do include a challenge in the OTP computation and then only if the mechanism (and token) allows exclusion of this value. A Cryptoki library shall return CKR_MECHANISM_PARAM_INVALID if exclusion of the value is not allowed. |
| CKF_EXCLUDE_PIN | 0x00000010 | True (i.e. set) if the OTP computation must not include a PIN value. Will have an effect only on mechanisms that do include a PIN in the OTP computation and then only if the mechanism (and token) allows exclusion of this value. A Cryptoki library shall return CKR_MECHANISM_PARAM_INVALID if exclusion of the value is not allowed. |
| CKF_USER_FRIENDLY OTP | 0x00000020 | True (i.e. set) if the OTP returned shall be in a form suitable for human consumption. If this flag is set, and the call is successful, then the returned CK_OTP_VALUE shall be a UTF8-encoded printable string. A Cryptoki library shall return CKR_MECHANISM_PARAM_INVALID if this flag is set when CKA_OTP_USER_FRIENDLY_MODE for the key in question is CK_FALSE. |

13062 Note: Even if CKA_OTP_FORMAT is not set to CK_OTP_FORMAT_BINARY, then there may still be
 13063 value in setting the CKF_USER_FRIENDLY OTP flag (assuming CKA_OTP_USER_FRIENDLY_MODE
 13064 is CK_TRUE, of course) if the intent is for a human to read the generated OTP value, since it may
 13065 become shorter or otherwise better suited for a user. Applications that do not intend to provide a returned
 13066 OTP value to a user should not set the CKF_USER_FRIENDLY OTP flag.

◆ CK_OTP_PARAM; CK_OTP_PARAM_PTR

13068 **CK_OTP_PARAM** is a structure that includes the type, value, and length of an OTP parameter. It is
 13069 defined as follows:

```

13070     typedef struct CK OTP PARAM {
13071         CK OTP PARAM TYPE type;
13072         CK VOID PTR pValue;
13073         CK ULONG ulValueLen;
13074     } CK OTP PARAM;
13075 The fields of the structure have the following meanings:
13076             type      the parameter type
13077             pValue    pointer to the value of the parameter
13078             ulValueLen length in bytes of the value
13079 If a parameter has no value, then ulValueLen = 0, and the value of pValue is irrelevant. Note that pValue
13080 is a "void" pointer, facilitating the passing of arbitrary values. Both the application and the Cryptoki library
13081 must ensure that the pointer can be safely cast to the expected type (i.e., without word-alignment errors).
13082 CK OTP PARAM PTR is a pointer to a CK OTP PARAM.
13083
13084 ◆ CK OTP PARAMS; CK OTP PARAMS PTR
13085 CK OTP PARAMS is a structure that is used to provide parameters for OTP mechanisms in a generic
13086 fashion. It is defined as follows:
13087     typedef struct CK OTP PARAMS {
13088         CK OTP PARAM PTR pParams;
13089         CK ULONG ulCount;
13090     } CK OTP PARAMS;
13091 The fields of the structure have the following meanings:
13092             pParams   pointer to an array of OTP parameters
13093             ulCount   the number of parameters in the array
13094 CK OTP PARAMS PTR is a pointer to a CK OTP PARAMS.
13095
13096 When calling C_SignInit or C_VerifyInit with a mechanism that takes a CK OTP PARAMS structure as a
13097 parameter, the CK OTP PARAMS structure shall be populated in accordance with the
13098 CKA OTP X REQUIREMENT key attributes for the identified key, where X is PIN, CHALLENGE, TIME,
13099 or COUNTER.
13100 For example, if CKA OTP TIME REQUIREMENT = CK OTP PARAM MANDATORY, then the
13101 CK OTP TIME parameter shall be present. If CKA OTP TIME REQUIREMENT =
13102 CK OTP PARAM OPTIONAL, then a CK OTP TIME parameter may be present. If it is not present,
13103 then the library may collect it (during the C_Sign call). If CKA OTP TIME REQUIREMENT =
13104 CK OTP PARAM IGNORED, then a provided CK OTP TIME parameter will always be ignored.
13105 Additionally, a provided CK OTP TIME parameter will always be ignored if CKF EXCLUDE_TIME is set
13106 in a CK OTP FLAGS parameter. Similarly, if this flag is set, a library will not attempt to collect the value
13107 itself, and it will also instruct the token not to make use of any internal value, subject to token policies. It is
13108 an error (CKR_MECHANISM_PARAM_INVALID) to set the CKF EXCLUDE_TIME flag when the
13109 CKA OTP TIME REQUIREMENT attribute is CK OTP PARAM MANDATORY.
13110 The above discussion holds for all CKA OTP X REQUIREMENT attributes (i.e.,
13111 CKA OTP PIN REQUIREMENT, CKA OTP CHALLENGE REQUIREMENT,
13112 CKA OTP COUNTER REQUIREMENT, CKA OTP TIME REQUIREMENT). A library may set a
13113 particular CKA OTP X REQUIREMENT attribute to CK OTP PARAM OPTIONAL even if it is required
13114 by the mechanism as long as the token (or the library itself) has the capability of providing the value to the
13115 computation. One example of this is a token with an on-board clock.

```

13116 In addition, applications may use the CK OTP FLAGS, the CK OTP OUTPUT FORMAT and the
13117 CKA OTP LENGTH parameters to set additional parameters.

13118

13119 ◆ **CK OTP SIGNATURE INFO, CK OTP SIGNATURE INFO PTR**

13120 **CK OTP SIGNATURE INFO** is a structure that is returned by all OTP mechanisms in successful calls to
13121 **C_Sign** (**C_SignFinal**). The structure informs applications of actual parameter values used in particular
13122 OTP computations in addition to the OTP value itself. It is used by all mechanisms for which the key
13123 belongs to the class **CKO OTP KEY** and is defined as follows:

```
13124     typedef struct CK OTP SIGNATURE INFO {  
13125         CK OTP PARAM PTR pParams;  
13126         CK ULONG ulCount;  
13127     } CK OTP SIGNATURE INFO;
```

13128 The fields of the structure have the following meanings:

| | | |
|-------|---------|---|
| 13129 | pParams | pointer to an array of OTP parameter values |
| 13130 | ulCount | the number of parameters in the array |

13131 After successful calls to **C_Sign** or **C_SignFinal** with an OTP mechanism, the *pSignature* parameter will
13132 be set to point to a **CK OTP SIGNATURE INFO** structure. One of the parameters in this structure will be
13133 the OTP value itself, identified with the **CK OTP VALUE** tag. Other parameters may be present for
13134 informational purposes, e.g. the actual time used in the OTP calculation. In order to simplify OTP
13135 validations, authentication protocols may permit authenticating parties to send some or all of these
13136 parameters in addition to OTP values themselves. Applications should therefore check for their presence
13137 in returned **CK OTP SIGNATURE INFO** values whenever such circumstances apply.

13138 Since **C_Sign** and **C_SignFinal** follows the convention described in Section 5.2 on producing output, a
13139 call to **C_Sign** (or **C_SignFinal**) with *pSignature* set to **NULL_PTR** will return (in the *pulSignatureLen*
13140 parameter) the required number of bytes to hold the **CK OTP SIGNATURE INFO** structure as well as all
13141 the data in all its **CK OTP PARAM** components. If an application allocates a memory block based on
13142 this information, it shall therefore not subsequently de-allocate components of such a received value but
13143 rather de-allocate the complete **CK OTP PARAMS** structure itself. A Cryptoki library that is called with a
13144 non-**NULL** *pSignature* pointer will assume that it points to a *contiguous* memory block of the size
13145 indicated by the *pulSignatureLen* parameter.

13146 When verifying an OTP value using an OTP mechanism, *pSignature* shall be set to the OTP value itself,
13147 e.g. the value of the **CK OTP VALUE** component of a **CK OTP PARAM** structure returned by a call to
13148 **C_Sign**. The **CK OTP PARAM** value supplied in the **C_VerifyInit** call sets the values to use in the
13149 verification operation.

13150 **CK OTP SIGNATURE INFO PTR** points to a **CK OTP SIGNATURE INFO**.

13151 **6.53.8 RSA SecurID**

13152 **6.53.8.1 RSA SecurID secret key objects**

13153 RSA SecurID secret key objects (object class **CKO OTP KEY**, key type **CKK SECURID**) hold RSA
13154 SecurID secret keys. The following table defines the RSA SecurID secret key object attributes, in
13155 addition to the common attributes defined for this object class:

13156 *Table 229, RSA SecurID secret key object attributes*

| Attribute | Data type | Meaning |
|------------------------------------|-----------|--|
| CKA OTP TIME INTERVAL ¹ | CK ULONG | Interval between OTP values produced with this key, in seconds. Default is 60. |

13157 ¹Refer to Table 11 for footnotes

13158 The following is a sample template for creating an RSA SecurID secret key object:

```

13159     CK_OBJECT_CLASS class = CKO OTP KEY;
13160     CK_KEY_TYPE keyType = CKK SECURID;
13161     CK_DATE endDate = {...};
13162     CK_UTF8CHAR label[] = "RSA SecurID secret key object";
13163     CK_BYTE keyId[] = {...};
13164     CK ULONG outputFormat = CK OTP FORMAT DECIMAL;
13165     CK ULONG outputLength = 6;
13166     CK ULONG needPIN = CK OTP PARAM MANDATORY;
13167     CK ULONG timeInterval = 60;
13168     CK_BYTE value[] = {...};
13169     CK_BBOOL true = CK TRUE;
13170     CK_ATTRIBUTE template[] = {
13171         {CKA_CLASS, &class, sizeof(class)},
13172         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
13173         {CKA_END_DATE, &endDate, sizeof(endDate)},
13174         {CKA_TOKEN, &true, sizeof(true)},
13175         {CKA_SENSITIVE, &true, sizeof(true)},
13176         {CKA_LABEL, label, sizeof(label)-1},
13177         {CKA_SIGN, &true, sizeof(true)},
13178         {CKA_VERIFY, &true, sizeof(true)},
13179         {CKA_ID, keyId, sizeof(keyId)},
13180         {CKA OTP FORMAT, &outputFormat, sizeof(outputFormat)},
13181         {CKA OTP LENGTH, &outputLength, sizeof(outputLength)},
13182         {CKA OTP PIN REQUIREMENT, &needPIN, sizeof(needPIN)},
13183         {CKA OTP TIME INTERVAL, &timeInterval,
13184             sizeof(timeInterval)},
13185         {CKA_VALUE, value, sizeof(value)}}
13186     };

```

13187 6.53.8.2 RSA SecurID key generation

13188 The RSA SecurID key generation mechanism, denoted **CKM SECURID KEY GEN**, is a key generation
 13189 mechanism for the RSA SecurID algorithm.
 13190 It does not have a parameter.
 13191 The mechanism generates RSA SecurID keys with a particular set of attributes as specified in the
 13192 template for the key.
 13193 The mechanism contributes at least the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_VALUE_LEN**, and
 13194 **CKA_VALUE** attributes to the new key. Other attributes supported by the RSA SecurID key type may be
 13195 specified in the template for the key, or else are assigned default initial values.
 13196 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 13197 specify the supported range of SecurID key sizes, in bytes.

13198 6.53.8.3 SecurID OTP generation and validation

13199 **CKM SECURID** is the mechanism for the retrieval and verification of RSA SecurID OTP values.
 13200 The mechanism takes a pointer to a **CK OTP PARAMS** structure as a parameter.
 13201 When signing or verifying using the **CKM SECURID** mechanism, *pData* shall be set to **NULL_PTR** and
 13202 *ulDataLen* shall be set to 0.

- 13203 **6.53.8.4 Return values**
- 13204 Support for the CKM_SECURID mechanism extends the set of return values for C_Verify with the
 13205 following values:
- 13206 • CKR_NEW_PIN_MODE: The supplied OTP was not accepted and the library requests a new OTP
 13207 computed using a new PIN. The new PIN is set through means out of scope for this document.
 - 13208 • CKR_NEXT OTP: The supplied OTP was correct but indicated a larger than normal drift in the
 13209 token's internal state (e.g. clock, counter). To ensure this was not due to a temporary problem, the
 13210 application should provide the next one-time password to the library for verification.

13211 **6.53.9 OATH HOTP**

13212 **6.53.9.1 OATH HOTP secret key objects**

13213 HOTP secret key objects (object class **CKO OTP KEY**, key type **CKK_HOTP**) hold generic secret keys
 13214 and associated counter values.

13215 The **CKA OTP COUNTER** value may be set at key generation; however, some tokens may set it to a
 13216 fixed initial value. Depending on the token's security policy, this value may not be modified and/or may
 13217 not be revealed if the object has its **CKA SENSITIVE** attribute set to CK_TRUE or its
 13218 **CKA_EXTRACTABLE** attribute set to CK_FALSE.

13219 For HOTP keys, the **CKA OTP COUNTER** value shall be an 8 bytes unsigned integer in big endian (i.e.
 13220 network byte order) form. The same holds true for a **CK OTP COUNTER** value in a **CK OTP PARAM**
 13221 structure.

13222 The following is a sample template for creating a HOTP secret key object:

```

13223     CK_OBJECT_CLASS class = CKO OTP KEY;
13224     CK_KEY_TYPE keyType = CKK_HOTP;
13225     CK_UTF8CHAR label[] = "HOTP secret key object";
13226     CK_BYTE keyId[] = {...};
13227     CK ULONG outputFormat = CK OTP FORMAT DECIMAL;
13228     CK ULONG outputLength = 6;
13229     CK DATE endDate = {...};
13230     CK_BYTE counterValue[8] = {0};
13231     CK_BYTE value[] = {...};
13232         CK_BBOOL true = CK_TRUE;
13233     CK_ATTRIBUTE template[] = {
13234         {CKA CLASS, &class, sizeof(class)},
13235         {CKA KEY TYPE, &keyType, sizeof(keyType)},
13236         {CKA END DATE, &endDate, sizeof(endDate)},
13237         {CKA TOKEN, &true, sizeof(true)},
13238         {CKA SENSITIVE, &true, sizeof(true)},
13239         {CKA LABEL, label, sizeof(label)-1},
13240         {CKA SIGN, &true, sizeof(true)},
13241         {CKA VERIFY, &true, sizeof(true)},
13242         {CKA ID, keyId, sizeof(keyId)},
13243         {CKA OTP FORMAT, &outputFormat, sizeof(outputFormat)},
13244         {CKA OTP LENGTH, &outputLength, sizeof(outputLength)},
13245         {CKA OTP COUNTER, counterValue, sizeof(counterValue)},
13246         {CKA VALUE, value, sizeof(value)}
13247     };

```

13248 **6.53.9.2 HOTP key generation**

13249 The HOTP key generation mechanism, denoted **CKM_HOTP_KEY_GEN**, is a key generation mechanism
13250 for the HOTP algorithm.

13251 It does not have a parameter.

13252 The mechanism generates HOTP keys with a particular set of attributes as specified in the template for
13253 the key.

13254 The mechanism contributes at least the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA OTP COUNTER**,
13255 **CKA_VALUE** and **CKA_VALUE_LEN** attributes to the new key. Other attributes supported by the HOTP
13256 key type may be specified in the template for the key, or else are assigned default initial values.

13257 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
13258 specify the supported range of HOTP key sizes, in bytes.

13259 **6.53.9.3 HOTP OTP generation and validation**

13260 **CKM_HOTP** is the mechanism for the retrieval and verification of HOTP OTP values based on the current
13261 internal counter, or a provided counter.

13262 The mechanism takes a pointer to a **CK OTP PARAMS** structure as a parameter.

13263 As for the **CKM_SECURID** mechanism, when signing or verifying using the **CKM_HOTP** mechanism,
13264 *pData* shall be set to **NULL_PTR** and *ulDataLen* shall be set to 0.

13265 For verify operations, the counter value **CK OTP COUNTER** must be provided as a **CK OTP PARAM**
13266 parameter to **C VerifyInit**. When verifying an OTP value using the **CKM_HOTP** mechanism, *pSignature*
13267 shall be set to the OTP value itself, e.g. the value of the **CK OTP VALUE** component of a
13268 **CK OTP PARAM** structure in the case of an earlier call to **C Sign**.

13269 **6.53.10 ActivIdentity ACTI**

13270 **6.53.10.1 ACTI secret key objects**

13271 ACTI secret key objects (object class **CKO OTP KEY**, key type **CKK_ACTI**) hold ActivIdentity ACTI
13272 secret keys.

13273 For ACTI keys, the **CKA OTP COUNTER** value shall be an 8 bytes unsigned integer in big endian (i.e.
13274 network byte order) form. The same holds true for the **CK OTP COUNTER** value in the
13275 **CK OTP PARAM** structure.

13276 The **CKA OTP COUNTER** value may be set at key generation; however, some tokens may set it to a
13277 fixed initial value. Depending on the token's security policy, this value may not be modified and/or may
13278 not be revealed if the object has its **CKA SENSITIVE** attribute set to CK_TRUE or its
13279 **CKA EXTRACTABLE** attribute set to CK_FALSE.

13280 The **CKA OTP TIME** value may be set at key generation; however, some tokens may set it to a fixed
13281 initial value. Depending on the token's security policy, this value may not be modified and/or may not be
13282 revealed if the object has its **CKA SENSITIVE** attribute set to CK_TRUE or its **CKA EXTRACTABLE**
13283 attribute set to CK_FALSE.

13284 The following is a sample template for creating an ACTI secret key object:

```
13285     CK_OBJECT_CLASS class = CKO OTP KEY;  
13286     CK_KEY_TYPE keyType = CKK_ACTI;  
13287     CK_UTF8CHAR label[] = "ACTI secret key object";  
13288     CK_BYTE keyId[] = {...};  
13289     CK ULONG outputFormat = CK OTP FORMAT DECIMAL;  
13290     CK ULONG outputLength = 6;  
13291     CK_DATE endDate = {...};  
13292     CK_BYTE counterValue[8] = {0};
```

```

13293     CK_BYTE value[] = { ... };
13294     CK_BBOOL true = CK_TRUE;
13295     CK_ATTRIBUTE template[] = {
13296         { CKA_CLASS, &class, sizeof(class) },
13297         { CKA_KEY_TYPE, &keyType, sizeof(keyType) },
13298         { CKA_END_DATE, &endDate, sizeof(endDate) },
13299         { CKA_TOKEN, &true, sizeof(true) },
13300         { CKA_SENSITIVE, &true, sizeof(true) },
13301         { CKA_LABEL, label, sizeof(label)-1 },
13302         { CKA_SIGN, &true, sizeof(true) },
13303         { CKA_VERIFY, &true, sizeof(true) },
13304         { CKA_ID, keyId, sizeof(keyId) },
13305         { CKA OTP_FORMAT, &outputFormat,
13306             sizeof(outputFormat) },
13307         { CKA OTP_LENGTH, &outputLength,
13308             sizeof(outputLength) },
13309         { CKA OTP_COUNTER, counterValue,
13310             sizeof(counterValue) },
13311         { CKA_VALUE, value, sizeof(value) }
13312     };

```

13313 **6.53.10.2 ACTI key generation**

13314 The ACTI key generation mechanism, denoted **CKM_ACTI_KEY_GEN**, is a key generation mechanism
 13315 for the ACTI algorithm.

13316 It does not have a parameter.

13317 The mechanism generates ACTI keys with a particular set of attributes as specified in the template for the
 13318 key.

13319 The mechanism contributes at least the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_VALUE** and
 13320 **CKA_VALUE_LEN** attributes to the new key. Other attributes supported by the ACTI key type may be
 13321 specified in the template for the key, or else are assigned default initial values.

13322 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 13323 specify the supported range of ACTI key sizes, in bytes.

13324 **6.53.10.3 ACTI OTP generation and validation**

13325 **CKM_ACTI** is the mechanism for the retrieval and verification of ACTI OTP values.

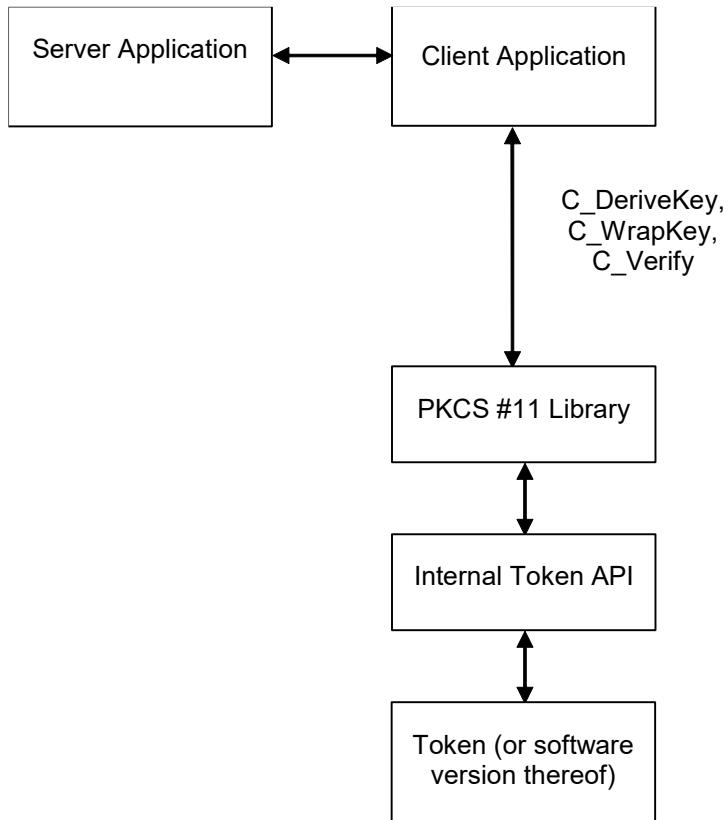
13326 The mechanism takes a pointer to a **CK OTP_PARAMS** structure as a parameter.

13327 When signing or verifying using the **CKM_ACTI** mechanism, *pData* shall be set to **NULL_PTR** and
 13328 *ulDataLen* shall be set to 0.

13329 When verifying an OTP value using the **CKM_ACTI** mechanism, *pSignature* shall be set to the OTP value
 13330 itself, e.g. the value of the **CK OTP_VALUE** component of a **CK OTP_PARAM** structure in the case of
 13331 an earlier call to **C_Sign**.

13332 **6.54 CT-KIP**

13333 **6.54.1 Principles of Operation**



13334

13335 *Figure 5: PKCS #11 and CT-KIP integration*

13336 Figure 5 shows an integration of PKCS #11 into an application that generates cryptographic keys through
13337 the use of CT-KIP. The application invokes **C_DeriveKey** to derive a key of a particular type on the token.
13338 The key may subsequently be used as a basis to e.g., generate one-time password values. The
13339 application communicates with a CT-KIP server that participates in the key derivation and stores a copy
13340 of the key in its database. The key is transferred to the server in wrapped form, after a call to
13341 **C_WrapKey**. The server authenticates itself to the client and the client verifies the authentication by calls
13342 to **C_Verify**.

13343 **6.54.2 Mechanisms**

13344 The following table shows, for the mechanisms defined in this document, their support by different
13345 cryptographic operations. For any particular token, of course, a particular operation may well support
13346 only a subset of the mechanisms listed. There is also no guarantee that a token that supports one
13347 mechanism for some operation supports any other mechanism for any other operation (or even supports
13348 that same mechanism for any other operation).

13349 *Table 230: CT-KIP Mechanisms vs. applicable functions*

| Mechanism | Functions | | | | | | |
|----------------|-------------------|---------------|----------------------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_KIP_DERIVE | | | | | | | ✓ |
| CKM_KIP_WRAP | | | | | | ✓ | |
| CKM_KIP_MAC | | ✓ | | | | | |

13350 The remainder of this section will present in detail the mechanisms and the parameters that are supplied
 13351 to them.

13352 6.54.3 Definitions

13353 Mechanisms:

13354 CKM_KIP_DERIVE

13355 CKM_KIP_WRAP

13356 CKM_KIP_MAC

13357 6.54.4 CT-KIP Mechanism parameters

13358 ♦ CK_KIP_PARAMS; CK_KIP_PARAMS_PTR

13359 CK_KIP_PARAMS is a structure that provides the parameters to all the CT-KIP related mechanisms: The
 13360 CKM_KIP_DERIVE key derivation mechanism, the CKM_KIP_WRAP key wrap and key unwrap
 13361 mechanism, and the CKM_KIP_MAC signature mechanism. The structure is defined as follows:

```
13362     typedef struct CK_KIP_PARAMS {
13363         CK_MECHANISM_PTR pMechanism;
13364         CK_OBJECT_HANDLE hKey;
13365         CK_BYTE_PTR pSeed;
13366         CK_ULONG ulSeedLen;
13367     } CK_KIP_PARAMS;
```

13368 The fields of the structure have the following meanings:

13369 pMechanism pointer to the underlying cryptographic mechanism (e.g. AES, SHA-
 13370 256)

13371 hKey handle to a key that will contribute to the entropy of the derived key
 13372 (CKM_KIP_DERIVE) or will be used in the MAC operation
 13373 (CKM_KIP_MAC)

13374 pSeed pointer to an input seed

13375 ulSeedLen length in bytes of the input seed

13376 CK_KIP_PARAMS_PTR is a pointer to a CK_KIP_PARAMS structure.

13377 6.54.5 CT-KIP key derivation

13378 The CT-KIP key derivation mechanism, denoted CKM_KIP_DERIVE, is a key derivation mechanism that
 13379 is capable of generating secret keys of potentially any type, subject to token limitations.

13380 It takes a parameter of type CK_KIP_PARAMS which allows for the passing of the desired underlying
 13381 cryptographic mechanism as well as some other data. In particular, when the hKey parameter is a handle

13382 to an existing key, that key will be used in the key derivation in addition to the *hBaseKey* of **C_DeriveKey**.
13383 The *pSeed* parameter may be used to seed the key derivation operation.
13384 The mechanism derives a secret key with a particular set of attributes as specified in the attributes of the
13385 template for the key.
13386 The mechanism contributes the **CKA_CLASS** and **CKA_VALUE** attributes to the new key. Other
13387 attributes supported by the key type may be specified in the template for the key, or else will be assigned
13388 default initial values. Since the mechanism is generic, the **CKA_KEY_TYPE** attribute should be set in the
13389 template, if the key is to be used with a particular mechanism.

13390 **6.54.6 CT-KIP key wrap and key unwrap**

13391 The CT-KIP key wrap and unwrap mechanism, denoted **CKM_KIP_WRAP**, is a key wrap mechanism that
13392 is capable of wrapping and unwrapping generic secret keys.
13393 It takes a parameter of type **CK_KIP_PARAMS**, which allows for the passing of the desired underlying
13394 cryptographic mechanism as well as some other data. It does not make use of the *hKey* parameter of
13395 **CK_KIP_PARAMS**.

13396 **6.54.7 CT-KIP signature generation**

13397 The CT-KIP signature (MAC) mechanism, denoted **CKM_KIP_MAC**, is a mechanism used to produce a
13398 message authentication code of arbitrary length. The keys it uses are secret keys.
13399 It takes a parameter of type **CK_KIP_PARAMS**, which allows for the passing of the desired underlying
13400 cryptographic mechanism as well as some other data. The mechanism does not make use of the *pSeed*
13401 and the *uSeedLen* parameters of **CT_KIP_PARAMS**.
13402 This mechanism produces a MAC of the length specified by *puSignatureLen* parameter in calls to
13403 **C_Sign**.
13404 If a call to **C_Sign** with this mechanism fails, then no output will be generated.

13405 **6.55 GOST 28147-89**

13406 GOST 28147-89 is a block cipher with 64-bit block size and 256-bit keys.

13407

13408 *Table 231, GOST 28147-89 Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|------------------------|-------------------|---------------|---------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_GOST28147_KEY_GEN | | | | | ✓ | | |
| CKM_GOST28147_ECB | ✓ | | | | | ✓ | |
| CKM_GOST28147 | ✓ | | | | | ✓ | |
| CKM_GOST28147_MAC | | ✓ | | | | | |
| CKM_GOST28147_KEY_WRAP | | | | | | ✓ | |

13409

13410 **6.55.1 Definitions**

13411 This section defines the key type “CKK_GOST28147” for type **CK_KEY_TYPE** as used in the
13412 **CKA_KEY_TYPE** attribute of key objects and domain parameter objects.

13413 Mechanisms:
 13414 CKM_GOST28147_KEY_GEN
 13415 CKM_GOST28147_ECB
 13416 CKM_GOST28147
 13417 CKM_GOST28147_MAC
 13418 CKM_GOST28147_KEY_WRAP

6.55.2 GOST 28147-89 secret key objects

13420 GOST 28147-89 secret key objects (object class **CKO_SECRET_KEY**, key type **CKK_GOST28147**) hold
 13421 GOST 28147-89 keys. The following table defines the GOST 28147-89 secret key object attributes, in
 13422 addition to the common attributes defined for this object class:

13423 *Table 232, GOST 28147-89 Secret Key Object Attributes*

| Attribute | Data type | Meaning |
|---------------------------------------|------------|--|
| CKA_VALUE ^{1,4,6,7} | Byte array | 32 bytes in little endian order |
| CKA_GOST28147_PARAMS ^{1,3,5} | Byte array | <p>DER-encoding of the object identifier indicating the data object type of GOST 28147-89.</p> <p>When key is used the domain parameter object of key type CKK_GOST28147 must be specified with the same attribute CKA_OBJECT_ID</p> |

13424 - Refer to Table 11 for footnotes

13425 The following is a sample template for creating a GOST 28147-89 secret key object:

```

13426 CK_OBJECT_CLASS class = CKO_SECRET_KEY;
13427 CK_KEY_TYPE keyType = CKK_GOST28147;
13428 CK_UTF8CHAR label[] = "A GOST 28147-89 secret key object";
13429 CK_BYTE value[32] = {...};
13430 CK_BYTE params_oid[] = {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02,
13431           0x02, 0x1f, 0x00};
13432 CK_BBOOL true = CK_TRUE;
13433 CK_ATTRIBUTE template[] = {
13434     {CKA_CLASS, &class, sizeof(class)},
13435     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
13436     {CKA_TOKEN, &true, sizeof(true)},
13437     {CKA_LABEL, label, sizeof(label)-1},
13438     {CKA_ENCRYPT, &true, sizeof(true)},
13439     {CKA_GOST28147_PARAMS, params_oid, sizeof(params_oid)},
13440     {CKA_VALUE, value, sizeof(value)}}
13441 };
  
```

6.55.3 GOST 28147-89 domain parameter objects

13443 GOST 28147-89 domain parameter objects (object class **CKO_DOMAIN_PARAMETERS**, key type
 13444 **CKK_GOST28147**) hold GOST 28147-89 domain parameters.

13445 The following table defines the GOST 28147-89 domain parameter object attributes, in addition to the
 13446 common attributes defined for this object class:

13447 Table 233, GOST 28147-89 Domain Parameter Object Attributes

| Attribute | Data Type | Meaning |
|----------------------------|------------|--|
| CKA_VALUE ¹ | Byte array | DER-encoding of the domain parameters as it was introduced in [4] section 8.1 (type <i>Gost28147-89-ParamSetParameters</i>) |
| CKA_OBJECT_ID ¹ | Byte array | DER-encoding of the object identifier indicating the domain parameters |

13448 Refer to Table 11 for footnotes

13449 For any particular token, there is no guarantee that a token supports domain parameters loading up
 13450 and/or fetching out. Furthermore, applications, that make direct use of domain parameters objects, should
 13451 take in account that **CKA_VALUE** attribute may be inaccessible.

13452 The following is a sample template for creating a GOST 28147-89 domain parameter object:

```

13453 CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
13454 CK_KEY_TYPE keyType = CKK_GOST28147;
13455 CK_UTF8CHAR label[] = "A GOST 28147-89 cryptographic
13456     parameters object";
13457 CK_BYTE oid[] = {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02,
13458                 0x1f, 0x00};
13459 CK_BYTE value[] = {
13460     0x30, 0x62, 0x04, 0x40, 0x4c, 0xde, 0x38, 0x9c, 0x29, 0x89, 0xef, 0xb6,
13461     0xff, 0xeb, 0x56, 0xc5, 0x5e, 0xc2, 0x9b, 0x02, 0x98, 0x75, 0x61, 0x3b,
13462     0x11, 0x3f, 0x89, 0x60, 0x03, 0x97, 0x0c, 0x79, 0x8a, 0xa1, 0xd5, 0x5d,
13463     0xe2, 0x10, 0xad, 0x43, 0x37, 0x5d, 0xb3, 0x8e, 0xb4, 0x2c, 0x77, 0xe7,
13464     0xcd, 0x46, 0xca, 0xfa, 0xd6, 0x6a, 0x20, 0x1f, 0x70, 0xf4, 0x1e, 0xa4,
13465     0xab, 0x03, 0xf2, 0x21, 0x65, 0xb8, 0x44, 0xd8, 0x02, 0x01, 0x00, 0x02,
13466     0x01, 0x40, 0x30, 0x0b, 0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x0e,
13467     0x00, 0x05, 0x00
13468 };
13469 CK_BBOOL true = CK_TRUE;
13470 CK_ATTRIBUTE template[] = {
13471     {CKA_CLASS, &class, sizeof(class)},
13472     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
13473     {CKA_TOKEN, &true, sizeof(true)},
13474     {CKA_LABEL, label, sizeof(label)-1},
13475     {CKA_OBJECT_ID, oid, sizeof(oid)},
13476     {CKA_VALUE, value, sizeof(value)}
13477 };

```

13478 6.55.4 GOST 28147-89 key generation

13479 The GOST 28147-89 key generation mechanism, denoted **CKM_GOST28147_KEY_GEN**, is a key
 13480 generation mechanism for GOST 28147-89.

13481 It does not have a parameter.

13482 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
 13483 key. Other attributes supported by the GOST 28147-89 key type may be specified for objects of object
 13484 class **CKO_SECRET_KEY**.

13485 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** are not
 13486 used.

13487 6.55.5 GOST 28147-89-ECB

13488 GOST 28147-89-ECB, denoted **CKM_GOST28147_ECB**, is a mechanism for single and multiple-part
13489 encryption and decryption; key wrapping; and key unwrapping, based on GOST 28147-89 and electronic
13490 codebook mode.

13491 It does not have a parameter.

13492 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to
13493 wrap/unwrap every secret key that it supports.

13494 For wrapping (**C_WrapKey**), the mechanism encrypts the value of the **CKA_VALUE** attribute of the key
13495 that is wrapped, padded on the trailing end with up to block size so that the resulting length is a multiple
13496 of the block size.

13497 For unwrapping (**C_UnwrapKey**), the mechanism decrypts the wrapped key, and truncates the result
13498 according to the **CKA_KEY_TYPE** attribute of the template and, if it has one, and the key type supports
13499 it, the **CKA_VALUE_LEN** attribute of the template. The mechanism contributes the result as the
13500 **CKA_VALUE** attribute of the new key.

13501 Constraints on key types and the length of data are summarized in the following table:

13502 *Table 234, GOST 28147-89-ECB: Key and Data Length*

| Function | Key type | Input length | Output length |
|-------------|---------------|------------------------|---|
| C_Encrypt | CKK_GOST28147 | Multiple of block size | Same as input length |
| C_Decrypt | CKK_GOST28147 | Multiple of block size | Same as input length |
| C_WrapKey | CKK_GOST28147 | Any | Input length rounded up to multiple of block size |
| C_UnwrapKey | CKK_GOST28147 | Multiple of block size | Determined by type of key being unwrapped |

13503

13504 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
13505 are not used.

13506 6.55.6 GOST 28147-89 encryption mode except ECB

13507 GOST 28147-89 encryption mode except ECB, denoted **CKM_GOST28147**, is a mechanism for single
13508 and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on
13509 [GOST 28147-89] and CFB, counter mode, and additional CBC mode defined in [RFC 4357] section 2.
13510 Encryption's parameters are specified in object identifier of attribute **CKA_GOST28147_PARAMS**.

13511 It has a parameter, which is an 8-byte initialization vector. This parameter may be omitted then a zero
13512 initialization vector is used.

13513 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to
13514 wrap/unwrap every secret key that it supports.

13515 For wrapping (**C_WrapKey**), the mechanism encrypts the value of the **CKA_VALUE** attribute of the key
13516 that is wrapped.

13517 For unwrapping (**C_UnwrapKey**), the mechanism decrypts the wrapped key, and contributes the result as
13518 the **CKA_VALUE** attribute of the new key.

13519 Constraints on key types and the length of data are summarized in the following table:

13520 *Table 235, GOST 28147-89 encryption modes except ECB: Key and Data Length*

| Function | Key type | Input length | Output length |
|-------------|---------------|--------------|---|
| C_Encrypt | CKK_GOST28147 | Any | For counter mode and CFB is the same as input length. For CBC is the same as input length padded on the trailing end with up to block size so that the resulting length is a multiple of the block size |
| C_Decrypt | CKK_GOST28147 | Any | |
| C_WrapKey | CKK_GOST28147 | Any | |
| C_UnwrapKey | CKK_GOST28147 | Any | |

13521

13522 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
13523 are not used.

13524 6.55.7 GOST 28147-89-MAC

13525 GOST 28147-89-MAC, denoted **CKM_GOST28147_MAC**, is a mechanism for data integrity and
13526 authentication based on GOST 28147-89 and key meshing algorithms [RFC 4357] section 2.3.

13527 MACing parameters are specified in object identifier of attribute **CKA_GOST28147_PARAMS**.

13528 The output bytes from this mechanism are taken from the start of the final GOST 28147-89 cipher block
13529 produced in the MACing process.

13530 It has a parameter, which is an 8-byte MAC initialization vector. This parameter may be omitted then a
13531 zero initialization vector is used.

13532 Constraints on key types and the length of data are summarized in the following table:

13533 *Table 236, GOST28147-89-MAC: Key and Data Length*

| Function | Key type | Data length | Signature length |
|----------|---------------|-------------|------------------|
| C_Sign | CKK_GOST28147 | Any | 4 bytes |
| C_Verify | CKK_GOST28147 | Any | 4 bytes |

13534

13535 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
13536 are not used.

13537

13538 6.55.8 GOST 28147-89 keys wrapping/unwrapping with GOST 28147-89

13539 GOST 28147-89 keys as a KEK (key encryption keys) for encryption GOST 28147-89 keys, denoted by
13540 **CKM_GOST28147_KEY_WRAP**, is a mechanism for key wrapping; and key unwrapping, based on
13541 GOST 28147-89. Its purpose is to encrypt and decrypt keys have been generated by key generation
13542 mechanism for GOST 28147-89.

13543 For wrapping (**C_WrapKey**), the mechanism first computes MAC from the value of the **CKA_VALUE**
13544 attribute of the key that is wrapped and then encrypts in ECB mode the value of the **CKA_VALUE**
13545 attribute of the key that is wrapped. The result is 32 bytes of the key that is wrapped and 4 bytes of MAC.

13546 For unwrapping (**C_UnwrapKey**), the mechanism first decrypts in ECB mode the 32 bytes of the key that
13547 was wrapped and then computes MAC from the unwrapped key. Then compared together 4 bytes MAC
13548 has computed and 4 bytes MAC of the input. If these two MACs do not match the wrapped key is
13549 disallowed. The mechanism contributes the result as the **CKA_VALUE** attribute of the unwrapped key.

13550 It has a parameter, which is an 8-byte MAC initialization vector. This parameter may be omitted then a
13551 zero initialization vector is used.

13552 Constraints on key types and the length of data are summarized in the following table:

13553 Table 237, GOST 28147-89 keys as KEK: Key and Data Length

| Function | Key type | Input length | Output length |
|-------------|---------------|--------------|---------------|
| C_WrapKey | CKK_GOST28147 | 32 bytes | 36 bytes |
| C_UnwrapKey | CKK_GOST28147 | 32 bytes | 36 bytes |

13554

13555 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
13556 are not used.

13557

13558 6.56 GOST R 34.11-94

13559 GOST R 34.11-94 is a mechanism for message digesting, following the hash algorithm with 256-bit
13560 message digest defined in [GOST R 34.11-94].

13561

13562 Table 238, GOST R 34.11-94 Mechanisms vs. Functions

| Mechanism | Functions | | | | | | |
|--------------------|-------------------|---------------|---------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_GOSTR3411 | | | | ✓ | | | |
| CKM_GOSTR3411_HMAC | | ✓ | | | | | |

13563

13564 6.56.1 Definitions

13565 This section defines the key type "CKK_GOSTR3411" for type CK_KEY_TYPE as used in the
13566 CKA_KEY_TYPE attribute of domain parameter objects.

13567 Mechanisms:

13568 CKM_GOSTR3411

13569 CKM_GOSTR3411_HMAC

13570 6.56.2 GOST R 34.11-94 domain parameter objects

13571 GOST R 34.11-94 domain parameter objects (object class **CKO_DOMAIN_PARAMETERS**, key type
13572 **CKK_GOSTR3411**) hold GOST R 34.11-94 domain parameters.

13573 The following table defines the GOST R 34.11-94 domain parameter object attributes, in addition to the
13574 common attributes defined for this object class:

13575 Table 239, GOST R 34.11-94 Domain Parameter Object Attributes

| Attribute | Data Type | Meaning |
|----------------------------|------------|--|
| CKA_VALUE ¹ | Byte array | DER-encoding of the domain parameters as it was introduced in [4] section 8.2 (type <i>GostR3411-94-ParamSetParameters</i>) |
| CKA_OBJECT_ID ¹ | Byte array | DER-encoding of the object identifier indicating the domain parameters |

13576 ¹ Refer to Table 11 for footnotes

13577 For any particular token, there is no guarantee that a token supports domain parameters loading up
 13578 and/or fetching out. Furthermore, applications, that make direct use of domain parameters objects, should
 13579 take in account that **CKA_VALUE** attribute may be inaccessible.

13580 The following is a sample template for creating a GOST R 34.11-94 domain parameter object:

```

13581 CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
13582 CK_KEY_TYPE keyType = CKK_GOSTR3411;
13583 CK_UTF8CHAR label[] = "A GOST R34.11-94 cryptographic
13584     parameters object";
13585 CK_BYTE oid[] = {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02,
13586     0x1e, 0x00};
13587 CK_BYTE value[] = {
13588     0x30, 0x64, 0x04, 0x40, 0x4e, 0x57, 0x64, 0xd1, 0xab, 0x8d, 0xcb, 0xbf,
13589     0x94, 0x1a, 0x7a, 0x4d, 0x2c, 0xd1, 0x10, 0x10, 0xd6, 0xa0, 0x57, 0x35,
13590     0x8d, 0x38, 0xf2, 0xf7, 0x0f, 0x49, 0xd1, 0x5a, 0xea, 0x2f, 0x8d, 0x94,
13591     0x62, 0xee, 0x43, 0x09, 0xb3, 0xf4, 0xa6, 0xa2, 0x18, 0xc6, 0x98, 0xe3,
13592     0xc1, 0x7c, 0xe5, 0x7e, 0x70, 0x6b, 0x09, 0x66, 0xf7, 0x02, 0x3c, 0x8b,
13593     0x55, 0x95, 0xbf, 0x28, 0x39, 0xb3, 0x2e, 0xcc, 0x04, 0x20, 0x00, 0x00,
13594     0x00, 0x00,
13595     0x00, 0x00,
13596     0x00, 0x00
13597 };
13598 CK_BBOOL true = CK_TRUE;
13599 CK_ATTRIBUTE template[] = {
13600     {CKA_CLASS, &class, sizeof(class)},
13601     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
13602     {CKA_TOKEN, &true, sizeof(true)},
13603     {CKA_LABEL, label, sizeof(label)-1},
13604     {CKA_OBJECT_ID, oid, sizeof(oid)},
13605     {CKA_VALUE, value, sizeof(value)}
13606 };

```

13607 6.56.3 GOST R 34.11-94 digest

13608 GOST R 34.11-94 digest, denoted **CKM_GOSTR3411**, is a mechanism for message digesting based on
 13609 GOST R 34.11-94 hash algorithm [GOST R 34.11-94].

13610 As a parameter this mechanism utilizes a DER-encoding of the object identifier. A mechanism parameter
 13611 may be missed then parameters of the object identifier *id-GostR3411-94-CryptoProParamSet* [RFC 4357]
 13612 (section 11.2) must be used.

13613 Constraints on the length of input and output data are summarized in the following table. For single-part
 13614 digesting, the data and the digest may begin at the same location in memory.

13615 *Table 240, GOST R 34.11-94: Data Length*

| Function | Input length | Digest length |
|----------|--------------|---------------|
| C_Digest | Any | 32 bytes |

13616

13617 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 13618 are not used.

6.56.4 GOST R 34.11-94 HMAC

GOST R 34.11-94 HMAC mechanism, denoted **CKM_GOSTR3411_HMAC**, is a mechanism for signatures and verification. It uses the HMAC construction, based on the GOST R 34.11-94 hash function [GOST R 34.11-94] and core HMAC algorithm [RFC 2104]. The keys it uses are of generic key type **CKK_GENERIC_SECRET** or **CKK_GOST28147**.

To be conformed to GOST R 34.11-94 hash algorithm [GOST R 34.11-94] the block length of core HMAC algorithm is 32 bytes long (see [RFC 2104] section 2, and [RFC 4357] section 3).

As a parameter this mechanism utilizes a DER-encoding of the object identifier. A mechanism parameter may be missed then parameters of the object identifier *id-GostR3411-94-CryptoProParamSet* [RFC 4357] (section 11.2) must be used.

Signatures (MACs) produced by this mechanism are of 32 bytes long.

Constraints on the length of input and output data are summarized in the following table:

Table 241, GOST R 34.11-94 HMAC: Key And Data Length

| Function | Key type | Data length | Signature length |
|----------|-------------------------------------|-------------|------------------|
| C_Sign | CKK_GENERIC_SECRET or CKK_GOST28147 | Any | 32 byte |
| C_Verify | CKK_GENERIC_SECRET or CKK_GOST28147 | Any | 32 bytes |

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure are not used.

6.57 GOST R 34.10-2001

GOST R 34.10-2001 is a mechanism for single- and multiple-part signatures and verification, following the digital signature algorithm defined in [GOST R 34.10-2001].

13637

Table 242, GOST R34.10-2001 Mechanisms vs. Functions

| Mechanism | Functions | | | | | | |
|------------------------------|-------------------|----------------|-----------|--------|---------------------|---------------|---------|
| | Encrypt & Decrypt | Sign & Verify | S R & V R | Digest | Gen . Key/ Key Pair | Wrap & Unwrap | Deriv e |
| CKM_GOSTR3410_KEY_PAIR_GEN | | | | | ✓ | | |
| CKM_GOSTR3410 | | ✓ ¹ | | | | | |
| CKM_GOSTR3410_WITH_GOSTR3411 | | ✓ | | | | | |
| CKM_GOSTR3410_KEY_WRAP | | | | | | ✓ | |
| CKM_GOSTR3410_DERIVE | | | | | | | ✓ |

¹ Single-part operations only

13640

6.57.1 Definitions

This section defines the key type "CKK_GOSTR3410" for type **CK_KEY_TYPE** as used in the **CKA_KEY_TYPE** attribute of key objects and domain parameter objects.

Mechanisms:

13645 CKM_GOSTR3410_KEY_PAIR_GEN
 13646 CKM_GOSTR3410
 13647 CKM_GOSTR3410_WITH_GOSTR3411
 13648 CKM_GOSTR3410
 13649 CKM_GOSTR3410_KEY_WRAP
 13650 CKM_GOSTR3410_DERIVE

13651 6.57.2 GOST R 34.10-2001 public key objects

13652 GOST R 34.10-2001 public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_GOSTR3410**)
 13653 hold GOST R 34.10-2001 public keys.

13654 The following table defines the GOST R 34.10-2001 public key object attributes, in addition to the
 13655 common attributes defined for this object class:

13656 *Table 243, GOST R 34.10-2001 Public Key Object Attributes*

| Attribute | Data Type | Meaning |
|---------------------------------------|------------|---|
| CKA_VALUE ^{1,4} | Byte array | 64 bytes for public key; 32 bytes for each coordinates X and Y of Elliptic Curve point P(X, Y) in little endian order |
| CKA_GOSTR3410_PARAMS ^{1,3} | Byte array | DER-encoding of the object identifier indicating the data object type of GOST R 34.10-2001. When key is used the domain parameter object of key type CKK_GOSTR3410 must be specified with the same attribute CKA_OBJECT_ID |
| CKA_GOSTR3411_PARAMS ^{1,3,8} | Byte array | DER-encoding of the object identifier indicating the data object type of GOST R 34.11-94. When key is used the domain parameter object of key type CKK_GOSTR3411 must be specified with the same attribute CKA_OBJECT_ID |
| CKA_GOST28147_PARAMS ⁸ | Byte array | DER-encoding of the object identifier indicating the data object type of GOST 28147-89. When key is used the domain parameter object of key type CKK_GOST28147 must be specified with the same attribute CKA_OBJECT_ID. The attribute value may be omitted |

13657 ¹Refer to Table 11 for footnotes

13658 The following is a sample template for creating an GOST R 34.10-2001 public key object:

```

13659 CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
13660 CK_KEY_TYPE keyType = CKK_GOSTR3410;
13661 CK_UTF8CHAR label[] = "A GOST R34.10-2001 public key object";
13662 CK_BYTE gostR3410params_oid[] =
13663     {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x23, 0x00};
13664 CK_BYTE gostR3411params_oid[] =

```

```

13665     { 0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1e, 0x00 };
13666     CK_BYTE gost28147params_oid[] =
13667         { 0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1f, 0x00 };
13668     CK_BYTE value[64] = {...};
13669     CK_BBOOL true = CK_TRUE;
13670     CK_ATTRIBUTE template[] = {
13671         { CKA_CLASS, &class, sizeof(class) },
13672         { CKA_KEY_TYPE, &keyType, sizeof(keyType) },
13673         { CKA_TOKEN, &true, sizeof(true) },
13674         { CKA_LABEL, label, sizeof(label)-1 },
13675         { CKA_GOSTR3410_PARAMS, gostR3410params_oid,
13676             sizeof(gostR3410params_oid) },
13677         { CKA_GOSTR3411_PARAMS, gostR3411params_oid,
13678             sizeof(gostR3411params_oid) },
13679         { CKA_GOST28147_PARAMS, gost28147params_oid,
13680             sizeof(gost28147params_oid) },
13681         { CKA_VALUE, value, sizeof(value) }
13682     };

```

13683 6.57.3 GOST R 34.10-2001 private key objects

13684 GOST R 34.10-2001 private key objects (object class **CKO_PRIVATE_KEY**, key type
13685 **CKK_GOSTR3410**) hold GOST R 34.10-2001 private keys.

13686 The following table defines the GOST R 34.10-2001 private key object attributes, in addition to the
13687 common attributes defined for this object class:

13688 *Table 244, GOST R 34.10-2001 Private Key Object Attributes*

| Attribute | Data Type | Meaning |
|---|------------|---|
| CKA_VALUE ^{1,4,6,7} | Byte array | 32 bytes for private key in little endian order |
| CKA_GOSTR3410_PARAMS ^{1,4,6} | Byte array | DER-encoding of the object identifier indicating the data object type of GOST R 34.10-2001. When key is used the domain parameter object of key type CKK_GOSTR3410 must be specified with the same attribute CKA_OBJECT_ID |
| CKA_GOSTR3411_PARAMS ^{1,4,6,8} | Byte array | DER-encoding of the object identifier indicating the data object type of GOST R 34.11-94. When key is used the domain parameter object of key type CKK_GOSTR3411 must be specified with the same attribute CKA_OBJECT_ID |
| CKA_GOST28147_PARAMS ^{4,6,8} | Byte array | DER-encoding of the object identifier indicating the data object type of GOST 28147-89. |

| Attribute | Data Type | Meaning |
|-----------|-----------|--|
| | | When key is used the domain parameter object of key type CKK_GOST28147 must be specified with the same attribute CKA_OBJECT_ID. The attribute value may be omitted |

13689 - Refer to Table 11 for footnotes

13690 Note that when generating an GOST R 34.10-2001 private key, the GOST R 34.10-2001 domain parameters are *not* specified in the key's template. This is because GOST R 34.10-2001 private keys are only generated as part of an GOST R 34.10-2001 key pair, and the GOST R 34.10-2001 domain parameters for the pair are specified in the template for the GOST R 34.10-2001 public key.

13694 The following is a sample template for creating an GOST R 34.10-2001 private key object:

```

13695     CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
13696     CK_KEY_TYPE keyType = CKK_GOSTR3410;
13697     CK_UTF8CHAR label[] = "A GOST R34.10-2001 private key
13698         object";
13699     CK_BYTE subject[] = {...};
13700     CK_BYTE id[] = {123};
13701     CK_BYTE gostR3410params_oid[] =
13702         {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x23, 0x00};
13703     CK_BYTE gostR3411params_oid[] =
13704         {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1e, 0x00};
13705     CK_BYTE gost28147params_oid[] =
13706         {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1f, 0x00};
13707     CK_BYTE value[32] = {...};
13708     CK_BBOOL true = CK_TRUE;
13709     CK_ATTRIBUTE template[] = {
13710         {CKA_CLASS, &class, sizeof(class)},
13711         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
13712         {CKA_TOKEN, &true, sizeof(true)},
13713         {CKA_LABEL, label, sizeof(label)-1},
13714         {CKA_SUBJECT, subject, sizeof(subject)},
13715         {CKA_ID, id, sizeof(id)},
13716         {CKA_SENSITIVE, &true, sizeof(true)},
13717         {CKA_SIGN, &true, sizeof(true)},
13718         {CKA_GOSTR3410_PARAMS, gostR3410params_oid,
13719             sizeof(gostR3410params_oid)},
13720         {CKA_GOSTR3411_PARAMS, gostR3411params_oid,
13721             sizeof(gostR3411params_oid)},
13722         {CKA_GOST28147_PARAMS, gost28147params_oid,
13723             sizeof(gost28147params_oid)},
13724         {CKA_VALUE, value, sizeof(value)}}
13725     };
13726 
```

6.57.4 GOST R 34.10-2001 domain parameter objects

GOST R 34.10-2001 domain parameter objects (object class **CKO_DOMAIN_PARAMETERS**, key type **CKK_GOSTR3410**) hold GOST R 34.10-2001 domain parameters.

The following table defines the GOST R 34.10-2001 domain parameter object attributes, in addition to the common attributes defined for this object class:

Table 245, GOST R 34.10-2001 Domain Parameter Object Attributes

| Attribute | Data Type | Meaning |
|----------------------------|------------|--|
| CKA_VALUE ¹ | Byte array | DER-encoding of the domain parameters as it was introduced in [4] section 8.4 (type <i>GostR3410-2001-ParamSetParameters</i>) |
| CKA_OBJECT_ID ¹ | Byte array | DER-encoding of the object identifier indicating the domain parameters |

¹ Refer to Table 11 for footnotes

For any particular token, there is no guarantee that a token supports domain parameters loading up and/or fetching out. Furthermore, applications, that make direct use of domain parameters objects, should take in account that **CKA_VALUE** attribute may be inaccessible.

The following is a sample template for creating a GOST R 34.10-2001 domain parameter object:

```
13738     CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
13739     CK_KEY_TYPE keyType = CKK_GOSTR3410;
13740     CK_UTF8CHAR label[] = "A GOST R34.10-2001 cryptographic
13741         parameters object";
13742     CK_BYTE oid[] =
13743         {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x23, 0x00};
13744     CK_BYTE value[] = {
13745         0x30, 0x81, 0x90, 0x02, 0x01, 0x07, 0x02, 0x20, 0x5f, 0xbff, 0xf4, 0x98,
13746         0xaa, 0x93, 0x8c, 0xe7, 0x39, 0xb8, 0xe0, 0x22, 0xfb, 0xaf, 0xef, 0x40,
13747         0x56, 0x3f, 0x6e, 0x6a, 0x34, 0x72, 0xfc, 0x2a, 0x51, 0x4c, 0x0c, 0xe9,
13748         0xda, 0xe2, 0x3b, 0x7e, 0x02, 0x21, 0x00, 0x80, 0x00, 0x00, 0x00, 0x00,
13749         0x00, 0x00,
13750         0x00, 0x00,
13751         0x00, 0x04, 0x31, 0x02, 0x21, 0x00, 0x80, 0x00, 0x00, 0x00, 0x00, 0x00,
13752         0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 0x50, 0xfe,
13753         0x8a, 0x18, 0x92, 0x97, 0x61, 0x54, 0xc5, 0x9c, 0xfc, 0x19, 0x3a, 0xcc,
13754         0xf5, 0xb3, 0x02, 0x01, 0x02, 0x20, 0x08, 0xe2, 0xa8, 0xa0, 0xe6,
13755         0x51, 0x47, 0xd4, 0xbd, 0x63, 0x16, 0x03, 0x0e, 0x16, 0xd1, 0x9c, 0x85,
13756         0xc9, 0x7f, 0x0a, 0x9c, 0xa2, 0x67, 0x12, 0x2b, 0x96, 0xab, 0xbc, 0xea,
13757         0x7e, 0x8f, 0xc8
13758     };
13759     CK_BBOOL true = CK_TRUE;
13760     CK_ATTRIBUTE template[] = {
13761         {CKA_CLASS, &class, sizeof(class)},
13762         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
13763         {CKA_TOKEN, &true, sizeof(true)},
13764         {CKA_LABEL, label, sizeof(label)-1},
13765         {CKA_OBJECT_ID, oid, sizeof(oid)},
13766         {CKA_VALUE, value, sizeof(value)}
13767     };

```

13768

13769 6.57.5 GOST R 34.10-2001 mechanism parameters

13770 ♦ CK_GOSTR3410_KEY_WRAP_PARAMS

13771 **CK_GOSTR3410_KEY_WRAP_PARAMS** is a structure that provides the parameters to the
13772 **CKM_GOSTR3410_KEY_WRAP** mechanism. It is defined as follows:

```
13773     typedef struct CK_GOSTR3410_KEY_WRAP_PARAMS {  
13774         CK_BYTE_PTR          pWrapOID;  
13775         CK ULONG             ulWrapOIDLen;  
13776         CK_BYTE_PTR          pUKM;  
13777         CK ULONG             ulUKMLen;  
13778         CK_OBJECT_HANDLE    hKey;  
13779     } CK_GOSTR3410_KEY_WRAP_PARAMS;
```

13780

13781 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 |

13782 **CK_GOSTR3410_KEY_WRAP_PARAMS_PTR** is a pointer to a
13783 **CK_GOSTR3410_KEY_WRAP_PARAMS**.

13784 ♦ CK_GOSTR3410_DERIVE_PARAMS

13785 **CK_GOSTR3410_DERIVE_PARAMS** is a structure that provides the parameters to the
13786 **CKM_GOSTR3410_DERIVE** mechanism. It is defined as follows:

```
13787     typedef struct CK_GOSTR3410_DERIVE_PARAMS {  
13788         CK EC KDF_TYPE kdf;  
13789         CK BYTE_PTR      pPublicData;  
13790         CK ULONG          ulPublicDataLen;
```

```
13791     CK_BYTE_PTR      pUKM;
13792     CK ULONG          ulUKMLen;
13793 } CK_GOSTR3410_DERIVE_PARAMS;
```

13794

13795 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, 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) |

13796

13797 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
13798 them is 32 bytes long and represented in little endian order.

13799 CK_GOSTR3410_DERIVE_PARAMS_PTR is a pointer to a CK_GOSTR3410_DERIVE_PARAMS.

13800 6.57.6 GOST R 34.10-2001 key pair generation

13801 The GOST R 34.10-2001 key pair generation mechanism, denoted
13802 **CKM_GOSTR3410_KEY_PAIR_GEN**, is a key pair generation mechanism for GOST R 34.10-2001.

13803 This mechanism does not have a parameter.

13804 The mechanism generates GOST R 34.10-2001 public/private key pairs with particular
13805 GOST R 34.10-2001 domain parameters, as specified in the **CKA_GOSTR3410_PARAMS**,
13806 **CKA_GOSTR3411_PARAMS**, and **CKA_GOST28147_PARAMS** attributes of the template for the public
13807 key. Note that **CKA_GOST28147_PARAMS** attribute may not be present in the template.

13808 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
13809 public key and the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_VALUE**, and **CKA_GOSTR3410_PARAMS**,
13810 **CKA_GOSTR3411_PARAMS**, **CKA_GOST28147_PARAMS** attributes to the new private key.

13811 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
13812 are not used.

13813 6.57.7 GOST R 34.10-2001 without hashing

13814 The GOST R 34.10-2001 without hashing mechanism, denoted **CKM_GOSTR3410**, is a mechanism for
13815 single-part signatures and verification for GOST R 34.10-2001. (This mechanism corresponds only to the
13816 part of GOST R 34.10-2001 that processes the 32-bytes hash value; it does not compute the hash value.)

13817 This mechanism does not have a parameter.

13818 For the purposes of these mechanisms, a GOST R 34.10-2001 signature is an octet string of 64 bytes
13819 long. The signature octets correspond to the concatenation of the GOST R 34.10-2001 values s and r',
13820 both represented as a 32 bytes octet string in big endian order with the most significant byte first [RFC
13821 4490] section 3.2, and [RFC 4491] section 2.2.2.

13822 The input for the mechanism is an octet string of 32 bytes long with digest has computed by means of
13823 GOST R 34.11-94 hash algorithm in the context of signed or should be signed message.

13824 *Table 246, GOST R 34.10-2001 without hashing: Key and Data Length*

| Function | Key type | Input length | Output length |
|-----------------------|---------------|--------------|---------------|
| C_Sign ¹ | CKK_GOSTR3410 | 32 bytes | 64 bytes |
| C_Verify ¹ | CKK_GOSTR3410 | 32 bytes | 64 bytes |

13825 ¹ Single-part operations only.

13826 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
13827 are not used.

13828 **6.57.8 GOST R 34.10-2001 with GOST R 34.11-94**

13829 The GOST R 34.10-2001 with GOST R 34.11-94, denoted **CKM_GOSTR3410_WITH_GOSTR3411**, is a
13830 mechanism for signatures and verification for GOST R 34.10-2001. This mechanism computes the entire
13831 GOST R 34.10-2001 specification, including the hashing with GOST R 34.11-94 hash algorithm.

13832 As a parameter this mechanism utilizes a DER-encoding of the object identifier indicating
13833 GOST R 34.11-94 data object type. A mechanism parameter may be missed then parameters are
13834 specified in object identifier of attribute **CKA_GOSTR3411_PARAMS** must be used.

13835 For the purposes of these mechanisms, a GOST R 34.10-2001 signature is an octet string of 64 bytes
13836 long. The signature octets correspond to the concatenation of the GOST R 34.10-2001 values *s* and *r'*,
13837 both represented as a 32 bytes octet string in big endian order with the most significant byte first [RFC
13838 4490] section 3.2, and [RFC 4491] section 2.2.2.

13839 The input for the mechanism is signed or should be signed message of any length. Single- and multiple-
13840 part signature operations are available.

13841 *Table 247, GOST R 34.10-2001 with GOST R 34.11-94: Key and Data Length*

| Function | Key type | Input length | Output length |
|----------|---------------|--------------|---------------|
| C_Sign | CKK_GOSTR3410 | Any | 64 bytes |
| C_Verify | CKK_GOSTR3410 | Any | 64 bytes |

13842 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
13843 are not used.

13844 **6.57.9 GOST 28147-89 keys wrapping/unwrapping with GOST R 34.10-2001**

13845 GOST R 34.10-2001 keys as a KEK (key encryption keys) for encryption GOST 28147 keys, denoted by
13846 **CKM_GOSTR3410_KEY_WRAP**, is a mechanism for key wrapping; and key unwrapping, based on
13847 GOST R 34.10-2001. Its purpose is to encrypt and decrypt keys have been generated by key generation
13848 mechanism for GOST 28147-89. An encryption algorithm from [RFC 4490] (section 5.2) must be used.
13849 Encrypted key is a DER-encoded structure of ASN.1 *GostR3410-KeyTransport* type [RFC 4490] section
13850 4.2.

13851 It has a parameter, a **CK_GOSTR3410_KEY_WRAP_PARAMS** structure defined in section 6.57.5.

13852 For unwrapping (**C_UnwrapKey**), the mechanism decrypts the wrapped key, and contributes the result as
13853 the **CKA_VALUE** attribute of the new key.

13854 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
13855 are not used.

13856 **6.57.10 Common key derivation with assistance of GOST R 34.10-2001 keys**

13857 Common key derivation, denoted **CKM_GOSTR3410_DERIVE**, is a mechanism for key derivation with
13858 assistance of GOST R 34.10-2001 private and public keys. The key of the mechanism must be of object

13859 class **CKO_DOMAIN_PARAMETERS** and key type **CKK_GOSTR3410**. An algorithm for key derivation
13860 from [RFC 4357] (section 5.2) must be used.

13861 The mechanism contributes the result as the **CKA_VALUE** attribute of the new private key. All other
13862 attributes must be specified in a template for creating private key object.

13863 **6.58 ChaCha20**

13864 ChaCha20 is a secret-key stream cipher described in [**CHACHA**].

13865 *Table 248, ChaCha20 Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|----------------------|-------------------|---------------|----------------------|--------|-------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/Key Pair | Wrap & Unwrap | Derive |
| CKM_CHACHA20_KEY_GEN | | | | | ✓ | | |
| CKM_CHACHA20 | ✓ | | | | | ✓ | |

13866

13867 **6.58.1 Definitions**

13868 This section defines the key type “CKK_CHACHA20” for type CK_KEY_TYPE as used in the
13869 CKA_KEY_TYPE attribute of key objects.

13870 Mechanisms:

13871 CKM_CHACHA20_KEY_GEN

13872 CKM_CHACHA20

13873 **6.58.2 ChaCha20 secret key objects**

13874 ChaCha20 secret key objects (object class CKO_SECRET_KEY, key type CKK_CHACHA20) hold
13875 ChaCha20 keys. The following table defines the ChaCha20 secret key object attributes, in addition to the
13876 common attributes defined for this object class:

13877 *Table 249, ChaCha20 Secret Key Object*

| Attribute | Data type | Meaning |
|------------------------------|------------|--|
| CKA_VALUE ^{1,4,6,7} | Byte array | Key length is fixed at 256 bits. Bit length restricted to a byte array. |
| CKA_VALUE_LEN ^{2,3} | CK ULONG | Length in bytes of key value |

13878 The following is a sample template for creating a ChaCha20 secret key object:

```
13879 CK_OBJECT_CLASS class = CKO_SECRET_KEY;
13880 CK_KEY_TYPE keyType = CKK_CHACHA20;
13881 CK_UTF8CHAR label[] = "A ChaCha20 secret key object";
13882 CK_BYTE value[32] = {...};
13883 CK_BBOOL true = CK_TRUE;
13884 CK_ATTRIBUTE template[] = {
13885     {CKA_CLASS, &class, sizeof(class)},
13886     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
13887     {CKA_TOKEN, &true, sizeof(true)},
13888     {CKA_LABEL, label, sizeof(label)-1},
13889     {CKA_ENCRYPT, &true, sizeof(true)},
```

```
13890     { CKA_VALUE, value, sizeof(value) }  
13891 } ;  
13892 CKA_CHECK_VALUE: The value of this attribute is derived from the key object by taking the first  
13893 three bytes of the SHA-1 hash of the ChaCha20 secret key object's CKA_VALUE attribute.
```

13894 6.58.3 ChaCha20 mechanism parameters

◆ CK_CHACHA20_PARAMS; CK_CHACHA20_PARAMS_PTR

13896 CK_CHACHA20_PARAMS provides the parameters to the CKM_CHACHA20 mechanism. It is defined
13897 as follows:

```
13898     typedef struct CK_CHACHA20_PARAMS {  
13899         CK_BYTE_PTR    pBlockCounter;  
13900         CK ULONG      blockCounterBits;  
13901         CK_BYTE_PTR    pNonce;  
13902         CK ULONG      ulNonceBits;  
13903     } CK_CHACHA20_PARAMS;
```

13904 The fields of the structure have the following meanings:

| | | |
|-------|---------------------------|--|
| 13905 | <i>pBlockCounter</i> | <i>pointer to block counter</i> |
| 13906 | <i>ulblockCounterBits</i> | <i>length of block counter in bits (can be either 32 or 64)</i> |
| 13907 | <i>pNonce</i> | <i>nonce (This should be never re-used with the same key.)</i> |
| 13908 | <i>ulNonceBits</i> | <i>length of nonce in bits (is 64 for original, 96 for IETF and 192 for 13909 xchacha20 variant)</i> |

13910 The block counter is used to address 512 bit blocks in the stream. In certain settings (e.g. disk encryption)
13911 it is necessary to address these blocks in random order, thus this counter is exposed here.

13912 CK_CHACHA20_PARAMS_PTR is a pointer to CK_CHACHA20_PARAMS.

13913 6.58.4 ChaCha20 key generation

13914 The ChaCha20 key generation mechanism, denoted CKM_CHACHA20_KEY_GEN, is a key generation
13915 mechanism for ChaCha20.

13916 It does not have a parameter.

13917 The mechanism generates ChaCha20 keys of 256 bits.

13918 The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, and CKA_VALUE attributes to the new
13919 key. Other attributes supported by the key type (specifically, the flags indicating which functions the key
13920 supports) may be specified in the template for the key, or else are assigned default initial values.

13921 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK_MECHANISM_INFO structure
13922 specify the supported range of key sizes in bytes. As a practical matter, the key size for ChaCha20 is
13923 fixed at 256 bits.

13924

13925 6.58.5 ChaCha20 mechanism

13926 ChaCha20, denoted CKM_CHACHA20, is a mechanism for single and multiple-part encryption and
13927 decryption based on the ChaCha20 stream cipher. It comes in 3 variants, which only differ in the size and
13928 handling of their nonces, affecting the safety of using random nonces and the maximum size that can be
13929 encrypted safely.

13930 Chacha20 has a parameter, **CK_CHACHA20_PARAMS**, which indicates the nonce and initial block
 13931 counter value.
 13932 Constraints on key types and the length of input and output data are summarized in the following table:
 13933 *Table 250, ChaCha20: Key and Data Length*

| Function | Key type | Input length | Output length | Comments |
|-----------------|-----------------|---|----------------------|-----------------|
| C_Encrypt | ChaCha20 | Any / only up to 256 GB in case of IETF variant | Same as input length | No final part |
| C_Decrypt | ChaCha20 | Any / only up to 256 GB in case of IETF variant | Same as input length | No final part |

13934 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 13935 specify the supported range of ChaCha20 key sizes, in bits.
 13936 *Table 251, ChaCha20:Nonce and block counter lengths*

| Variant | Nonce | Block counter | Maximum message | Nonce generation |
|----------------|--------------|----------------------|------------------------|---|
| original | 64 bit | 64 bit | Virtually unlimited | 1 st msg: nonce ₀ =random n th msg: nonce _{n-1} ++ |
| IETF | 96 bit | 32 bit | Max ~256 GB | 1 st msg: nonce ₀ =random n th msg: nonce _{n-1} ++ |
| XChaCha20 | 192 bit | 64 bit | Virtually unlimited | Each nonce can be randomly generated. |

13937 Nonces must not ever be reused with the same key. However due to the birthday paradox the first two
 13938 variants cannot guarantee that randomly generated nonces are never repeating. Thus the recommended
 13939 way to handle this is to generate the first nonce randomly, then increase this for follow-up messages.
 13940 Only the last (XChaCha20) has large enough nonces so that it is virtually impossible to trigger with
 13941 randomly generated nonces the birthday paradox.

13942 **6.59 Salsa20**

13943 Salsa20 is a secret-key stream cipher described in [SALSA].

13944 *Table 252, Salsa20 Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|---------------------|-------------------|---------------|----------------------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_SALSA20_KEY_GEN | | | | | ✓ | | |
| CKM_SALSA20 | ✓ | | | | | ✓ | |

13945

6.59.1 Definitions

13947 This section defines the key type “CKK_SALSA20” and “CKK_SALSA20” for type CK_KEY_TYPE as
 13948 used in the CKA_KEY_TYPE attribute of key objects.

13949 Mechanisms:

13950 CKM_SALSA20_KEY_GEN
 13951 CKM_SALSA20

6.59.2 Salsa20 secret key objects

13953 Salsa20 secret key objects (object class CKO_SECRET_KEY, key type CKK_SALSA20) hold Salsa20
 13954 keys. The following table defines the Salsa20 secret key object attributes, in addition to the common
 13955 attributes defined for this object class:

13956 *Table 253, ChaCha20 Secret Key Object*

| Attribute | Data type | Meaning |
|------------------------------|------------|--|
| CKA_VALUE ^{1,4,6,7} | Byte array | Key length is fixed at 256 bits. Bit length restricted to a byte array. |
| CKA_VALUE_LEN ^{2,3} | CK ULONG | Length in bytes of key value |

13957 The following is a sample template for creating a Salsa20 secret key object:

```
13958     CK_OBJECT_CLASS class = CKO_SECRET_KEY;
13959     CK_KEY_TYPE keyType = CKK_SALSA20;
13960     CK_UTF8CHAR label[] = "A Salsa20 secret key object";
13961     CK_BYTE value[32] = {...};
13962     CK_BBOOL true = CK_TRUE;
13963     CK_ATTRIBUTE template[] = {
13964         {CKA_CLASS, &class, sizeof(class)},
13965         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
13966         {CKA_TOKEN, &true, sizeof(true)},
13967         {CKA_LABEL, label, sizeof(label)-1},
13968         {CKA_ENCRYPT, &true, sizeof(true)},
13969         {CKA_VALUE, value, sizeof(value)}}
13970     };
```

13971 CKA_CHECK_VALUE: The value of this attribute is derived from the key object by taking the first
 13972 three bytes of the SHA-1 hash of the ChaCha20 secret key object's CKA_VALUE attribute.

13973 6.59.3 Salsa20 mechanism parameters

13974 ◆ CK_SALSA20_PARAMS; CK_SALSA20_PARAMS_PTR

13975 CK_SALSA20_PARAMS provides the parameters to the CKM_SALSA20 mechanism. It is defined as
13976 follows:

```
13977     typedef struct CK_SALSA20_PARAMS {  
13978         CK_BYTE_PTR    pBlockCounter;  
13979         CK_BYTE_PTR    pNonce;  
13980         CK ULONG       ulNonceBits;  
13981     } CK_SALSA20_PARAMS;
```

13982 The fields of the structure have the following meanings:

13984 pBlockCounter pointer to block counter (64 bits)

13985 pNonce nonce

13986 ulNonceBits size of the nonce in bits (64 for classic and 192 for XSalsa20)

13987 The block counter is used to address 512 bit blocks in the stream. In certain settings (e.g. disk encryption)
13988 it is necessary to address these blocks in random order, thus this counter is exposed here.

13989 CK_SALSA20_PARAMS_PTR is a pointer to CK_SALSA20_PARAMS.

13990 6.59.4 Salsa20 key generation

13991 The Salsa20 key generation mechanism, denoted CKM_SALSA20_KEY_GEN, is a key generation
13992 mechanism for Salsa20.

13993 It does not have a parameter.

13994 The mechanism generates Salsa20 keys of 256 bits.

13995 The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, and CKA_VALUE attributes to the new
13996 key. Other attributes supported by the key type (specifically, the flags indicating which functions the key
13997 supports) may be specified in the template for the key, or else are assigned default initial values.

13998 For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure
13999 specify the supported range of key sizes in bytes. As a practical matter, the key size for Salsa20 is fixed
14000 at 256 bits.

14001 6.59.5 Salsa20 mechanism

14002 Salsa20, denoted CKM_SALSA20, is a mechanism for single and multiple-part encryption and decryption
14003 based on the Salsa20 stream cipher. Salsa20 comes in two variants which only differ in the size and
14004 handling of their nonces, affecting the safety of using random nonces.

14005 Salsa20 has a parameter, CK_SALSA20_PARAMS, which indicates the nonce and initial block counter
14006 value.

14007 Constraints on key types and the length of input and output data are summarized in the following table:

14008 *Table 254, Salsa20: Key and Data Length*

| Function | Key type | Input length | Output length | Comments |
|-----------|----------|--------------|----------------------|---------------|
| C_Encrypt | Salsa20 | Any | Same as input length | No final part |
| C_Decrypt | Salsa20 | Any | Same as input length | No final part |

14009 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
14010 specify the supported range of ChaCha20 key sizes, in bits.

14011 *Table 255, Salsa20: Nonce sizes*

| Variant | Nonce | Maximum message | Nonce generation |
|----------|---------|---------------------|---|
| original | 64 bit | Virtually unlimited | 1 st msg: nonce ₀ =random n th msg: nonce _{n-1} ++ |
| XSalsa20 | 192 bit | Virtually unlimited | Each nonce can be randomly generated. |

14012 Nonces must not ever be reused with the same key. However due to the birthday paradox the original
14013 variant cannot guarantee that randomly generated nonces are never repeating. Thus the recommended
14014 way to handle this is to generate the first nonce randomly, then increase this for follow-up messages.
14015 Only the XSalsa20 has large enough nonces so that it is virtually impossible to trigger with randomly
14016 generated nonces the birthday paradox.

14017 6.60 Poly1305

14018 Poly1305 is a message authentication code designed by D.J Bernstein [**POLY1305**]. Poly1305 takes a
14019 256 bit key and a message and produces a 128 bit tag that is used to verify the message.

14020 *Table 256, Poly1305 Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|----------------------|-------------------|---------------|----------------------|--------|-------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/Key Pair | Wrap & Unwrap | Derive |
| CKM_POLY1305_KEY_GEN | | | | | ✓ | | |
| CKM_POLY1305 | | ✓ | | | | | |

14021 6.60.1 Definitions

14022 This section defines the key type “CKK_POLY1305” for type CK_KEY_TYPE as used in the
14023 CKA_KEY_TYPE attribute of key objects.

14024 Mechanisms:

14025 CKM_POLY1305_KEY_GEN

14026 CKM_POLY1305

14027 6.60.2 Poly1305 secret key objects

14028 Poly1305 secret key objects (object class CKO_SECRET_KEY, key type CKK_POLY1305) hold
14029 Poly1305 keys. The following table defines the Poly1305 secret key object attributes, in addition to the
14030 common attributes defined for this object class:

14031 *Table 257, Poly1305 Secret Key Object*

| Attribute | Data type | Meaning |
|------------------------------|------------------|--|
| CKA_VALUE ^{1,4,6,7} | Byte array | Key length is fixed at 256 bits. Bit length restricted to a byte array. |
| CKA_VALUE_LEN ^{2,3} | CK ULONG | Length in bytes of key value |

14032 The following is a sample template for creating a Poly1305 secret key object:

```

14033     CK_OBJECT_CLASS class = CKO_SECRET_KEY;
14034     CK_KEY_TYPE keyType = CKK_POLY1305;
14035     CK_UTF8CHAR label[] = "A Poly1305 secret key object";
14036     CK_BYTE value[32] = {...};
14037     CK_BBOOL true = CK_TRUE;
14038     CK_ATTRIBUTE template[] = {
14039         {CKA_CLASS, &class, sizeof(class)},
14040         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
14041         {CKA_TOKEN, &true, sizeof(true)},
14042         {CKA_LABEL, label, sizeof(label)-1},
14043         {CKA_SIGN, &true, sizeof(true)},
14044         {CKA_VALUE, value, sizeof(value)}}
14045     };
```

14046

14047 **6.60.3 Poly1305 mechanism**

14048 Poly1305, denoted **CKM_POLY1305**, is a mechanism for producing an output tag based on a 256 bit key
14049 and arbitrary length input.

14050 It has no parameters.

14051 Signatures (MACs) produced by this mechanism will be fixed at 128 bits in size.

14052 *Table 258, Poly1305: Key and Data Length*

| Function | Key type | Data length | Signature Length |
|-----------------|-----------------|--------------------|-------------------------|
| C_Sign | Poly1305 | Any | 128 bits |
| C_Verify | Poly1305 | Any | 128 bits |

14053 **6.61 Chacha20/Poly1305 and Salsa20/Poly1305 Authenticated 14054 Encryption / Decryption**

14055 The stream ciphers Salsa20 and ChaCha20 are normally used in conjunction with the Poly1305
14056 authenticator, in such a construction they also provide Authenticated Encryption with Associated Data
14057 (AEAD). This section defines the combined mechanisms and their usage in an AEAD setting.

14058 *Table 259, Poly1305 Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|-----------------------|-------------------|---------------|----------------------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_CHACHA20_POLY1305 | ✓ | | | | | | |
| CKM_SALSA20_POLY1305 | ✓ | | | | | | |

14059 **6.61.1 Definitions**

14060 Mechanisms:

14061 CKM_CHACHA20_POLY1305

14062 CKM_SALSA20_POLY1305

14063 **6.61.2 Usage**

14064 Generic ChaCha20, Salsa20, Poly1305 modes are described in [CHACHA], [SALSA] and [POLY1305].
 14065 To set up for ChaCha20/Poly1305 or Salsa20/Poly1305 use the following process. ChaCha20/Poly1305
 14066 and Salsa20/Poly1305 both use CK_SALSA20_CHACHA20_POLY1305_PARAMS for Encrypt, Decrypt
 14067 and CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS for MessageEncrypt, and MessageDecrypt.

14068 Encrypt:

- Set the Nonce length *uNonceLen* in the parameter block. (this affects which variant of Chacha20 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)
- Set the Nonce data *pNonce* in the parameter block.
- Set the AAD data *pAAD* and size *uAADLen* in the parameter block. *pAAD* may be NULL if *uAADLen* is 0.
- Call C_EncryptInit() for **CKM_CHACHA20_POLY1305** or **CKM_SALSA20_POLY1305** mechanism with parameters and key *K*.
- Call C_Encrypt(), or C_EncryptUpdate()^{*10} C_EncryptFinal(), for the plaintext obtaining ciphertext and authentication tag output.

14078 Decrypt:

- Set the Nonce length *uNonceLen* in the parameter block. (this affects which variant of Chacha20 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)
- Set the Nonce data *pNonce* in the parameter block.
- Set the AAD data *pAAD* and size *uAADLen* in the parameter block. *pAAD* may be NULL if *uAADLen* is 0.
- Call C_DecryptInit() for **CKM_CHACHA20_POLY1305** or **CKM_SALSA20_POLY1305** mechanism with parameters and key *K*.
- Call C_Decrypt(), or C_DecryptUpdate()^{*1} C_DecryptFinal(), for the ciphertext, including the appended tag, obtaining plaintext output. Note: since **CKM_CHACHA20_POLY1305** and **CKM_SALSA20_POLY1305** are AEAD ciphers, no data should be returned until C_Decrypt() or C_DecryptFinal().

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

14090 MessageEncrypt::

- 14091 • Set the Nonce length *uNonceLen* in the parameter block. (this affects which variant of Chacha20 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)
- 14093 • Set the Nonce data *pNonce* in the parameter block.
- 14094 • Set *pTag* to hold the tag data returned from *C_EncryptMessage()* or the final *C_EncryptMessageNext()*.
- 14096 • Call *C_MessageEncryptInit()* for **CKM_CHACHA20_POLY1305** or **CKM_SALSA20_POLY1305** mechanism with key *K*.
- 14098 • Call *C_EncryptMessage()*, or *C_EncryptMessageBegin* followed by *C_EncryptMessageNext()**¹¹. The mechanism parameter is passed to all three of these functions.
- 14100 • Call *C_MessageEncryptFinal()* to close the message decryption.

14101 MessageDecrypt:

- 14102 • Set the Nonce length *uNonceLen* in the parameter block. (this affects which variant of Chacha20 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)
- 14104 • Set the Nonce data *pNonce* in the parameter block.
- 14105 • Set the tag data *pTag* in the parameter block before *C_DecryptMessage* or the final *C_DecryptMessageNext()*
- 14107 • Call *C_MessageDecryptInit()* for **CKM_CHACHA20_POLY1305** or **CKM_SALSA20_POLY1305** mechanism with key *K*.
- 14109 • Call *C_DecryptMessage()*, or *C_DecryptMessageBegin* followed by *C_DecryptMessageNext()**¹². The mechanism parameter is passed to all three of these functions.
- 14111 • Call *C_MessageDecryptFinal()* to close the message decryption

14112

14113 *uNonceLen* is the length of the nonce in bits.

14114 In Encrypt and Decrypt the tag is appended to the cipher text. In MessageEncrypt the tag is returned in the *pTag* field of *CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS*. In MessageDecrypt the tag is provided by the *pTag* field of *CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS*. The application must provide 16 bytes of space for the tag.

14115 The key type for *K* must be compatible with **CKM_CHACHA20** or **CKM_SALSA20** respectively and the *C_EncryptInit/C_DecryptInit* calls shall behave, with respect to *K*, as if they were called directly with **CKM_CHACHA20** or **CKM_SALSA20**, *K* and NULL parameters.

14116 Unlike the atomic Salsa20/ChaCha20 mechanism the AEAD mechanism based on them does not expose the block counter, as the AEAD construction is based on a message metaphor in which random access is not needed.

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

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

14124 **6.61.3 ChaCha20/Poly1305 and Salsa20/Poly1305 Mechanism parameters**

- 14125 ◆ **CK_SALSA20_CHACHA20_POLY1305_PARAMS;**
14126 **CK_SALSA20_CHACHA20_POLY1305_PARAMS_PTR**

14127 **CK_SALSA20_CHACHA20_POLY1305_PARAMS** is a structure that provides the parameters to the
14128 **CKM_CHACHA20_POLY1305** and **CKM_SALSA20_POLY1305** mechanisms. It is defined as follows:

```
14129    typedef struct CK_SALSA20_CHACHA20_POLY1305_PARAMS {  
14130        CK_BYTE_PTR pNonce;  
14131        CK ULONG ulNonceLen;  
14132        CK_BYTE_PTR pAAD;  
14133        CK ULONG ulAADLen;  
14134 } CK_SALSA20_CHACHA20_POLY1305_PARAMS;
```

14135 The fields of the structure have the following meanings:

14136 *pNonce* *nonce (This should be never re-used with the same key.)*

14137 *ulNonceLen* *length of nonce in bits (is 64 for original, 96 for IETF (only for
14138 chacha20) and 192 for xchacha20/xsalsa20 variant)*

14139 *pAAD* *pointer to additional authentication data. This data is authenticated
14140 but not encrypted.*

14141 *ulAADLen* *length of pAAD in bytes.*

14142 **CK_SALSA20_CHACHA20_POLY1305_PARAMS_PTR** is a pointer to a
14143 **CK_SALSA20_CHACHA20_POLY1305_PARAMS**.

- 14144 ◆ **CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS;**
14145 **CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS_PTR**

14146 **CK_CHACHA20POLY1305_PARAMS** is a structure that provides the parameters to the **CKM_**
14147 **CHACHA20_POLY1305** mechanism. It is defined as follows:

```
14148    typedef struct CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS {  
14149        CK_BYTE_PTR pNonce;  
14150        CK ULONG ulNonceLen;  
14151        CK_BYTE_PTR pTag;  
14152 } CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS;
```

14153 The fields of the structure have the following meanings:

14154 *pNonce* *pointer to nonce*

14155 *ulNonceLen* *length of nonce in bits. The length of the influences which variant of
14156 the ChaCha20 will be used (64 original, 96 IETF(only for
14157 ChaCha20), 192 XChaCha20/XSalsa20)*

14158 *pTag* *location of the authentication tag which is returned on
14159 MessageEncrypt, and provided on MessageDecrypt.*

14160 **CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS_PTR** is a pointer to a
14161 **CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS**.

6.62 HKDF Mechanisms

Details for HKDF key derivation mechanisms can be found in [RFC 5869].

14164

14165 *Table 260, HKDF Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|------------------|-------------------|---------------|----------------------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_HKDF_DERIVE | | | | | | | ✓ |
| CKM_HKDF_DATA | | | | | | | ✓ |
| CKM_HKDF_KEY_GEN | | | | | ✓ | | |

6.62.1 Definitions

Mechanisms:

CKM_HKDF_DERIVE

CKM_HKDF_DATA

CKM_HKDF_KEY_GEN

14171

Key Types:

CKK_HKDF

6.62.2 HKDF mechanism parameters

◆ CK_HKDF_PARAMS; CK_HKDF_PARAMS_PTR

CK_HKDF_PARAMS is a structure that provides the parameters to the CKM_HKDF_DERIVE and CKM_HKDF_DATA mechanisms. It is defined as follows:

```
14178 typedef struct CK_HKDF_PARAMS {  
14179     CK_BBOOL bExtract;  
14180     CK_BBOOL bExpand;  
14181     CK_MECHANISM_TYPE prfHashMechanism;  
14182     CK ULONG ulSaltType;  
14183     CK_BYTE_PTR pSalt;  
14184     CK ULONG ulSaltLen;  
14185     CK_OBJECT_HANDLE hSaltKey;  
14186     CK_BYTE_PTR pInfo;  
14187     CK ULONG ulInfoLen;  
14188 } CK_HKDF_PARAMS;
```

14189

The fields of the structure have the following meanings:

14191 bExtract execute the extract portion of HKDF.

14192 bExpand execute the expand portion of HKDF.

14193 prfHashMechanism base hash used for the HMAC in the underlying HKDF operation.

14194 ulSaltType specifies how the salt for the extract portion of the KDF is supplied.

| | | |
|-------|------------|---|
| 14195 | | CKF_HKDF_SALT_NULL no salt is supplied. |
| 14196 | | CKF_HKDF_SALT_DATA salt is supplied as a data in pSalt with |
| 14197 | | length ulSaltLen. |
| 14198 | | CKF_HKDF_SALT_KEY salt is supplied as a key in hSaltKey. |
| 14199 | pSalt | pointer to the salt. |
| 14200 | ulSaltLen | length of the salt pointed to in pSalt. |
| 14201 | hSaltKey | object handle to the salt key. |
| 14202 | pInfo | info string for the expand stage. |
| 14203 | ullInfoLen | length of the info string for the expand stage. |
| 14204 | | |

14205 **CK_HKDF_PARAMS_PTR** is a pointer to a **CK_HKDF_PARAMS**.

14206 6.62.3 HKDF derive

14207 HKDF derivation implements the HKDF as specified in [RFC 5869]. The two booleans bExtract and
14208 bExpand control whether the extract section of the HKDF or the expand section of the HKDF is in use.
14209 It has a parameter, a **CK_HKDF_PARAMS** structure, which allows for the passing of the salt and or the
14210 expansion info. The structure contains the bools *bExtract* and *bExpand* which control whether the extract
14211 or expand portions of the HKDF is to be used. This structure is defined in Section 6.62.2.
14212 The input key must be of type **CKK_HKDF** or **CKK_GENERIC_SECRET** and the length must be the size
14213 of the underlying hash function specified in *prfHashMechanism*. The exception is a data object which has
14214 the same size as the underlying hash function, and which may be supplied as an input key. In this case
14215 *bExtract* should be true and non-null salt should be supplied.
14216 Either *bExtract* or *bExpand* must be set to true. If they are both set to true, input key is first extracted then
14217 expanded. The salt is used in the extraction stage. If *bExtract* is set to true and no salt is given, a 'zero'
14218 salt (salt whose length is the same as the underlying hash and values all set to zero) is used as specified
14219 by the RFC. If *bExpand* is set to true, **CKA_VALUE_LEN** should be set to the desired key length. If it is
14220 false **CKA_VALUE_LEN** may be set to the length of the hash, but that is not necessary as the mechanism
14221 will supply this value. The salt should be ignored if *bExtract* is false. The *pInfo* should be ignored if
14222 *bExpand* is set to false.
14223 The mechanism also contributes the **CKA_CLASS**, and **CKA_VALUE** attributes to the new key. Other
14224 attributes may be specified in the template, or else are assigned default values.
14225 The template sent along with this mechanism during a **C_DeriveKey** call may indicate that the object
14226 class is **CKO_SECRET_KEY**. However, since these facts are all implicit in the mechanism, there is no
14227 need to specify any of them.
14228 This mechanism has the following rules about key sensitivity and extractability:
14229

- The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both
14230 be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some
14231 default value.
- If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key
14233 will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the
14234 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
14235 **CKA_SENSITIVE** attribute.
- Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
14237 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
14238 CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
14239 value from its **CKA_EXTRACTABLE** attribute.

14240 6.62.4 HKDF Data

14241 HKDF Data derive mechanism, denoted **CKM_HKDF_DATA**, is identical to HKDF Derive except the
14242 output is a **CKO_DATA** object whose value is the result to the derive operation. Some tokens may restrict
14243 what data may be successfully derived based on the *pInfo* portion of the **CK_HKDF_PARAMS**. Tokens
14244 may reject requests based on the *pInfo* values. Allowed *pInfo* values are specified in the profile document
14245 and applications could then query the appropriate profile before depending on the mechanism.

14246 6.62.5 HKDF Key gen

14247 HKDF key gen, denoted **CKM_HKDF_KEY_GEN** generates a new random HKDF key.
14248 **CKA_VALUE_LEN** must be set in the template.

14249 6.63 NULL Mechanism

14250 **CKM_NULL** is a mechanism used to implement the trivial pass-through function.

14251

14252 *Table 261, CKM_NULL Mechanisms vs. Functions*

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

¹SR = SignRecover, VR = VerifyRecover

14253

14254 6.63.1 Definitions

14255 Mechanisms:

14256 **CKM_NULL**

14257 6.63.2 CKM_NULL mechanism parameters

14258 CKM_NULL does not have a parameter.

14259

14260 When used for encrypting / decrypting data, the input data is copied unchanged to the output data.

14261 When used for signing, the input data is copied to the signature. When used for signature verification, it
14262 compares the input data and the signature, and returns **CKR_OK** (indicating that both are identical) or
14263 **CKR_SIGNATURE_INVALID**.

14264 When used for digesting data, the input data is copied to the message digest.

14265 When used for wrapping a private or secret key object, the wrapped key will be identical to the key to be
14266 wrapped. When used for unwrapping, a new object with the same value as the wrapped key will be
14267 created.

14268 When used for deriving a key, the derived key has the same value as the base key.

14269

14270 6.64 IKE Mechanisms

14271

14272 *Table 262, IKE Mechanisms vs. Functions*

| Mechanism | Functions | | | | | | |
|--------------------------|-------------------|---------------|----------------------|--------|--------------------|---------------|--------|
| | Encrypt & Decrypt | Sign & Verify | SR & VR ¹ | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_IKE2_PRF_PLUS_DERIVE | | | | | | | ✓ |
| CKM_IKE_PRF_DERIVE | | | | | | | ✓ |
| CKM_IKE1_PRF_DERIVE | | | | | | | ✓ |
| CKM_IKE1_EXTENDED_DERIVE | | | | | | | ✓ |

14273

14274 6.64.1 Definitions

14275 Mechanisms:

14276 CKM_IKE2_PRF_PLUS_DERIVE
 14277 CKM_IKE_PRF_DERIVE
 14278 CKM_IKE1_PRF_DERIVE
 14279 CKM_IKE1_EXTENDED_DERIVE

14280

14281 6.64.2 IKE mechanism parameters

- ◆ **CK_IKE2_PRF_PLUS_DERIVE_PARAMS;**
- CK_IKE2_PRF_PLUS_DERIVE_PARAMS_PTR**

14284 CK_IKE2_PRF_PLUS_DERIVE_PARAMS is a structure that provides the parameters to the
 14285 CKM_IKE2_PRF_PLUS_DERIVE mechanism. It is defined as follows:

```
14286     typedef struct CK_IKE2_PRF_PLUS_DERIVE_PARAMS {
14287         CK_MECHANISM_TYPE prfMechanism;
14288         CK_BBOOL bHasSeedKey;
14289         CK_OBJECT_HANDLE hSeedKey;
14290         CK_BYTE_PTR pSeedData;
14291         CK ULONG ulSeedDataLen;
14292     } CK_IKE2_PRF_PLUS_DERIVE_PARAMS;
```

14293

14294 The fields of the structure have the following meanings:

| | |
|---------------------------------------|---|
| 14295 prfMechanism | underlying MAC mechanism used to generate the prf |
| 14296 bHasSeedKey | hSeed key is present |
| 14297 hSeedKey | optional seed from key |
| 14298 pSeedData | optional seed from data |
| 14299 ulSeedDataLen | length of optional seed data. If no seed data is present this value is 0 |

14301 CK_IKE2_PRF_PLUS_DERIVE_PARAMS_PTR is a pointer to a
 14302 CK_IKE2_PRF_PLUS_DERIVE_PARAMS.

14303

14304 ◆ **CK_IKE_PRF_DERIVE_PARAMS; CK_IKE_PRF_DERIVE_PARAMS_PTR**

14305 **CK_IKE_PRF_DERIVE_PARAMS** is a structure that provides the parameters to the
14306 **CKM_IKE_PRF_DERIVE** mechanism. It is defined as follows:

```
14307
14308     typedef struct CK_IKE_PRF_DERIVE_PARAMS {
14309         CK_MECHANISM_TYPE prfMechanism;
14310         CK_BBOOL bDataAsKey;
14311         CK_BBOOL bRekey;
14312         CK_BYTE_PTR pNi;
14313         CK ULONG ulNiLen;
14314         CK_BYTE_PTR pNr;
14315         CK ULONG ulNrLen;
14316         CK_OBJECT_HANDLE hNewKey;
14317     } CK_IKE_PRF_DERIVE_PARAMS;
```

14318 The fields of the structure have the following meanings:

| | | |
|-------|--------------|---|
| 14320 | prfMechanism | underlying MAC mechanism used to generate the prf |
| 14321 | bDataAsKey | Ni Nr is used as the key for the prf rather than baseKey |
| 14322 | bRekey | rekey operation. hNewKey must be present |
| 14323 | pNi | Ni value |
| 14324 | ulNiLen | length of Ni |
| 14325 | pNr | Nr value |
| 14326 | ulNrLen | length of Nr |
| 14327 | hNewKey | New key value to drive the rekey. |

14328 **CK_IKE_PRF_DERIVE_PARAMS_PTR** is a pointer to a **CK_IKE_PRF_DERIVE_PARAMS**.

14329

14330 ◆ **CK_IKE1_PRF_DERIVE_PARAMS; CK_IKE1_PRF_DERIVE_PARAMS_PTR**

14331 **CK_IKE1_PRF_DERIVE_PARAMS** is a structure that provides the parameters to the
14332 **CKM_IKE1_PRF_DERIVE** mechanism. It is defined as follows:

```
14333     typedef struct CK_IKE1_PRF_DERIVE_PARAMS {
14334         CK_MECHANISM_TYPE prfMechanism;
14335         CK_BBOOL bHasPrevKey;
14336         CK_OBJECT_HANDLE hKeygxy;
14337         CK_OBJECT_HANDLE hPrevKey;
14338         CK_BYTE_PTR pCKYi;
14339         CK ULONG ulCKYiLen;
14340         CK_BYTE_PTR pCKYr;
14341         CK ULONG ulCKYrLen;
14342         CK_BYTE keyNumber;
14343     } CK_IKE1_PRF_DERIVE_PARAMS;
```

14344

14345 The fields of the structure have the following meanings:

| | | |
|-------|--------------|---|
| 14346 | prfMechanism | underlying MAC mechanism used to generate the prf |
|-------|--------------|---|

14347 bHasPrevkey hPrevKey is present
14348 hKeygxy handle to the exchanged g^xy key
14349 hPrevKey handle to the previously derived key
14350 pCKYi CKYi value
14351 ulCKYiLen length of CKYi
14352 pCKYr CKYr value
14353 ulCKYrLen length of CKYr
14354 keyNumber unique number for this key derivation

14355 **CK_IKE1_PRF_DERIVE_PARAMS_PTR** is a pointer to a **CK_IKE1_PRF_DERIVE_PARAMS**.

14356

14357 ◆ **CK_IKE1_EXTENDED_DERIVE_PARAMS;**
14358 **CK_IKE1_EXTENDED_DERIVE_PARAMS_PTR**

14359 **CK_IKE1_EXTENDED_DERIVE_PARAMS** is a structure that provides the parameters to the
14360 **CKM_IKE1_EXTENDED_DERIVE** mechanism. It is defined as follows:

14361

```
14362        typedef struct CK_IKE1_EXTENDED_DERIVE_PARAMS {  
14363            CK_MECHANISM_TYPE prfMechanism;  
14364            CK_BBOOL bHasKeygxy;  
14365            CK_OBJECT_HANDLE hKeygxy;  
14366            CK_BYTE_PTR pExtraData;  
14367            CK ULONG ulExtraDataLen;  
14368       } CK_IKE1_EXTENDED_DERIVE_PARAMS;
```

14369 The fields of the structure have the following meanings:

14370 prfMechanism underlying MAC mechanism used to generate the prf
14371 bHasKeygxy hKeygxy key is present
14372 hKeygxy optional key g^xy
14373 pExtraData optional extra data
14374 ulExtraDataLen length of optional extra data. If no extra data is present this value is
14375 0

14376 **CK_IKE2_PRF_PLUS_DERIVE_PARAMS_PTR** is a pointer to a
14377 **CK_IKE2_PRF_PLUS_DERIVE_PARAMS**.

14378

14379 **6.64.3 IKE PRF DERIVE**

14380 The IKE PRF Derive mechanism denoted **CKM_IKE_PRF_DERIVE** is used in IPSEC both IKEv1 and
14381 IKEv2 to generate an initial key that is used to generate additional keys. It takes a
14382 **CK_IKE_PRF_DERIVE_PARAMS** as a mechanism parameter. *baseKey* is the base key passed into
14383 **C_DeriveKey**. *baseKey* must be of type **CKK_GENERIC_SECRET** if *bDataAsKey* is TRUE and the key
14384 type of the underlying prf if *bDataAsKey* is FALSE. *hNewKey* must be of type **CKK_GENERIC_SECRET**.
14385 Depending on the parameter settings, it generates keys with a **CKA_VALUE** of:

14386

14387 1. prf(pNijpNr, baseKey); (*bDataAsKey*=TRUE, *bRekey*=FALSE)

14388 2. prf(baseKey, pNi|pNr); (bDataAsKkey=FALSE, bRekey=FALSE)
14389 3. prf(baseKey, ValueOf(hNewKey)| pNi | pNr); (bDataAsKey=FALSE, bRekey=TRUE)
14390 The resulting output key is always the length of the underlying prf. The combination of
14391 *bDataAsKey*=TRUE and *bRekey*=TRUE is not allowed. If both are set, **CKR_ARGUMENTS_BAD** is
14392 returned.
14393 Case 1 is used in
14394 a. ikev2 (RFC 5996) *baseKey* is called *g[^]ir*, the output is called SKEYSEED
14395 b. ikev1 (RFC 2409) *baseKey* is called *g[^]ir*, the output is called SKEYID
14396 Case 2 is used in ikev1 (RFC 2409) *inkey* is called pre-shared-key, output is called SKEYID
14397 Case 3 is used in ikev2 (RFC 5996) rekey case, *baseKey* is *SK_d*, *hNewKey* is *g[^]ir* (new), the output is
14398 called SKEYSEED. The derived key will have a length of the length of the underlying prf. If
14399 **CKA_VALUE_LEN** is specified, it must equal the underlying prf or **CKR_KEY_SIZE_RANGE** is returned.
14400 If **CKA_KEY_TYPE** is not specified in the template, it will be the underlying key type of the prf.
14401

14402 6.64.4 IKEv1 PRF DERIVE

14403 The IKEv1 PRF Derive mechanism denoted **CKM_IKE1_PRF_DERIVE** is used in IPSEC IKEv1 to
14404 generate various additional keys from the initial SKEYID. It takes a **CK_IKE1_PRF_DERIVE_PARAMS**
14405 as a mechanism parameter. SKEYID is the base key passed into **C_DeriveKey**.

14406

14407 This mechanism derives a key with **CKA_VALUE** set to either:

14408 prf(baseKey, ValueOf(hKeygxy) || pCKYi || pCKYr || key_number)

14409 or

14410 prf(baseKey, ValueOf(hPrevKey) || ValueOf(hKeygxy) || pCKYi || pCKYr || key_number)

14411 depending on the state of *bHasPrevKey*.

14412 The key type of *baseKey* must be the key type of the prf, and the key type of *hKeygxy* must be
14413 **CKK_GENERIC_SECRET**. The key type of *hPrevKey* can be any key type.

14414

14415 This is defined in RFC 2409. For each of the following keys.

14416 *baseKey* is SKEYID, *hKeygxy* is *g[^]xy*

14417 for *outKey* = SKEYID_d, *bHasPrevKey* = false, *key_number* = 0

14418 for *outKey* = SKEYID_a, *hPrevKey* = SKEYID_d, *key_number* = 1

14419 for *outKey* = SKEYID_e, *hPrevKey* = SKEYID_a, *key_number* = 2

14420 If **CKA_VALUE_LEN** is not specified, the resulting key will be the length of the prf. If **CKA_VALUE_LEN**
14421 is greater than the prf, **CKR_KEY_SIZE_RANGE** is returned. If it is less the key is truncated taking the
14422 left most bytes. The value **CKA_KEY_TYPE** must be specified in the template or
14423 **CKR_TEMPLATE_INCOMPLETE** is returned.

14424

14425 6.64.5 IKEv2 PRF PLUS DERIVE

14426 The IKEv2 PRF PLUS Derive mechanism denoted **CKM_IKE2_PRF_PLUS_DERIVE** is used in IPSEC
14427 IKEv2 to derive various additional keys from the initial SKEYSEED. It takes a
14428 **CK_IKE2_PRF_PLUS_DERIVE_PARAMS** as a mechanism parameter. SKEYSEED is the base key
14429 passed into **C_DeriveKey**. The key type of *baseKey* must be the key type of the underlying prf. This
14430 mechanism uses the base key and a feedback version of the prf to generate a single key with sufficient
14431 bytes to cover all additional keys. The application will then use **CKM_EXTRACT_KEY_FROM_KEY**
14432 several times to pull out the various keys. **CKA_VALUE_LEN** must be set in the template and its value

14433 must not be bigger than 255 times the size of the prf function output or **CKR_KEY_SIZE_RANGE** will be
14434 returned. If **CKA_KEY_TYPE** is not specified, the output key type will be **CKK_GENERIC_SECRET**.
14435
14436 This mechanism derives a key with a **CKA_VALUE** of (from RFC 5996):
14437
14438 $\text{prfplus} = T1 | T2 | T3 | T4 | \dots | Tn$
14439 where:
14440 $T1 = \text{prf}(K, S | 0x01)$
14441 $T2 = \text{prf}(K, T1 | S | 0x02)$
14442 $T3 = \text{prf}(K, T2 | S | 0x03)$
14443 $T4 = \text{prf}(K, T3 | S | 0x04)$
14444 .
14445 $Tn = \text{prf}(K, T(n-1) | n)$
14446 $K = \text{baseKey}, S = \text{valueOf}(hSeedKey) | pSeedData$
14447

14448 6.64.6 IKEv1 Extended Derive

14449 The IKE Extended Derive mechanism denoted **CKM_IKE1_EXTENDED_DERIVE** is used in IPSEC
14450 IKEv1 to derive longer keys than **CKM_IKE1_EXTENDED_DERIVE** can from the initial SKEYID. It is
14451 used to support RFC 2409 appendix B and RFC 2409 section 5.5 (Quick Mode). It takes a
14452 **CK_IKE1_EXTENDED_DERIVE_PARAMS** as a mechanism parameter. SKEYID is the base key passed
14453 into **C_DeriveKey**. **CKA_VALUE_LEN** must be set in the template and its value must not be bigger than
14454 255 times the size of the prf function output or **CKR_KEY_SIZE_RANGE** will be returned. If
14455 **CKA_KEY_TYPE** is not specified, the output key type will be **CKK_GENERIC_SECRET**. The key type of
14456 SKEYID must be the key type of the prf, and the key type of *hKeygxy* (if present) must be
14457 **CKK_GENERIC_SECRET**.

14458
14459 This mechanism derives a key with **CKA_VALUE** (from RFC 2409 appendix B and section 5.5):
14460 $Ka = K1 | K2 | K3 | K4 | \dots | Kn$
14461 where:
14462 $K1 = \text{prf}(K, \text{valueOf}(hKeygxy) | pExtraData)$ or $\text{prf}(K, 0x00)$ if *bHashKeygxy* is FALSE and *ulExtraData*
14463 is 0
14464 $K2 = \text{prf}(K, K1 | \text{valueOf}(hKeygxy) | pExtraData)$
14465 $K3 = \text{prf}(K, K2 | \text{valueOf}(hKeygxy) | pExtraData)$
14466 $K4 = \text{prf}(K, K3 | \text{valueOf}(hKeygxy) | pExtraData)$
14467 .
14468 $Kn = \text{prf}(K, K(n-1) | \text{valueOf}(hKeygxy) | pExtraData)$
14469 $K = \text{baseKey}$

14470
14471 If **CKA_VALUE_LEN** is less than or equal to the prf length and *bHasKeygxy* is CK_FALSE, then the new
14472 key is simply the base key truncated to **CKA_VALUE_LEN** (specified in RFC 2409 appendix B).
14473 Otherwise the prf is executed and the derived keys value is **CKA_VALUE_LEN** bytes of the resulting prf.

14474 6.65 HSS

14475 HSS is a mechanism for single-part signatures and verification, following the digital signature algorithm
14476 defined in [RFC 8554] and [NIST 802-208].

14477

14478 *Table 263, HSS Mechanisms vs. Functions*

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

1 Single-part operations only

14479

14480

14481 **6.65.1 Definitions**

14482 This section defines the key type **CKK_HSS** for type **CK_KEY_TYPE** as used in the **CKA_KEY_TYPE** attribute of key objects and domain parameter objects.

14484 Mechanisms:

14485 CKM_HSS_KEY_PAIR_GEN

14486 CKM_HSS

14487 **6.65.2 HSS public key objects**

14488 HSS public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_HSS**) hold HSS public keys.

14489 The following table defines the HSS public key object attributes, in addition to the common attributes defined for this object class:

14491 Table 264, *HSS Public Key Object Attributes*

| Attribute | Data Type | Meaning |
|-----------------------------------|------------|---|
| CKA_HSS_LEVELS ^{2,4} | CK ULONG | The number of levels in the HSS scheme. |
| CKA_HSS_LMS_TYPE ^{2,4} | CK ULONG | The encoding for the Merkle tree heights of the top level LMS tree in the hierarchy. |
| CKA_HSS_LMOTS_TYPE ^{2,4} | CK ULONG | The encoding for the Winternitz parameter of the one-time-signature scheme of the top level LMS tree. |
| CKA_VALUE ^{1,4} | Byte array | XDR-encoded public key as defined in [RFC8554]. |

14492 ¹ Refer to Table 11 for footnotes

14493

14494 The following is a sample template for creating an HSS public key object:

14495

```
14496     CK_OBJECT_CLASS keyClass = CKO_PUBLIC_KEY;
14497     CK_KEY_TYPE keyType = CKK_HSS;
14498     CK_UTF8CHAR label[] = "An HSS public key object";
14499     CK_BYTE value[] = { ... };
```

```
14500     CK_BBOOL true = CK_TRUE;  
14501     CK_BBOOL false = CK_FALSE;  
14502  
14503     CK_ATTRIBUTE template[] = {  
14504         {CKA_CLASS, &keyClass, sizeof(keyClass)},  
14505         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},  
14506         {CKA_TOKEN, &false, sizeof(false)},  
14507         {CKA_LABEL, label, sizeof(label)-1},  
14508         {CKA_VALUE, value, sizeof(value)},  
14509         {CKA_VERIFY, &true, sizeof(true)}  
14510     };
```

14511 **6.65.3 HSS private key objects**

14512 HSS private key objects (object class **CKO_PRIVATE_KEY**, key type **CKK_HSS**) hold HSS private keys.
14513 The following table defines the HSS private key object attributes, in addition to the common attributes
14514 defined for this object class:

14515
14516 *Table 265, HSS Private Key Object Attributes*

| Attribute | Data Type | Meaning |
|---------------------------------------|------------------|---|
| CKA_HSS_LEVELS ^{1,3} | CK ULONG | The number of levels in the HSS scheme. |
| CKA_HSS_LMS_TYPES ^{1,3} | CK ULONG_PTR | A list of encodings for the Merkle tree heights of the LMS trees in the hierarchy from top to bottom. The number of encodings in the array is the ulValueLen component of the attribute divided by the size of CK ULONG. This number must match the CKA_HSS_LEVELS attribute value. |
| CKA_HSS_LMOTS_TYPES ^{1,3} | CK ULONG_PTR | A list of encodings for the Winternitz parameter of the one-time-signature scheme of the LMS trees in the hierarchy from top to bottom. The number of encodings in the array is the ulValueLen component of the attribute divided by the size of CK ULONG. This number must match the CKA_HSS_LEVELS attribute value. |
| CKA_VALUE ^{1,4,6,7} | Byte array | Vendor defined, must include state information. Note that exporting this value is dangerous as it would allow key reuse. |
| CKA_HSS_KEYS_REMAINING ^{2,4} | CK ULONG | The minimum of the following two values: 1) The number of one-time private keys remaining; 2) 2^32-1 |

14517 - Refer to Table 11 for footnotes

14518

14519 The encodings for CKA_HSS_LMOTS_TYPES and CKA_HSS_LMS_TYPES are defined in [RFC 8554]
14520 and [NIST 802-208].

14521

14522 The following is a sample template for creating an LMS private key object:

```
14523
14524     CK_OBJECT_CLASS keyClass = CKO_PRIVATE_KEY;
14525     CK_KEY_TYPE keyType = CKK_HSS;
14526     CK_UTF8CHAR label[] = "An HSS private key object";
14527     CK ULONG hssLevels = 123;
14528     CK ULONG lmsTypes[] = {123,...};
14529     CK ULONG lmotsTypes[] = {123,...};
14530     CK_BYTE value[] = {...};
14531     CK_BBOOL true = CK_TRUE;
14532     CK_BBOOL false = CK_FALSE;
14533     CK_ATTRIBUTE template[] = {
```

```

14534     { CKA_CLASS, &keyClass, sizeof(keyClass) },
14535     { CKA_KEY_TYPE, &keyType, sizeof(keyType) },
14536     { CKA_TOKEN, &true, sizeof(true) },
14537     { CKA_LABEL, label, sizeof(label)-1 },
14538     { CKA_SENSITIVE, &true, sizeof(true) },
14539     { CKA_EXTRACTABLE, &false, sizeof(true) },
14540     { CKA_HSS_LEVELS, &hssLevels, sizeof(hssLevels) },
14541     { CKA_HSS_LMS_TYPES, lmsTypes, sizeof(lmsTypes) },
14542     { CKA_HSS_LMOTS_TYPES, lmotsTypes, sizeof(lmotsTypes) },
14543     { CKA_VALUE, value, sizeof(value) },
14544     { CKA_SIGN, &true, sizeof(true) }
14545 } ;
14546
14547 CKA_SENSITIVE MUST be true, CKA_EXTRACTABLE MUST be false, and CKA_COPYABLE MUST
14548 be false for this key.

```

14549 **6.65.4 HSS key pair generation**

14550 The HSS key pair generation mechanism, denoted **CKM_HSS_KEY_PAIR_GEN**, is a key pair generation
14551 mechanism for HSS.

14552 This mechanism does not have a parameter.

14553 The mechanism generates HSS public/private key pairs for the scheme specified by the
14554 **CKA_HSS_LEVELS**, **CKA_HSS_LMS_TYPES**, and **CKA_HSS_LMOTS_TYPES** attributes of the
14555 template for the private key.

14556 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_HSS_LEVELS**,
14557 **CKA_HSS_LMS_TYPE**, **CKA_HSS_LMOTS_TYPE**, and **CKA_VALUE** attributes to the new public key
14558 and the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_VALUE**, and **CKA_HSS_KEYS_REMAINING** attributes
14559 to the new private key.

14560 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
14561 are not used and must be set to 0.

14562 **6.65.5 HSS without hashing**

14563 The HSS without hashing mechanism, denoted **CKM_HSS**, is a mechanism for single-part signatures and
14564 verification for HSS. (This mechanism corresponds only to the part of LMS that processes the hash value,
14565 which may be of any length; it does not compute the hash value.)

14566 This mechanism does not have a parameter.

14567 For the purposes of these mechanisms, an HSS signature is a byte string with length depending on
14568 **CKA_HSS_LEVELS**, **CKA_HSS_LMS_TYPES**, **CKA_HSS_LMOTS_TYPES** as described in the
14569 following table.

14570 Table 266, *HSS without hashing: Key and Data Length*

| Function | Key type | Input length | Output length |
|-----------------------|-----------------|------------------------------|-------------------------|
| C_Sign ¹ | HSS Private Key | any | 1296-74988 ² |
| C_Verify ¹ | HSS Public Key | any, 1296-74988 ² | N/A |

14571 1 Single-part operations only.

14572 2 $4 + (\text{levels} - 1) * 56 + \text{levels} * (8 + (36 + 32 * p) + h * 32)$ 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.

14574 14575 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure are not used and must be set to 0.

14576 If the number of signatures is exhausted, CKR_KEY_EXHAUSTED will be returned.

14577 **7 PKCS #11 Implementation Conformance**

14578 **7.1 PKCS#11 Consumer Implementation Conformance**

14579 An implementation is a conforming PKCS#11 Consumer if the implementation meets the conditions
14580 specified in one or more consumer profiles specified in [PKCS11-Prof].

14581 A PKCS#11 consumer implementation SHALL be a conforming PKCS#11 Consumer.

14582 If a PKCS#11 consumer implementation claims support for a particular consumer profile, then the
14583 implementation SHALL conform to all normative statements within the clauses specified for that profile
14584 and for any subclauses to each of those clauses.

14585 **7.2 PKCS#11 Provider Implementation Conformance**

14586 An implementation is a conforming PKCS#11 Provider if the implementation meets the conditions
14587 specified in one or more provider profiles specified in [PKCS11-Prof].

14588 A PKCS#11 provider implementation SHALL be a conforming PKCS#11 Provider.

14589 Appendix A. Acknowledgments

14590 The following individuals have participated in the creation of this specification and are gratefully
14591 acknowledged:

14592 Participants:

| Salutation | First Name | Last Name | Company |
|------------|------------|----------------|----------------------------------|
| Dr. | Warren | Armstrong | QuintessenceLabs Pty Ltd. |
| | Anthony | Berglas | Cryptsoft Pty Ltd. |
| Mr. | Dieter | Bong | Utimaco IS GmbH |
| Mr. | Roland | Bramm | PrimeKey Solutions AB |
| | Andrew | Byrne | Dell |
| | Hamish | Cameron | nCipher |
| | Kenli | Chong | QuintessenceLabs Pty Ltd. |
| Mr. | Justin | Corlett | Cryptsoft Pty Ltd. |
| | Xuelei | Fan | Oracle |
| Mr. | Jan | Friedel | Oracle |
| Ms. | Susan | Gleeson | Oracle |
| Mr. | Thomas | Hardjono | M.I.T. |
| Mrs. | Jane | Harnad | OASIS |
| | David | Horton | Dell |
| | Tim | Hudson | Cryptsoft Pty Ltd. |
| Mr. | Gershon | Janssen | Individual |
| Mr. | Jakub | Jelen | Red Hat |
| Dr. | Mark | Joseph | P6R, Inc |
| Mr. | Paul | King | nCipher |
| Ms. | Dina | Kurktchi-Nimeh | Oracle |
| | John | Leiseboer | QuintessenceLabs Pty Ltd. |
| Mr. | John | Leser | Oracle |
| | Chris | Malafis | Red Hat |
| Dr. | Michael | Markowitz | Information Security Corporation |
| Mr. | Scott | Marshall | Cryptsoft Pty Ltd. |

| Salutation | First Name | Last Name | Company |
|------------|------------|----------------|----------------------------------|
| Mr. | Chris | Meyer | Utimaco IS GmbH |
| Mr. | Darren | Moffat | Oracle |
| Dr. | Florian | Poppa | QuintessenceLabs Pty Ltd. |
| | Roland | Reichenberg | Utimaco IS GmbH |
| Mr. | Robert | Relyea | Red Hat |
| Mr. | Jonathan | Schulze-Hewett | Information Security Corporation |
| Mr. | Greg | Scott | Cryptsoft Pty Ltd. |
| Mr. | Martin | Shannon | QuintessenceLabs Pty Ltd. |
| Mr. | Oscar | So | Individual |
| | Patrick | Steuer | IBM |
| Mr. | Gerald | Stueve | Fornetix |
| | Jim | Susoy | P6R, Inc |
| Mr. | Sander | Temme | nCipher |
| Mr. | Manish | Upasani | Utimaco IS GmbH |
| Mr. | Charles | White | Fornetix |
| Ms. | Magda | Zdunkiewicz | Cryptsoft Pty Ltd. |

14593

14594

Appendix B. Manifest constants

14595 The definitions for manifest constants specified in this document can be found in the Additional artifacts
14596 section.

14597

14598

Appendix C. Revision History

14599

| Revision | Date | Editor | Changes Made |
|------------|-------------|-------------|--|
| CSD02 WD01 | 12 May 2022 | Dieter Bong | <p>Changes made compared to Committee Specification CSD01, as working draft of Committee Specification CSD02</p> <ul style="list-style-type: none">- Editorial changes resolving comments by Paul Knight, OASIS, https://lists.oasis-open.org/archives/pkcs11-comment/202203/msg00001.html- Reference [PKCS11-curr] replaced by reference within document- Correction of typos |
| | | | |

14600

14601 Appendix D. Notices

14602 Copyright © OASIS Open 2023. All Rights Reserved.

14603 All capitalized terms in the following text have the meanings assigned to them in the OASIS Intellectual
14604 Property Rights Policy (the "OASIS IPR Policy"). The full [Policy](https://www.oasis-open.org/policies-guidelines/ipr/) may be found at the OASIS website:
14605 [<https://www.oasis-open.org/policies-guidelines/ipr/>].

14606 This document and translations of it may be copied and furnished to others, and derivative works that
14607 comment on or otherwise explain it or assist in its implementation may be prepared, copied, published,
14608 and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice
14609 and this section are included on all such copies and derivative works. However, this document itself may
14610 not be modified in any way, including by removing the copyright notice or references to OASIS, except as
14611 needed for the purpose of developing any document or deliverable produced by an OASIS Technical
14612 Committee (in which case the rules applicable to copyrights, as set forth in the OASIS IPR Policy, must
14613 be followed) or as required to translate it into languages other than English.

14614 The limited permissions granted above are perpetual and will not be revoked by OASIS or its successors
14615 or assigns.

14616 This document and the information contained herein is provided on an "AS IS" basis and OASIS
14617 DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY
14618 WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY
14619 OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A
14620 PARTICULAR PURPOSE. OASIS AND ITS MEMBERS WILL NOT BE LIABLE FOR ANY DIRECT,
14621 INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THIS
14622 DOCUMENT OR ANY PART THEREOF.

14623 As stated in the OASIS IPR Policy, the following three paragraphs in brackets apply to OASIS Standards
14624 Final Deliverable documents (Committee Specifications, OASIS Standards, or Approved Errata).

14625 [OASIS requests that any OASIS Party or any other party that believes it has patent claims that would
14626 necessarily be infringed by implementations of this OASIS Standards Final Deliverable, to notify OASIS
14627 TC Administrator and provide an indication of its willingness to grant patent licenses to such patent claims
14628 in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this
14629 deliverable.]

14630 [OASIS invites any party to contact the OASIS TC Administrator if it is aware of a claim of ownership of
14631 any patent claims that would necessarily be infringed by implementations of this OASIS Standards Final
14632 Deliverable by a patent holder that is not willing to provide a license to such patent claims in a manner
14633 consistent with the IPR Mode of the OASIS Technical Committee that produced this OASIS Standards
14634 Final Deliverable. OASIS may include such claims on its website, but disclaims any obligation to do so.]

14635 [OASIS takes no position regarding the validity or scope of any intellectual property or other rights that
14636 might be claimed to pertain to the implementation or use of the technology described in this OASIS
14637 Standards Final Deliverable or the extent to which any license under such rights might or might not be
14638 available; neither does it represent that it has made any effort to identify any such rights. Information on
14639 OASIS' procedures with respect to rights in any document or deliverable produced by an OASIS
14640 Technical Committee can be found on the OASIS website. Copies of claims of rights made available for
14641 publication and any assurances of licenses to be made available, or the result of an attempt made to
14642 obtain a general license or permission for the use of such proprietary rights by implementers or users of
14643 this OASIS Standards Final Deliverable, can be obtained from the OASIS TC Administrator. OASIS
14644 makes no representation that any information or list of intellectual property rights will at any time be
14645 complete, or that any claims in such list are, in fact, Essential Claims.]

14646 The name "OASIS" is a trademark of [OASIS](https://www.oasis-open.org/policies-guidelines/trademark/), the owner and developer of this document, and should be
14647 used only to refer to the organization and its official outputs. OASIS welcomes reference to, and
14648 implementation and use of, documents, while reserving the right to enforce its marks against misleading
14649 uses. Please see <https://www.oasis-open.org/policies-guidelines/trademark/> for above guidance.