PKCS #11 Cryptographic Token Interface
Current Mechanisms Specification
Version 3.0

Committee Specification Draft 01 /
Public Review Draft 01

29 May 2019

This version:
https://docs.oasis-open.org/pkcs11/pkcs11-curr/v3.0/csprd01/pkcs11-curr-v3.0-csprd01.docx (Authoritative)
https://docs.oasis-open.org/pkcs11/pkcs11-curr/v3.0/csprd01/pkcs11-curr-v3.0-csprd01.html

Previous version:
N/A

Latest version:
https://docs.oasis-open.org/pkcs11/pkcs11-curr/v3.0/pkcs11-curr-v3.0.docx (Authoritative)
https://docs.oasis-open.org/pkcs11/pkcs11-curr/v3.0/pkcs11-curr-v3.0.html

Technical Committee:
OASIS PKCS 11 TC

Chairs:
Tony Cox (tony.cox@cryptsoft.com), Cryptsoft Pty Ltd
Robert Relyea (rrelyea@redhat.com), Red Hat

Editors:
Chris Zimman (chris@wmpp.com), Individual
Dieter Bong (dieter.bong@utimaco.com), Utimaco IS GmbH

Additional artifacts:
This prose specification is one component of a Work Product that also includes:
- PKCS #11 header files:
  https://docs.oasis-open.org/pkcs11/pkcs11-curr/v3.0/csprd01/include/pkcs11-v3.0.docx

Related work:
This specification replaces or supersedes:

This specification is related to:

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

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

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

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

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

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

[PKCS11-Current-v3.0]
Notices

Copyright © OASIS Open 2019. All Rights Reserved.

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

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

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

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

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

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

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

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

1 Introduction .................................................................................................................. 16
  1.1 IPR Policy ................................................................................................................. 16
  1.2 Terminology .............................................................................................................. 16
  1.3 Definitions ................................................................................................................ 16
  1.4 Normative References .............................................................................................. 18
  1.5 Non-Normative References ...................................................................................... 19

2 Mechanisms .................................................................................................................. 22
  2.1 RSA .......................................................................................................................... 22
    2.1.1 Definitions ......................................................................................................... 23
    2.1.2 RSA public key objects ................................................................................. 24
    2.1.3 RSA private key objects .............................................................................. 25
    2.1.4 PKCS #1 RSA key pair generation ............................................................. 26
    2.1.5 X9.31 RSA key pair generation ................................................................. 27
    2.1.6 PKCS #1 v1.5 RSA ....................................................................................... 27
    2.1.7 PKCS #1 RSA OAEP mechanism parameters ........................................... 28
    2.1.8 PKCS #1 RSA OAEP .................................................................................... 30
    2.1.9 PKCS #1 RSA PSS mechanism parameters .............................................. 30
    2.1.10 PKCS #1 RSA PSS ...................................................................................... 31
    2.1.11 ISO/IEC 9796 RSA ..................................................................................... 31
    2.1.12 X.509 (raw) RSA ......................................................................................... 32
    2.1.13 ANSI X9.31 RSA ......................................................................................... 33
    2.1.14 PKCS #1 v1.5 RSA signature with MD2, MD5, SHA-1, SHA-256, SHA-384, SHA-512, RIPE-MD 128 or RIPE-MD 160 ......................................................................................... 34
    2.1.15 PKCS #1 v1.5 RSA signature with SHA-224 ............................................ 34
    2.1.16 PKCS #1 RSA PSS signature with SHA-224 ............................................. 34
    2.1.17 PKCS #1 RSA PSS signature with SHA-1, SHA-256, SHA-384 or SHA-512 .................................................................................................................. 35
    2.1.18 PKCS #1 v1.5 RSA signature with SHA3 .................................................. 35
    2.1.19 PKCS #1 RSA PSS signature with SHA3 .................................................. 35
    2.1.20 ANSI X9.31 RSA signature with SHA-1 ..................................................... 35
    2.1.21 TPM 1.1b and TPM 1.2 PKCS #1 v1.5 RSA ......................................... 36
    2.1.22 TPM 1.1b and TPM 1.2 PKCS #1 RSA OAEP ....................................... 36
    2.1.23 RSA AES KEY WRAP .............................................................................. 37
    2.1.24 RSA AES KEY WRAP mechanism parameters .................................. 38
    2.1.25 FIPS 186-4 ................................................................................................. 39

2.2 DSA .......................................................................................................................... 39
  2.2.1 Definitions ......................................................................................................... 39
  2.2.2 DSA public key objects ................................................................................. 40
  2.2.3 DSA Key Restrictions ...................................................................................... 41
  2.2.4 DSA private key objects .............................................................................. 41
  2.2.5 DSA domain parameter objects .................................................................... 42
  2.2.6 DSA key pair generation .............................................................................. 43
  2.2.7 DSA domain parameter generation ............................................................. 43
  2.2.8 DSA probabilistic domain parameter generation ........................................ 44
  2.2.9 DSA Shawe-Taylor domain parameter generation .................................... 44
2.2.10 DSA base domain parameter generation .......................................................... 44
2.2.11 DSA without hashing ......................................................................................... 45
2.2.12 DSA with SHA-1 ............................................................................................... 45
2.2.13 FIPS 186-4 ....................................................................................................... 45
2.2.14 DSA with SHA-224 ........................................................................................ 46
2.2.15 DSA with SHA-256 ........................................................................................ 46
2.2.16 DSA with SHA-384 ........................................................................................ 46
2.2.17 DSA with SHA-512 ........................................................................................ 47
2.2.18 DSA with SHA3-224 ....................................................................................... 47
2.2.19 DSA with SHA3-256 ....................................................................................... 48
2.2.20 DSA with SHA3-384 ....................................................................................... 48
2.2.21 DSA with SHA3-512 ....................................................................................... 48
2.3 Elliptic Curve ........................................................................................................ 49
  2.3.1 EC Signatures .................................................................................................... 51
  2.3.2 Definitions ........................................................................................................ 51
  2.3.3 ECDSA public key objects ............................................................................. 52
  2.3.4 Elliptic curve private key objects .................................................................... 53
  2.3.5 Edwards Elliptic curve public key objects ..................................................... 55
  2.3.6 Edwards Elliptic curve private key objects .................................................... 55
  2.3.7 Montgomery Elliptic curve public key objects .............................................. 56
  2.3.8 Montgomery Elliptic curve private key objects ............................................. 57
  2.3.9 Elliptic curve key pair generation .................................................................. 58
  2.3.10 Edwards Elliptic curve key pair generation ................................................ 59
  2.3.11 Montgomery Elliptic curve key pair generation ........................................ 59
  2.3.12 ECDSA without hashing ............................................................................. 60
  2.3.13 ECDSA with hashing .................................................................................... 60
  2.3.14 EdDSA ........................................................................................................... 61
  2.3.15 For this mechanism, the 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. XEdDSA .......................................................................................... 61
  2.3.16 EC mechanism parameters ......................................................................... 62
  2.3.17 Elliptic curve Diffie-Hellman key derivation .............................................. 67
  2.3.18 Elliptic curve Diffie-Hellman with cofactor key derivation ........................ 68
  2.3.19 Elliptic curve Menezes-Qu-Vanstone key derivation .................................. 68
  2.3.20 ECDH AES KEY WRAP ........................................................................... 69
  2.3.21 ECDH AES KEY WRAP mechanism parameters ...................................... 70
  2.3.22 FIPS 186-4 .................................................................................................. 71
2.4 Diffie-Hellman ........................................................................................................ 71
  2.4.1 Definitions ....................................................................................................... 72
  2.4.2 Diffie-Hellman public key objects ................................................................ 72
  2.4.3 X9.42 Diffie-Hellman public key objects .................................................... 73
  2.4.4 Diffie-Hellman private key objects ................................................................ 74
  2.4.5 X9.42 Diffie-Hellman private key objects .................................................... 74
  2.4.6 Diffie-Hellman domain parameter objects ................................................... 75
2.12 AES CBC with Cipher Text Stealing CTS .................................................. 101
  2.12.1 Definitions ......................................................................................... 102
  2.12.2 AES CTS mechanism parameters ....................................................... 102
2.13 Additional AES Mechanisms .................................................................... 102
  2.13.1 Definitions ......................................................................................... 102
  2.13.2 AES-GCM Authenticated Encryption / Decryption ............................... 103
  2.13.3 AES-CCM authenticated Encryption / Decryption ............................... 104
  2.13.4 AES-GMAC ....................................................................................... 106
  2.13.5 AES GCM and CCM Mechanism parameters ....................................... 107
2.14 AES CMAC ............................................................................................... 110
  2.14.1 Definitions ......................................................................................... 110
  2.14.2 Mechanism parameters ...................................................................... 110
  2.14.3 General-length AES-CMAC ............................................................... 110
  2.14.4 AES-CMAC ....................................................................................... 110
2.15 AES XTS .................................................................................................... 111
  2.15.1 Definitions ......................................................................................... 111
  2.15.2 AES-XTS secret key objects ............................................................... 111
  2.15.3 AES-XTS key generation .................................................................... 111
  2.15.4 AES-XTS .......................................................................................... 112
2.16 AES Key Wrap .......................................................................................... 112
  2.16.1 Definitions ......................................................................................... 112
  2.16.2 AES Key Wrap Mechanism parameters ............................................. 112
  2.16.3 AES Key Wrap .................................................................................. 112
2.17 Key derivation by data encryption – DES & AES ...................................... 113
  2.17.1 Definitions ......................................................................................... 113
  2.17.2 Mechanism Parameters ...................................................................... 114
  2.17.3 Mechanism Description ...................................................................... 114
2.18 Double and Triple-length DES ................................................................. 114
  2.18.1 Definitions ......................................................................................... 115
  2.18.2 DES2 secret key objects ...................................................................... 115
  2.18.3 DES3 secret key objects ...................................................................... 116
  2.18.4 Double-length DES key generation .................................................... 116
  2.18.5 Triple-length DES Order of Operations ............................................ 117
  2.18.6 Triple-length DES in CBC Mode ....................................................... 117
  2.18.7 DES and Triple length DES in OFB Mode .......................................... 117
  2.18.8 DES and Triple length DES in CFB Mode ........................................ 118
2.19 Double and Triple-length DES CMAC .................................................... 118
  2.19.1 Definitions ......................................................................................... 118
  2.19.2 Mechanism parameters ...................................................................... 119
  2.19.3 General-length DES3-MAC ............................................................... 119
  2.19.4 DES3-CMAC ..................................................................................... 119
2.20 SHA-1 ....................................................................................................... 119
  2.20.1 Definitions ......................................................................................... 120
  2.20.2 SHA-1 digest ...................................................................................... 120
  2.20.3 General-length SHA-1-HMAC ........................................................... 120
  2.20.4 SHA-1-HMAC ................................................................................... 121
2.20.5 SHA-1 key derivation ................................................................. 121
2.20.6 SHA-1 HMAC key generation ..................................................... 121
2.21 SHA-224 ..................................................................................... 122
  2.21.1 Definitions ............................................................................... 122
  2.21.2 SHA-224 digest ................................................................. 122
  2.21.3 General-length SHA-224-HMAC .............................................. 123
  2.21.4 SHA-224-HMAC ............................................................. 123
  2.21.5 SHA-224 key derivation ....................................................... 123
  2.21.6 SHA-224 HMAC key generation .............................................. 123
2.22 SHA-256 ..................................................................................... 124
  2.22.1 Definitions ............................................................................... 124
  2.22.2 SHA-256 digest ................................................................. 124
  2.22.3 General-length SHA-256-HMAC .............................................. 124
  2.22.4 SHA-256-HMAC ............................................................. 125
  2.22.5 SHA-256 key derivation ....................................................... 125
  2.22.6 SHA-256 HMAC key generation .............................................. 125
2.23 SHA-384 ..................................................................................... 125
  2.23.1 Definitions ............................................................................... 126
  2.23.2 SHA-384 digest ................................................................. 126
  2.23.3 General-length SHA-384-HMAC .............................................. 126
  2.23.4 SHA-384-HMAC ............................................................. 126
  2.23.5 SHA-384 key derivation ....................................................... 127
  2.23.6 SHA-384 HMAC key generation .............................................. 127
2.24 SHA-512 ..................................................................................... 127
  2.24.1 Definitions ............................................................................... 127
  2.24.2 SHA-512 digest ................................................................. 127
  2.24.3 General-length SHA-512-HMAC .............................................. 128
  2.24.4 SHA-512-HMAC ............................................................. 128
  2.24.5 SHA-512 key derivation ....................................................... 128
  2.24.6 SHA-512 HMAC key generation .............................................. 128
2.25 SHA-512/224 .............................................................................. 129
  2.25.1 Definitions ............................................................................... 129
  2.25.2 SHA-512/224 digest ........................................................... 129
  2.25.3 General-length SHA-512/224-HMAC ................................. 129
  2.25.4 SHA-512/224-HMAC ....................................................... 130
  2.25.5 SHA-512/224 key derivation ................................................. 130
  2.25.6 SHA-512/224 HMAC key generation ..................................... 130
2.26 SHA-512/256 .............................................................................. 130
  2.26.1 Definitions ............................................................................... 131
  2.26.2 SHA-512/256 digest ........................................................... 131
  2.26.3 General-length SHA-512/256-HMAC ................................. 131
  2.26.4 SHA-512/256-HMAC ....................................................... 132
  2.26.5 SHA-512/256 key derivation ................................................. 132
  2.26.6 SHA-512/256 HMAC key generation ..................................... 132
2.27 SHA-512/t ................................................................................... 132
  2.27.1 Definitions ............................................................................... 133
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.27.2 SHA-512/t digest</td>
<td>133</td>
</tr>
<tr>
<td>2.27.3 General-length SHA-512/t-HMAC</td>
<td>133</td>
</tr>
<tr>
<td>2.27.4 SHA-512/t-HMAC</td>
<td>133</td>
</tr>
<tr>
<td>2.27.5 SHA-512/t key derivation</td>
<td>134</td>
</tr>
<tr>
<td>2.27.6 SHA-512/t HMAC key generation</td>
<td>134</td>
</tr>
<tr>
<td>2.28 SHA3-224</td>
<td>134</td>
</tr>
<tr>
<td>2.28.1 Definitions</td>
<td>134</td>
</tr>
<tr>
<td>2.28.2 SHA3-224 digest</td>
<td>135</td>
</tr>
<tr>
<td>2.28.3 General-length SHA3-224-HMAC</td>
<td>135</td>
</tr>
<tr>
<td>2.28.4 SHA3-224-HMAC</td>
<td>135</td>
</tr>
<tr>
<td>2.28.5 SHA3-224 key derivation</td>
<td>135</td>
</tr>
<tr>
<td>2.28.6 SHA3-224 HMAC key generation</td>
<td>135</td>
</tr>
<tr>
<td>2.29 SHA3-256</td>
<td>136</td>
</tr>
<tr>
<td>2.29.1 Definitions</td>
<td>136</td>
</tr>
<tr>
<td>2.29.2 SHA3-256 digest</td>
<td>136</td>
</tr>
<tr>
<td>2.29.3 General-length SHA3-256-HMAC</td>
<td>136</td>
</tr>
<tr>
<td>2.29.4 SHA3-256-HMAC</td>
<td>137</td>
</tr>
<tr>
<td>2.29.5 SHA3-256 key derivation</td>
<td>137</td>
</tr>
<tr>
<td>2.29.6 SHA3-256 HMAC key generation</td>
<td>137</td>
</tr>
<tr>
<td>2.30 SHA3-384</td>
<td>137</td>
</tr>
<tr>
<td>2.30.1 Definitions</td>
<td>138</td>
</tr>
<tr>
<td>2.30.2 SHA3-384 digest</td>
<td>138</td>
</tr>
<tr>
<td>2.30.3 General-length SHA3-384-HMAC</td>
<td>138</td>
</tr>
<tr>
<td>2.30.4 SHA3-384-HMAC</td>
<td>139</td>
</tr>
<tr>
<td>2.30.5 SHA3-384 key derivation</td>
<td>139</td>
</tr>
<tr>
<td>2.30.6 SHA3-384 HMAC key generation</td>
<td>139</td>
</tr>
<tr>
<td>2.31 SHA3-512</td>
<td>139</td>
</tr>
<tr>
<td>2.31.1 Definitions</td>
<td>140</td>
</tr>
<tr>
<td>2.31.2 SHA3-512 digest</td>
<td>140</td>
</tr>
<tr>
<td>2.31.3 General-length SHA3-512-HMAC</td>
<td>140</td>
</tr>
<tr>
<td>2.31.4 SHA3-512-HMAC</td>
<td>140</td>
</tr>
<tr>
<td>2.31.5 SHA3-512 key derivation</td>
<td>141</td>
</tr>
<tr>
<td>2.31.6 SHA3-512 HMAC key generation</td>
<td>141</td>
</tr>
<tr>
<td>2.32 SHAKE</td>
<td>141</td>
</tr>
<tr>
<td>2.32.1 Definitions</td>
<td>141</td>
</tr>
<tr>
<td>2.32.2 SHAKE Key Derivation</td>
<td>141</td>
</tr>
<tr>
<td>2.33 Blake2b-160</td>
<td>142</td>
</tr>
<tr>
<td>2.33.1 Definitions</td>
<td>142</td>
</tr>
<tr>
<td>2.33.2 BLAKE2B-160 digest</td>
<td>142</td>
</tr>
<tr>
<td>2.33.3 General-length BLAKE2B-160-HMAC</td>
<td>143</td>
</tr>
<tr>
<td>2.33.4 BLAKE2B-160-HMAC</td>
<td>143</td>
</tr>
<tr>
<td>2.33.5 BLAKE2B-160 key derivation</td>
<td>143</td>
</tr>
<tr>
<td>2.33.6 BLAKE2B-160 HMAC key generation</td>
<td>143</td>
</tr>
<tr>
<td>2.34 BLAKE2B-256</td>
<td>143</td>
</tr>
<tr>
<td>2.34.1 Definitions</td>
<td>144</td>
</tr>
<tr>
<td>2.34.2 BLAKE2B-256 digest</td>
<td>144</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>2.34.3</td>
<td>General-length BLAKE2B-256-HMAC</td>
</tr>
<tr>
<td>2.34.4</td>
<td>BLAKE2B-256-HMAC</td>
</tr>
<tr>
<td>2.34.5</td>
<td>BLAKE2B-256 key derivation</td>
</tr>
<tr>
<td>2.34.6</td>
<td>BLAKE2B-256 HMAC key generation</td>
</tr>
<tr>
<td>2.35</td>
<td>BLAKE2B-384</td>
</tr>
<tr>
<td>2.35.1</td>
<td>Definitions</td>
</tr>
<tr>
<td>2.35.2</td>
<td>BLAKE2B-384 digest</td>
</tr>
<tr>
<td>2.35.3</td>
<td>General-length BLAKE2B-384-HMAC</td>
</tr>
<tr>
<td>2.35.4</td>
<td>BLAKE2B-384-HMAC</td>
</tr>
<tr>
<td>2.35.5</td>
<td>BLAKE2B-384 key derivation</td>
</tr>
<tr>
<td>2.35.6</td>
<td>BLAKE2B-384 HMAC key generation</td>
</tr>
<tr>
<td>2.36</td>
<td>BLAKE2B-512</td>
</tr>
<tr>
<td>2.36.1</td>
<td>Definitions</td>
</tr>
<tr>
<td>2.36.2</td>
<td>BLAKE2B-512 digest</td>
</tr>
<tr>
<td>2.36.3</td>
<td>General-length BLAKE2B-512-HMAC</td>
</tr>
<tr>
<td>2.36.4</td>
<td>BLAKE2B-512-HMAC</td>
</tr>
<tr>
<td>2.36.5</td>
<td>BLAKE2B-512 key derivation</td>
</tr>
<tr>
<td>2.36.6</td>
<td>BLAKE2B-512 HMAC key generation</td>
</tr>
<tr>
<td>2.37</td>
<td>PKCS #5 and PKCS #5-style password-based encryption (PBE)</td>
</tr>
<tr>
<td>2.37.1</td>
<td>Definitions</td>
</tr>
<tr>
<td>2.37.2</td>
<td>Password-based encryption/authentication mechanism parameters</td>
</tr>
<tr>
<td>2.37.3</td>
<td>PKCS #5 PBKDF2 key generation mechanism parameters</td>
</tr>
<tr>
<td>2.37.4</td>
<td>PKCS #5 PBKDF2 key generation</td>
</tr>
<tr>
<td>2.38</td>
<td>PKCS #12 password-based encryption/authentication mechanisms</td>
</tr>
<tr>
<td>2.38.1</td>
<td>SHA-1-PBE for 3-key triple-DES-CBC</td>
</tr>
<tr>
<td>2.38.2</td>
<td>SHA-1-PBE for 2-key triple-DES-CBC</td>
</tr>
<tr>
<td>2.38.3</td>
<td>SHA-1-PBA for SHA-1-HMAC</td>
</tr>
<tr>
<td>2.39</td>
<td>SSL</td>
</tr>
<tr>
<td>2.39.1</td>
<td>Definitions</td>
</tr>
<tr>
<td>2.39.2</td>
<td>SSL mechanism parameters</td>
</tr>
<tr>
<td>2.39.3</td>
<td>Pre-master key generation</td>
</tr>
<tr>
<td>2.39.4</td>
<td>Master key derivation</td>
</tr>
<tr>
<td>2.39.5</td>
<td>Master key derivation for Diffie-Hellman</td>
</tr>
<tr>
<td>2.39.6</td>
<td>Key and MAC derivation</td>
</tr>
<tr>
<td>2.39.7</td>
<td>MD5 MACing in SSL 3.0</td>
</tr>
<tr>
<td>2.39.8</td>
<td>SHA-1 MACing in SSL 3.0</td>
</tr>
<tr>
<td>2.40</td>
<td>TLS 1.2 Mechanisms</td>
</tr>
<tr>
<td>2.40.1</td>
<td>Definitions</td>
</tr>
<tr>
<td>2.40.2</td>
<td>TLS 1.2 mechanism parameters</td>
</tr>
<tr>
<td>♦ CK_TLS12_MASTER_KEY_DERIVE_PARAMS; CK_TLS12_MASTER_KEY_DERIVE_PARAMS_PTR</td>
<td>160</td>
</tr>
<tr>
<td>♦ CK_TLS12_KEY_MAT_PARAMS; CK_TLS12_KEY_MAT_PARAMS_PTR</td>
<td>161</td>
</tr>
<tr>
<td>♦ CK_TLS_KDF_PARAMS; CK_TLS_KDF_PARAMS_PTR</td>
<td>161</td>
</tr>
<tr>
<td>♦ CK_TLS_MAC_PARAMS; CK_TLS_MAC_PARAMS_PTR</td>
<td>162</td>
</tr>
<tr>
<td>2.40.3</td>
<td>TLS MAC</td>
</tr>
<tr>
<td>2.40.4</td>
<td>Master key derivation</td>
</tr>
</tbody>
</table>
2.46 Twofish
  2.46.1 Definitions ........................................... 199
  2.46.2 Twofish secret key objects .............................. 199
  2.46.3 Twofish key generation .................................. 200
  2.46.4 Twofish -CBC ........................................... 200
  2.46.5 Twofish-CBC with PKCS padding .......................... 200
2.47 CAMELLIA
  2.47.1 Definitions ........................................... 200
  2.47.2 Camellia secret key objects ............................ 201
  2.47.3 Camellia key generation ................................ 201
  2.47.4 Camellia-ECB ........................................... 202
  2.47.5 Camellia-CBC ........................................... 203
  2.47.6 Camellia-CBC with PKCS padding ........................ 204
  2.47.7 CAMELLIA with Counter mechanism parameters ............ 204
  2.47.8 General-length Camellia-MAC .................................. 205
  2.47.9 Camellia-MAC ........................................... 205
2.48 Key derivation by data encryption - Camellia .................. 206
  2.48.1 Definitions ........................................... 206
  2.48.2 Mechanism Parameters .................................... 206
2.49 ARIA
  2.49.1 Definitions ........................................... 206
  2.49.2 Aria secret key objects ................................... 207
  2.49.3 ARIA key generation ..................................... 207
  2.49.4 ARIA-ECB ............................................. 208
  2.49.5 ARIA-CBC .............................................. 208
  2.49.6 ARIA-CBC with PKCS padding ............................. 209
  2.49.7 General-length ARIA-MAC .................................. 210
  2.49.8 ARIA-MAC .............................................. 210
2.50 Key derivation by data encryption - ARIA ........................ 210
  2.50.1 Definitions ........................................... 211
  2.50.2 Mechanism Parameters .................................... 211
2.51 SEED
  2.51.1 Definitions ........................................... 211
  2.51.2 SEED secret key objects .................................. 212
  2.51.3 SEED key generation ..................................... 212
  2.51.4 SEED-ECB .............................................. 213
  2.51.5 SEED-CBC .............................................. 213
  2.51.6 SEED-CBC with PKCS padding ............................. 213
  2.51.7 General-length SEED-MAC .................................. 214
  2.51.8 SEED-MAC .............................................. 214
2.52 Key derivation by data encryption - SEED ........................ 214
  2.52.1 Definitions ........................................... 214
  2.52.2 Mechanism Parameters .................................... 214
2.53 OTP
  2.53.1 Usage overview .......................................... 214
  2.53.2 Case 1: Generation of OTP values .......................... 215
2.53.3 Case 2: Verification of provided OTP values ........................................... 216
2.53.4 Case 3: Generation of OTP keys .......................................................... 216
2.53.5 OTP objects ......................................................................................... 217
  2.53.5.1 Key objects ....................................................................................... 217
2.53.6 OTP-related notifications ....................................................................... 220
2.53.7 OTP mechanisms .................................................................................. 220
  2.53.7.1 OTP mechanism parameters ............................................................. 220
2.53.8 RSA SecurID ......................................................................................... 224
  2.53.8.1 RSA SecurID secret key objects ...................................................... 224
  2.53.8.2 RSA SecurID key generation ............................................................. 225
  2.53.8.3 SecurID OTP generation and validation ........................................... 226
  2.53.8.4 Return values .................................................................................... 226
2.53.9 OATH HOTP ......................................................................................... 226
  2.53.9.1 OATH HOTP secret key objects ...................................................... 226
  2.53.9.2 HOTP key generation ...................................................................... 227
  2.53.9.3 HOTP OTP generation and validation ............................................. 227
2.53.10 ActivIdentity ACTI .............................................................................. 227
  2.53.10.1 ACTI secret key objects ................................................................. 227
  2.53.10.2 ACTI key generation .................................................................... 228
  2.53.10.3 ACTI OTP generation and validation ............................................ 228

2.54 CT-KIP ...................................................................................................... 229
  2.54.1 Principles of Operation ........................................................................ 229
  2.54.2 Mechanisms ....................................................................................... 229
  2.54.3 Definitions ......................................................................................... 230
  2.54.4 CT-KIP Mechanism parameters .......................................................... 230
  2.54.5 CT-KIP key derivation ......................................................................... 230
  2.54.6 CT-KIP key wrap and key unwrap ...................................................... 231
  2.54.7 CT-KIP signature generation .............................................................. 231

2.55 GOST 28147-89 ....................................................................................... 231
  2.55.1 Definitions ......................................................................................... 232
  2.55.2 GOST 28147-89 secret key objects ..................................................... 232
  2.55.3 GOST 28147-89 domain parameter objects ....................................... 233
  2.55.4 GOST 28147-89 key generation ............................................................ 233
  2.55.5 GOST 28147-89-ECB ....................................................................... 234
  2.55.6 GOST 28147-89 encryption mode except ECB ................................... 234
  2.55.7 GOST 28147-89-MAC ..................................................................... 235
  2.55.8 GOST 28147-89 keys wrapping/unwrapping with GOST 28147-89 .... 235

2.56 GOST R 34.11-94 .................................................................................... 236
  2.56.1 Definitions ......................................................................................... 236
  2.56.2 GOST R 34.11-94 domain parameter objects ..................................... 236
  2.56.3 GOST R 34.11-94 digest .................................................................... 237
  2.56.4 GOST R 34.11-94 HMAC ................................................................. 238

2.57 GOST R 34.10-2001 ............................................................................... 238
  2.57.1 Definitions ......................................................................................... 239
  2.57.2 GOST R 34.10-2001 public key objects ............................................ 239
  2.57.3 GOST R 34.10-2001 private key objects ........................................... 240
  2.57.4 GOST R 34.10-2001 domain parameter objects ............................... 242
B.9 Return values ........................................................................................................................................... 277
Appendix C. Revision History ................................................................................................................................ 280
1 Introduction

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

1.1 IPR Policy

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).

1.2 Terminology

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

1.3 Definitions

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

AES Advanced Encryption Standard, as defined in FIPS PUB 197.

CAMELLIA The Camellia encryption algorithm, as defined in RFC 3713.


CBC Cipher-Block Chaining mode, as defined in FIPS PUB 81.

CDMF Commercial Data Masking Facility, a block encipherment method specified by International Business Machines Corporation and based on DES.

CMAC Cipher-based Message Authenticate Code as defined in [NIST sp800-38b] and [RFC 4493].

CMS Cryptographic Message Syntax (see RFC 2630)

CT-KIP Cryptographic Token Key Initialization Protocol (as defined in [CT-KIP])

DES Data Encryption Standard, as defined in FIPS PUB 46-3.

DSA Digital Signature Algorithm, as defined in FIPS PUB 186-2.

EC Elliptic Curve

ECB Electronic Codebook mode, as defined in FIPS PUB 81.

ECDH Elliptic Curve Diffie-Hellman.
**ECDSA** Elliptic Curve DSA, as in ANSI X9.62.

**ECMQV** Elliptic Curve Menezes-Qu-Vanstone

**GOST 28147-89** The encryption algorithm, as defined in Part 2 [GOST 28147-89] and [RFC 4357] [RFC 4490], and RFC [4491].

**GOST R 34.11-94** Hash algorithm, as defined in [GOST R 34.11-94] and [RFC 4357], [RFC 4490], and [RFC 4491].

**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].

**IV** Initialization Vector.

**MAC** Message Authentication Code.

**MQV** Menezes-Qu-Vanstone

**OAEP** Optimal Asymmetric Encryption Padding for RSA.

**PKCS** Public-Key Cryptography Standards.

**PRF** Pseudo random function.

**PTD** Personal Trusted Device, as defined in MeT-PTD

**RSA** The RSA public-key cryptosystem.

**SHA-1** The (revised) Secure Hash Algorithm with a 160-bit message digest, as defined in FIPS PUB 180-2.

**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.

**SHA-256** The Secure Hash Algorithm with a 256-bit message digest, as defined in FIPS PUB 180-2.

**SHA-384** The Secure Hash Algorithm with a 384-bit message digest, as defined in FIPS PUB 180-2.

**SHA-512** The Secure Hash Algorithm with a 512-bit message digest, as defined in FIPS PUB 180-2.

**SSL** The Secure Sockets Layer 3.0 protocol.

**SO** A Security Officer user.

**TLS** Transport Layer Security.

**WIM** Wireless Identification Module.

1.4 Normative References

[ARIA] National Security Research Institute, Korea, “Block Cipher Algorithm ARIA”,

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

Algorithm, April 2004.
URL: http://www.ietf.org/rfc/rfc3713.txt

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

URL: http://cr.yp.to/chacha/chacha-20080128.pdf


[FIPS SP 800-56A] NIST. Special Publication 800-56A Revision 2: Recommendation for Pair-Wise
URL: http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar2.pdf

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

URL: http://tools.ietf.org/html/rfc6986


[OAEP] M. Bellare, P. Rogaway. Optimal Asymmetric Encryption – How to Encrypt with
URL: http://cseweb.ucsd.edu/users/mihir/papers/oaep.pdf

[PKCS11-Base] PKCS #11 Cryptographic Token Interface Base Specification Version 3.0. Edited
by Chris Zimman and Dieter Bong. Latest version. https://docs.oasis-
open.org/pkcs11/pkcs11-base/v3.0/pkcs11-base-v3.0.html

[PKCS11-Hist] PKCS #11 Cryptographic Token Interface Historical Mechanisms Specification
https://docs.oasis-open.org/pkcs11/pkcs11-hist/v3.0/pkcs11-hist-v3.0.html

[PKCS11-Prof] PKCS #11 Cryptographic Token Interface Profiles Version 3.0. Edited by Tim
Hudson. Latest version. https://docs.oasis-open.org/pkcs11/pkcs11-
profiles/v3.0/pkcs11-profiles-v3.0.html

URL: https://cr.yp.to/mac/poly1305-20050329.pdf

[RFC2119] Bradner, S., “Key words for use in RFCs to Indicate Requirement Levels”, BCP
URL: http://www.ietf.org/rfc/rfc2119.txt
1.5 Non-Normative References


[LEGIFRANCE] Avis relatif aux paramètres de courbes elliptiques définis par l’Etat français (Publication of elliptic curve parameters by the French state) URL: https://www.legifrance.gouv.fr/affichTexte.do?cidTexte=JORFTEXT00002466881


URL: https://tools.ietf.org/html/rfc7748

URL: https://tools.ietf.org/html/rfc8032


[SIGNAL] The X3DH Key Agreement Protocol, Revision 1, 2016-11-04, Moxie Marlinspike, Trevor Perrin (editor)
URL: https://signal.org/docs/specifications/x3dh/

URL: http://www.ietf.org/rfc/rfc5246.txt

URL: http://www.ietf.org/rfc/rfc5246.txt


[XEDDSA] The XEdDSA and VXEdDSA Signature Schemes - Revision 1, 2016-10-20, Trevor Perrin (editor)
URL: https://signal.org/docs/specifications/xeddsa/


2 Mechanisms

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

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

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

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR 1</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CKM_RSA_PKCS_KEY_PAIR_GEN</strong></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CKM_RSA_X9_31_KEY_PAIR_GEN</strong></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CKM_RSA_PKCS</strong></td>
<td>✓ 2</td>
<td>✓ 2</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓ 2</td>
<td></td>
</tr>
<tr>
<td><strong>CKM_RSA_PKCS_OAEP</strong></td>
<td>✓ 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CKM_RSA_PKCS_PSS</strong></td>
<td>✓ 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓ 2</td>
<td></td>
</tr>
<tr>
<td><strong>CKM_RSA_9796</strong></td>
<td>✓ 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CKM_RSA_X_509</strong></td>
<td>✓ 2</td>
<td>✓ 2</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓ 2</td>
<td></td>
</tr>
<tr>
<td><strong>CKM_RSA_X9_31</strong></td>
<td>✓ 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CKM_SHA1_RSA_PKCS</strong></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CKM_SHA256_RSA_PKCS</strong></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CKM_SHA384_RSA_PKCS</strong></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 SR = SignRecover, VR = VerifyRecover.

2 Single-part operations only.

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

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

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

2.1 RSA

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR 1</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CKM_RSA_PKCS_KEY_PAIR_GEN</strong></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CKM_RSA_X9_31_KEY_PAIR_GEN</strong></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CKM_RSA_PKCS</strong></td>
<td>✓ 2</td>
<td>✓ 2</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓ 2</td>
<td></td>
</tr>
<tr>
<td><strong>CKM_RSA_PKCS_OAEP</strong></td>
<td>✓ 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CKM_RSA_PKCS_PSS</strong></td>
<td>✓ 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓ 2</td>
<td></td>
</tr>
<tr>
<td><strong>CKM_RSA_9796</strong></td>
<td>✓ 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CKM_RSA_X_509</strong></td>
<td>✓ 2</td>
<td>✓ 2</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓ 2</td>
<td></td>
</tr>
<tr>
<td><strong>CKM_RSA_X9_31</strong></td>
<td>✓ 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CKM_SHA1_RSA_PKCS</strong></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CKM_SHA256_RSA_PKCS</strong></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CKM_SHA384_RSA_PKCS</strong></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 2.1.1 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
- CKM_RSA_X_509
- CKM_MD2_RSA_PKCS
- CKM_MD5_RSA_PKCS
- CKM_SHA1_RSA_PKCS
- CKM_SHA224_RSA_PKCS
- CKM_SHA256_RSA_PKCS
- CKM_SHA384_RSA_PKCS
- CKM_SHA512_RSA_PKCS
- CKM_RIPEMD128_RSA_PKCS
- CKM_RIPEMD160_RSA_PKCS
- CKM_RSA_PKCS_OAEP
- CKM_RSA_X9_31_KEY_PAIR_GEN
- CKM_RSA_X9_31
- CKM_SHA1_RSA_X9_31
- CKM_RSA_PKCS_PSS
- CKM_SHA1_RSA_PKCS_PSS

---

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_SHA512_RSA_PKCS</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA1_RSA_PKCS_PSS</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA256_RSA_PKCS_PSS</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA384_RSA_PKCS_PSS</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA512_RSA_PKCS_PSS</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA1_RSA_X9_31</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_RSA_PKCS_TPM_1_1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_RSA_PKCS_OAEP_TPM_1_1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA3_224_RSA_PKCS</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA3_256_RSA_PKCS</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA3_384_RSA_PKCS</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA3_512_RSA_PKCS</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA3_224_RSA_PKCS_PSS</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA3_256_RSA_PKCS_PSS</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA3_384_RSA_PKCS_PSS</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA3_512_RSA_PKCS_PSS</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 2.1.2 RSA public key objects

RSA public key objects (object class \texttt{CKO\_PUBLIC\_KEY}, key type \texttt{CKK\_RSA}) hold RSA public keys. The following table defines the RSA public key object attributes, in addition to the common attributes defined for this object class:

#### Table 2, RSA Public Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_MODULUS\textsuperscript{1,4}</td>
<td>Big integer</td>
<td>Modulus (n)</td>
</tr>
<tr>
<td>CKA_MODULUS_BITS\textsuperscript{2,3}</td>
<td>CK_ULONG</td>
<td>Length in bits of modulus (n)</td>
</tr>
<tr>
<td>CKA_PUBLIC_EXPONENT\textsuperscript{1}</td>
<td>Big integer</td>
<td>Public exponent (e)</td>
</tr>
</tbody>
</table>

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

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

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

```c
    CK\_OBJECT\_CLASS class = CKO\_PUBLIC\_KEY;
    CK\_KEY\_TYPE keyType = CKK\_RSA;
    CK\_UTF8CHAR label[\] = \"An RSA public key object\";
    CK\_BYTE modulus[\] = {...};
    CK\_BYTE exponent[\] = {...};
    CK\_BBOOL true = CK\_TRUE;
    CK\_ATTRIBUTE template[\] = {
        {CKA\_CLASS, &class, sizeof(class)},
        {CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},
        {CKA\_TOKEN, &true, sizeof(true)},
        {CKA\_LABEL, label, sizeof(label) -1},
        {CKA\_WRAP, &true, sizeof(true)},
        {CKA\_ENCRYPT, &true, sizeof(true)},
        {CKA\_MODULUS, modulus, sizeof(modulus)}
    }
```
2.1.3 RSA private key objects

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

Table 3, RSA Private Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_MODULUS(^{1,4,6})</td>
<td>Big integer</td>
<td>Modulus (n)</td>
</tr>
<tr>
<td>CKA_PUBLIC_EXPONENT(^{4,6})</td>
<td>Big integer</td>
<td>Public exponent (e)</td>
</tr>
<tr>
<td>CKA_PRIVATE_EXPONENT(^{1,4,6,7})</td>
<td>Big integer</td>
<td>Private exponent (d)</td>
</tr>
<tr>
<td>CKA_PRIME_1(^{4,6,7})</td>
<td>Big integer</td>
<td>Prime (p)</td>
</tr>
<tr>
<td>CKA_PRIME_2(^{24,6,7})</td>
<td>Big integer</td>
<td>Prime (q)</td>
</tr>
<tr>
<td>CKA_EXPONENT_1(^{4,6,7})</td>
<td>Big integer</td>
<td>Private exponent (d) modulo (p - 1)</td>
</tr>
<tr>
<td>CKA_EXPONENT_2(^{24,6,7})</td>
<td>Big integer</td>
<td>Private exponent (d) modulo (q - 1)</td>
</tr>
<tr>
<td>CKA_COEFFICIENT(^{4,6,7})</td>
<td>Big integer</td>
<td>CRT coefficient (q^{-1} \mod p)</td>
</tr>
</tbody>
</table>

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

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

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

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

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

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

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

```c
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_RSA;
CK_UTF8CHAR label[] = “An RSA private key object”;
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
```
CK_BYTE modulus[] = {...};
CK_BYTE publicExponent[] = {...};
CK_BYTE privateExponent[] = {...};
CK_BYTE prime1[] = {...};
CK_BYTE prime2[] = {...};
CK_BYTE exponent1[] = {...};
CK_BYTE exponent2[] = {...};
CK_BYTE coefficient[] = {...};
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_DECRYPT, &true, sizeof(true)},
    {CKA_SIGN, &true, sizeof(true)},
    {CKA_MODULUS, modulus, sizeof(modulus)},
    {CKA_PUBLIC_EXPONENT, publicExponent, sizeof(publicExponent)},
    {CKA_PRIVATE_EXPONENT, privateExponent, sizeof(privateExponent)},
    {CKA_PRIME_1, prime1, sizeof(prime1)},
    {CKA_PRIME_2, prime2, sizeof(prime2)},
    {CKA_EXPONENT_1, exponent1, sizeof(exponent1)},
    {CKA_EXPONENT_2, exponent2, sizeof(exponent2)},
    {CKA_COEFFICIENT, coefficient, sizeof(coefficient)}
};

2.1.4 PKCS #1 RSA key pair generation

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

It does not have a parameter.

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

Note: Implementations strictly compliant with version 2.11 or prior versions may generate an error if this attribute is omitted from the template. Experience has shown that many implementations of 2.11 and prior did allow the CKA_PUBLIC_EXPONENT attribute to be omitted from the template, and behaved as described above. The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, CKA_MODULUS, and CKA_PUBLIC_EXPONENT attributes to the new public key. CKA_PUBLIC_EXPONENT will be copied from the template if supplied. CKR_TEMPLATE_INCONSISTENT shall be returned if the implementation cannot use the supplied exponent value. It contributes the CKA_CLASS and CKA_KEY_TYPE attributes to the new private key; it
may also contribute some of the following attributes to the new private key: CKA_MODULUS, CKA_PUBLIC_EXPONENT, CKA_PRIVATE_EXPONENT, CKA_PRIME_1, CKA_PRIME_2, CKA_EXPONENT_1, CKA_EXPONENT_2, CKA_COEFFICIENT. Other attributes supported by the RSA public and private key types (specifically, the flags indicating which functions the keys support) may also be specified in the templates for the keys, or else are assigned default initial values.

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

### 2.1.5 X9.31 RSA key pair generation

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

It does not have a parameter.

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

The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, CKA_MODULUS, and CKA_PUBLIC_EXPONENT attributes to the new public key. It contributes the CKA_CLASS and CKA_KEY_TYPE attributes to the new private key; it may also contribute some of the following attributes to the new private key: CKA_MODULUS, CKA_PUBLIC_EXPONENT, CKA_PRIVATE_EXPONENT, CKA_PRIME_1, CKA_PRIME_2, CKA_EXPONENT_1, CKA_EXPONENT_2, CKA_COEFFICIENT.

Other attributes supported by the RSA public and private key types (specifically, the flags indicating which functions the keys support) may also be specified in the templates for the keys, or else are assigned default initial values. Unlike the CKM_RSA_PKCS_KEY_PAIR_GEN mechanism, this mechanism is guaranteed to generate $p$ and $q$ values, CKA_PRIME_1 and CKA_PRIME_2 respectively, that meet the strong primes requirement of X9.31.

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

### 2.1.6 PKCS #1 v1.5 RSA

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

This mechanism does not have a parameter.

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

Constraints on key types and the length of the data are summarized in the following table. For encryption, decryption, signatures and signature verification, the input and output data may begin at the same location in memory. In the table, $k$ is the length in bytes of the RSA modulus.
Table 4, PKCS #1 v1.5 RSA: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt(^1)</td>
<td>RSA public key</td>
<td>(\leq k-1)</td>
<td>(k)</td>
<td>block type 02</td>
</tr>
<tr>
<td>C_Decrypt(^1)</td>
<td>RSA private key</td>
<td>(k)</td>
<td>(\leq k-1)</td>
<td>block type 02</td>
</tr>
<tr>
<td>C_Sign(^1)</td>
<td>RSA private key</td>
<td>(\leq k-1)</td>
<td>(k)</td>
<td>block type 01</td>
</tr>
<tr>
<td>C_SignRecover</td>
<td>RSA private key</td>
<td>(\leq k-1)</td>
<td>(k)</td>
<td>block type 01</td>
</tr>
<tr>
<td>C_Verify(^1)</td>
<td>RSA public key</td>
<td>(\leq k-1, k^2)</td>
<td>N/A</td>
<td>block type 01</td>
</tr>
<tr>
<td>C_VerifyRecover</td>
<td>RSA public key</td>
<td>(k)</td>
<td>(\leq k-1)</td>
<td>block type 01</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>RSA public key</td>
<td>(\leq k-1)</td>
<td>(k)</td>
<td>block type 02</td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>RSA private key</td>
<td>(k)</td>
<td>(\leq k-1)</td>
<td>block type 02</td>
</tr>
</tbody>
</table>

\(^1\) Single-part operations only.

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

2.1.7 PKCS #1 RSA OAEP mechanism parameters

\* CK_RSA_PKCS_MGF_TYPE; CK_RSA_PKCS_MGF_TYPE_PTR

CK_RSA_PKCS_MGF_TYPE is used to indicate the Message Generation Function (MGF) applied to a message block when formatting a message block for the PKCS #1 OAEP encryption scheme or the PKCS #1 PSS signature scheme. It is defined as follows:

```c
typedef CK_ULONG CK_RSA_PKCS_MGF_TYPE;
```

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

Table 5, PKCS #1 Mask Generation Functions

<table>
<thead>
<tr>
<th>Source Identifier</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKG_MGF1_SHA1</td>
<td>0x000000001UL</td>
</tr>
<tr>
<td>CKG_MGF1_SHA224</td>
<td>0x000000005UL</td>
</tr>
<tr>
<td>CKG_MGF1_SHA256</td>
<td>0x000000002UL</td>
</tr>
<tr>
<td>CKG_MGF1_SHA384</td>
<td>0x000000003UL</td>
</tr>
<tr>
<td>CKG_MGF1_SHA512</td>
<td>0x000000004UL</td>
</tr>
<tr>
<td>CKG_MGF1_SHA3_224</td>
<td>0x000000006UL</td>
</tr>
<tr>
<td>CKG_MGF1_SHA3_256</td>
<td>0x000000007UL</td>
</tr>
<tr>
<td>CKG_MGF1_SHA3_384</td>
<td>0x000000008UL</td>
</tr>
<tr>
<td>CKG_MGF1_SHA3_512</td>
<td>0x000000009UL</td>
</tr>
</tbody>
</table>

CK_RSA_PKCS_MGF_TYPE_PTR is a pointer to a CK_RSA_PKCS_MGF_TYPE.

\* CK_RSA_PKCS_OAEP_SOURCE_TYPE; CK_RSA_PKCS_OAEP_SOURCE_TYPE_PTR

CK_RSA_PKCS_OAEP_SOURCE_TYPE is used to indicate the source of the encoding parameter when formatting a message block for the PKCS #1 OAEP encryption scheme. It is defined as follows:

```c
typedef CK_ULONG CK_RSA_PKCS_OAEP_SOURCE_TYPE;
```
The following encoding parameter sources are defined in PKCS #1. The following table lists the defined sources along with the corresponding data type for the pSourceData field in the CK_RSA_PKCS_OAEP_PARAMS structure defined below.

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

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

CK_RSA_PKCS_OAEP_SOURCE_TYPE_PTR is a pointer to a CK_RSA_PKCS_OAEP_SOURCE_TYPE.

♦ CK_RSA_PKCS_OAEP_PARAMS; CK_RSA_PKCS_OAEP_PARAMS_PTR

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

```c
typedef struct CK_RSA_PKCS_OAEP_PARAMS {
    CK_MECHANISM_TYPE hashAlg;
    CK_RSA_PKCS_MGF_TYPE mgf;
    CK_RSA_PKCS_OAEP_SOURCE_TYPE source;
    CK_VOID_PTR pSourceData;
    CK_ULONG ulSourceDataLen;
} CK_RSA_PKCS_OAEP_PARAMS;
```

The fields of the structure have the following meanings:

- `hashAlg`: mechanism ID of the message digest algorithm used to calculate the digest of the encoding parameter
- `mgf`: mask generation function to use on the encoded block
- `source`: source of the encoding parameter
- `pSourceData`: data used as the input for the encoding parameter source
- `ulSourceDataLen`: length of the encoding parameter source input

CK_RSA_PKCS_OAEP_PARAMS_PTR is a pointer to a CK_RSA_PKCS_OAEP_PARAMS.

Table 7, Add to following to table 7 (PKCS #1 Mask Generation Functions) in section 2,1..7 (PKCS #1 RSA OAEP mechanism parameters)

<table>
<thead>
<tr>
<th>Source Identifier</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKG_MGF1_SHA3_224</td>
<td>0x00000006UL</td>
</tr>
<tr>
<td>CKG_MGF1_SHA3_256</td>
<td>0x00000007UL</td>
</tr>
<tr>
<td>CKG_MGF1_SHA3_384</td>
<td>0x00000008UL</td>
</tr>
<tr>
<td>CKG_MGF1_SHA3_512</td>
<td>0x00000009UL</td>
</tr>
</tbody>
</table>
2.1.8 PKCS #1 RSA OAEP

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

It has a parameter, a CK_RSA_PKCS_OAEP_PARAMS structure.

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

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

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt(^1)</td>
<td>RSA public key</td>
<td>( \leq k-2hLen )</td>
<td>( k )</td>
</tr>
<tr>
<td>C_Decrypt(^1)</td>
<td>RSA private key</td>
<td>( k )</td>
<td>( \leq k-2hLen )</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>RSA public key</td>
<td>( \leq k-2hLen )</td>
<td>( k )</td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>RSA private key</td>
<td>( k )</td>
<td>( \leq k-2hLen )</td>
</tr>
</tbody>
</table>

\(^1\) Single-part operations only.

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

2.1.9 PKCS #1 RSA PSS mechanism parameters

**CK_RSA_PKCS_PSS_PARAMS; CK_RSA_PKCS_PSS_PARAMS_PTR**

\( \text{CK_RSA_PKCS_PSS_PARAMS} \) is a structure that provides the parameters to the \( \text{CKM_RSA_PKCS_PSS} \) mechanism. The structure is defined as follows:

```c
typedef struct CK_RSA_PKCS_PSS_PARAMS {
    CK_MECHANISM_TYPE hashAlg;
    CK_RSA_PKCS_MGF_TYPE mgf;
    CK_ULONG sLen;
} CK_RSA_PKCS_PSS_PARAMS;
```

The fields of the structure have the following meanings:

- **hashAlg**: hash algorithm used in the PSS encoding; if the signature mechanism does not include message hashing, then this value must be the mechanism used by the application to generate the message hash; if the signature mechanism includes hashing, then this value must match the hash algorithm indicated by the signature mechanism.

- **mgf**: mask generation function to use on the encoded block.
sLen  length, in bytes, of the salt value used in the PSS encoding; typical values are the length of the message hash and zero

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

### 2.1.10 PKCS #1 RSA PSS

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

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

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

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>C_Sign</strong></td>
<td>RSA private key</td>
<td><em>hLen</em></td>
<td><em>k</em></td>
</tr>
<tr>
<td><strong>C_Verify</strong></td>
<td>RSA public key</td>
<td><em>hLen, k</em></td>
<td>N/A</td>
</tr>
</tbody>
</table>

1 Single-part operations only.

2 Data length, signature length.

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

### 2.1.11 ISO/IEC 9796 RSA

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

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

- Data is converted between byte and bit string formats by interpreting the most-significant bit of the leading byte of the byte string as the leftmost bit of the bit string, and the least-significant bit of the trailing byte of the byte string as the rightmost bit of the bit string (this assumes the length in bits of the data is a multiple of 8).

- A signature is converted from a bit string to a byte string by padding the bit string on the left with 0 to 7 zero bits so that the resulting length in bits is a multiple of 8, and converting the resulting bit string as above; it is converted from a byte string to a bit string by converting the byte string as above, and removing bits from the left so that the resulting length in bits is the same as that of the RSA modulus.

This mechanism does not have a parameter.

Constraints on key types and the length of input and output data are summarized in the following table. In the table, *k* is the length in bytes of the RSA modulus.
For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of RSA modulus sizes, in bits.

### 2.1.12 X.509 (raw) RSA

The X.509 (raw) RSA mechanism, denoted **CKM_RSA_X_509**, is a multi-purpose mechanism based on the RSA public-key cryptosystem. It supports single-part encryption and decryption; single-part signatures and verification with and without message recovery; key wrapping; and key unwrapping. All these operations are based on so-called “raw” RSA, as assumed in X.509.

“Raw” RSA as defined here encrypts a byte string by converting it to an integer, most-significant byte first, applying “raw” RSA exponentiation, and converting the result to a byte string, most-significant byte first. The input string, considered as an integer, must be less than the modulus; the output string is also less than the modulus.

This mechanism does not have a parameter.

This mechanism can wrap and unwrap any secret key of appropriate length. Of course, a particular token may not be able to wrap/unwrap every appropriate-length secret key that it supports. For wrapping, the “input” to the encryption operation is the value of the **CKA_VALUE** attribute of the key that is wrapped; similarly for unwrapping. The mechanism does not wrap the key type, key length, or any other information about the key; the application must convey these separately, and supply them when unwrapping the key.

Unfortunately, X.509 does not specify how to perform padding for RSA encryption. For this mechanism, padding should be performed by prepending plaintext data with 0-valued bytes. In effect, to encrypt the sequence of plaintext bytes \(b_1 b_2 \ldots b_n (n \leq k)\), Cryptoki forms \(P = 2^{n-1} b_1 + 2^{n-2} b_2 + \ldots + b_n\). This number must be less than the RSA modulus. The \(k\)-byte ciphertext (\(k\) is the length in bytes of the RSA modulus) is produced by raising \(P\) to the RSA public exponent modulo the RSA modulus. Decryption of a \(k\)-byte ciphertext \(C\) is accomplished by raising \(C\) to the RSA private exponent modulo the RSA modulus, and returning the resulting value as a sequence of exactly \(k\) bytes. If the resulting plaintext is to be used to produce an unwrapped key, then however many bytes are specified in the template for the length of the key are taken *from the end* of this sequence of bytes.

Technically, the above procedures may differ very slightly from certain details of what is specified in X.509.

Executing cryptographic operations using this mechanism can result in the error returns **CKR_DATA_INVALID** (if plaintext is supplied which has the same length as the RSA modulus and is numerically at least as large as the modulus) and **CKR_ENCRYPTED_DATA_INVALID** (if ciphertext is supplied which has the same length as the RSA modulus and is numerically at least as large as the modulus).

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign(^1)</td>
<td>RSA private key</td>
<td>(\leq \lfloor k/2 \rfloor)</td>
<td>(k)</td>
</tr>
<tr>
<td>C_SignRecover</td>
<td>RSA private key</td>
<td>(\leq \lfloor k/2 \rfloor)</td>
<td>(k)</td>
</tr>
<tr>
<td>C_Verify(^1)</td>
<td>RSA public key</td>
<td>(\leq \lfloor k/2 \rfloor)</td>
<td>(k^2)</td>
</tr>
<tr>
<td>C_VerifyRecover</td>
<td>RSA public key</td>
<td>(k)</td>
<td>(\leq \lfloor k/2 \rfloor)</td>
</tr>
</tbody>
</table>

1 Single-part operations only.

2 Data length, signature length.

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

\(\lfloor x \rfloor\) denotes the floor of \(x\), the largest integer less than or equal to \(x\).
Table 11, X.509 (Raw) RSA: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt¹</td>
<td>RSA public key</td>
<td>$\leq k$</td>
<td>$k$</td>
</tr>
<tr>
<td>C_Decrypt¹</td>
<td>RSA private key</td>
<td>$k$</td>
<td>$k$</td>
</tr>
<tr>
<td>C_Sign¹</td>
<td>RSA private key</td>
<td>$\leq k$</td>
<td>$k$</td>
</tr>
<tr>
<td>C_SignRecover</td>
<td>RSA private key</td>
<td>$\leq k$</td>
<td>$k$</td>
</tr>
<tr>
<td>C_Verify¹</td>
<td>RSA public key</td>
<td>$\leq k, k^2$</td>
<td>N/A</td>
</tr>
<tr>
<td>C_VerifyRecover</td>
<td>RSA public key</td>
<td>$k$</td>
<td>$k$</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>RSA public key</td>
<td>$\leq k$</td>
<td>$k$</td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>RSA private key</td>
<td>$k$</td>
<td>$\leq k$ (specified in template)</td>
</tr>
</tbody>
</table>

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

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

This mechanism is intended for compatibility with applications that do not follow the PKCS #1 or ISO/IEC 9796 block formats.

### 2.1.13 ANSI X9.31 RSA

The ANSI X9.31 RSA mechanism, denoted `CKM_RSA_X9_31`, is a mechanism for single-part signatures and verification without message recovery based on the RSA public-key cryptosystem and the block formats defined in ANSI X9.31.

This mechanism applies the header and padding fields of the hash encapsulation. The trailer field must be applied by the application.

This mechanism processes only byte strings, whereas ANSI X9.31 operates on bit strings. Accordingly, the following transformations are performed:

- Data is converted between byte and bit string formats by interpreting the most-significant bit of the leading byte of the byte string as the leftmost bit of the bit string, and the least-significant bit of the trailing byte of the byte string as the rightmost bit of the bit string (this assumes the length in bits of the data is a multiple of 8).

- A signature is converted from a bit string to a byte string by padding the bit string on the left with 0 to 7 zero bits so that the resulting length in bits is a multiple of 8, and converting the resulting bit string as above; it is converted from a byte string to a bit string by converting the byte string as above, and removing bits from the left so that the resulting length in bits is the same as that of the RSA modulus.

This mechanism does not have a parameter.

Constraints on key types and the length of input and output data are summarized in the following table.

In the table, $k$ is the length in bytes of the RSA modulus. For all operations, the $k$ value must be at least 128 and a multiple of 32 as specified in ANSI X9.31.

Table 12, ANSI X9.31 RSA: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign¹</td>
<td>RSA private key</td>
<td>$\leq k-2$</td>
<td>$k$</td>
</tr>
<tr>
<td>C_Verify¹</td>
<td>RSA public key</td>
<td>$\leq k-2, k^2$</td>
<td>N/A</td>
</tr>
</tbody>
</table>

¹ Single-part operations only.
² Data length, signature length.
For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of RSA modulus sizes, in bits.

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

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>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>RSA private key</td>
<td>any</td>
<td>k</td>
<td>block type 01</td>
</tr>
<tr>
<td>C_Verify</td>
<td>RSA public key</td>
<td>any, (k^2)</td>
<td>N/A</td>
<td>block type 01</td>
</tr>
</tbody>
</table>

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.

### 2.1.15 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_SHA\(X\)_RSA_PKCS mechanisms but uses the SHA-224 hash function.

### 2.1.16 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_SHA\(X\)_RSA_PSS mechanisms but uses the SHA-224 hash function.
2.1.17 PKCS #1 RSA PSS signature with SHA-1, SHA-256, SHA-384 or SHA-512

The PKCS #1 RSA PSS signature with SHA-1 mechanism, denoted \texttt{CKM\_SHA1\_RSA\_PKCS\_PSS}, performs single- and multiple-part digital signatures and verification operations without message recovery. The operations performed are as described in PKCS #1 with the object identifier \texttt{id-RSASSA-PSS}, i.e., as in the scheme RSASSA-PSS in PKCS #1 where the underlying hash function is SHA-1.

The PKCS #1 RSA PSS signature with SHA-256, SHA-384, and SHA-512 mechanisms, denoted \texttt{CKM\_SHA256\_RSA\_PKCS\_PSS}, \texttt{CKM\_SHA384\_RSA\_PKCS\_PSS}, and \texttt{CKM\_SHA512\_RSA\_PKCS\_PSS} respectively, perform the same operations using the SHA-256, SHA-384 and SHA-512 hash functions.

The mechanisms have a parameter, a \texttt{CK\_RSA\_PKCS\_PSS\_PARAMS} structure. The \texttt{sLen} field must be less than or equal to \( k^* - 2 - hLen \) where \( hLen \) is the length in bytes of the hash value. \( k^* \) is the length in bytes of the RSA modulus, except if the length in bits of the RSA modulus is one more than a multiple of 8, in which case \( k^* \) is one less than the length in bytes of the RSA modulus.

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>RSA private key</td>
<td>any</td>
<td>( k )</td>
</tr>
<tr>
<td>C_Verify</td>
<td>RSA public key</td>
<td>any, ( k^2 )</td>
<td>N/A</td>
</tr>
</tbody>
</table>

2.1.18 PKCS #1 v1.5 RSA signature with SHA3

The PKCS #1 v1.5 RSA signature with SHA3-224, SHA3-256, SHA3-384, SHA3-512 mechanisms, denoted \texttt{CKM\_SHA3\_224\_RSA\_PKCS}, \texttt{CKM\_SHA3\_256\_RSA\_PKCS}, \texttt{CKM\_SHA3\_384\_RSA\_PKCS}, and \texttt{CKM\_SHA3\_512\_RSA\_PKCS} respectively, performs similarly as the other \texttt{CKM\_SHA\_RSA\_PKCS} mechanisms but uses the corresponding SHA3 hash functions.

2.1.19 PKCS #1 RSA PSS signature with SHA3

The PKCS #1 RSA PSS signature with SHA3-224, SHA3-256, SHA3-384, SHA3-512 mechanisms, denoted \texttt{CKM\_SHA3\_224\_RSA\_PKCS}, \texttt{CKM\_SHA3\_256\_RSA\_PKCS}, \texttt{CKM\_SHA3\_384\_RSA\_PKCS}, and \texttt{CKM\_SHA3\_512\_RSA\_PKCS} respectively, performs similarly as the other \texttt{CKM\_SHA\_RSA\_PKCS} mechanisms but uses the corresponding SHA-3 hash functions.

2.1.20 ANSI X9.31 RSA signature with SHA-1

The ANSI X9.31 RSA signature with SHA-1 mechanism, denoted \texttt{CKM\_SHA1\_RSA\_X9\_31}, performs single- and multiple-part digital signatures and verification operations without message recovery. The operations performed are as described in ANSI X9.31.

This mechanism does not have a parameter.

Constraints on key types and the length of the data for these mechanisms are summarized in the following table. In the table, \( k \) is the length in bytes of the RSA modulus. For all operations, the \( k \) value must be at least 128 and a multiple of 32 as specified in ANSI X9.31.
Table 15, ANSI X9.31 RSA Signatures with SHA-1: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>RSA private key</td>
<td>any</td>
<td>k</td>
</tr>
<tr>
<td>C_Verify</td>
<td>RSA public key</td>
<td>any, (k^2)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

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.

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

The TPM 1.1b and TPM 1.2 PKCS #1 v1.5 RSA mechanism, denoted \(\text{CKM_RSA_PKCS_TPM_1_1}\), is a multi-use mechanism based on the RSA public-key cryptosystem and the block formats initially defined in PKCS #1 v1.5, with additional formatting rules defined in TCPA TPM Specification Version 1.1b.

Additional formatting rules remained the same in TCG TPM Specification 1.2. The mechanism supports single-part encryption and decryption; key wrapping; and key unwrapping.

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

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

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

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt(^1)</td>
<td>RSA public key</td>
<td>(\leq k-11-5)</td>
<td>(k)</td>
</tr>
<tr>
<td>C_Decrypt(^1)</td>
<td>RSA private key</td>
<td>(k)</td>
<td>(\leq k-11-5)</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>RSA public key</td>
<td>(\leq k-11-5)</td>
<td>(k)</td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>RSA private key</td>
<td>(k)</td>
<td>(\leq k-11-5)</td>
</tr>
</tbody>
</table>

\(^1\) Single-part operations only.

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

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

The TPM 1.1b and TPM 1.2 PKCS #1 RSA OAEP mechanism, denoted \(\text{CKM_RSA_PKCS_OAEP_TPM_1_1}\), is a multi-purpose mechanism based on the RSA public-key cryptosystem and the OAEP block format defined in PKCS #1, with additional formatting defined in TCPA TPM Specification Version 1.1b. Additional formatting rules remained the same in TCG TPM Specification 1.2. The mechanism supports single-part encryption and decryption; key wrapping; and key unwrapping.
This mechanism does not have a parameter. It differs from the standard PKCS#1 OAEP RSA encryption mechanism in that the plaintext is wrapped in a TCPA_BOUND_DATA (TPM_BOUND_DATA for TPM 1.2) structure before being submitted to the encryption process and that all of the values of the parameters that are passed to a standard CKM_RSA_PKCS_OAEP operation are fixed. On encryption, the version field of the TCPA_BOUND_DATA (TPM_BOUND_DATA for TPM 1.2) structure must contain 0x01, 0x01, 0x00, 0x00. On decryption, any structure of the form 0x01, 0x01, 0xXX, 0xYY may be accepted.

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

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

For encryption and decryption, the input and output data may begin at the same location in memory. In the table, \( k \) is the length in bytes of the RSA modulus.

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt(^1)</td>
<td>RSA public key</td>
<td>( \leq k)-2-40-5</td>
<td>( k )</td>
</tr>
<tr>
<td>C_Decrypt(^1)</td>
<td>RSA private key</td>
<td>( k )</td>
<td>( \leq k)-2-40-5</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>RSA public key</td>
<td>( \leq k)-2-40-5</td>
<td>( k )</td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>RSA private key</td>
<td>( k )</td>
<td>( \leq k)-2-40-5</td>
</tr>
</tbody>
</table>

\(^1\) Single-part operations only.

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

### 2.1.23 RSA AES KEY WRAP

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

It has a parameter, a `CK_RSA_AES_KEY_WRAP_PARAMS` structure.

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

- A temporary AES key is used for wrapping the target key using `CKM_AES_KEY_WRAP_KWP` mechanism.
- The temporary AES key is wrapped with the wrapping RSA key using `CKM_RSA_PKCS_OAEP` mechanism.

For wrapping, the mechanism -

- Generates a temporary random AES key of \( ulAESKeyBits \) length. This key is not accessible to the user - no handle is returned.
- Wraps the AES key with the wrapping RSA key using `CKM_RSA_PKCS_OAEP` with parameters of `OAEPParams`.
- Wraps the target key with the temporary AES key using `CKM_AES_KEY_WRAP_KWP` ([AES KEYWRAP] section 6.3).
- Zeroizes the temporary AES key.
- Concatenates two wrapped keys and outputs the concatenated blob. The first is the wrapped AES key, and the second is the wrapped target key.

The recommended format for an asymmetric target key being wrapped is as a PKCS8 PrivateKeyInfo.

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

For unwrapping, the mechanism:

- Splits the input into two parts. The first is the wrapped AES key, and the second is the wrapped target key. The length of the first part is equal to the length of the unwrapping RSA key.
- Un-wraps the temporary AES key from the first part with the private RSA key using CKM_RSA_PKCS_OAEP with parameters of OAEPParams.
- Un-wraps the target key from the second part with the temporary AES key using CKM_AES_KEY_WRAP_KWP ([AES KEYWRAP] section 6.3).
- Zeroizes the temporary AES key.
- Returns the handle to the newly unwrapped target key.

Table 18, CKM_RSA_AES_KEY_WRAP Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt-Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_RSA_AES_KEY_WRAP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

\(^1\text{SR = SignRecover, VR = VerifyRecover}\)

2.1.24 RSA AES KEY WRAP mechanism parameters

- **CK_RSA_AES_KEY_WRAP_PARAMS; CK_RSA_AES_KEY_WRAP_PARAMS_PTR**

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

```c
typedef struct CK_RSA_AES_KEY_WRAP_PARAMS {
    CK_ULONG ulAESKeyBits;
    CK_RSA_PKCS_OAEP_PARAMS_PTR pOAEPParams;
} CK_RSA_AES_KEY_WRAP_PARAMS;
```

The fields of the structure have the following meanings:

- `ulAESKeyBits` length of the temporary AES key in bits. Can be only 128, 192 or 256.
- `pOAEPParams` pointer to the parameters of the temporary AES key wrapping. See also the description of PKCS #1 RSA OAEP mechanism parameters.
CK_RSA_AES_KEY_WRAP_PARAMS_PTR is a pointer to a CK_RSA_AES_KEY_WRAP_PARAMS.

2.1.25 FIPS 186-4

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

2.2 DSA

Table 19, DSA Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR¹</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_DSA_KEY_PAIR_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DSA_PARAMETER_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DSA_PROBABALISTIC_PARAMETER_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DSA_SHAWE_TAYLOR_PARAMETER_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DSA_FIPS_G_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DSA</td>
<td>✓²</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_DSA_SHA1</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_DSA_SHA224</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_DSA_SHA256</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_DSA_SHA384</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_DSA_SHA512</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_DSA_SHA3_224</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_DSA_SHA3_256</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_DSA_SHA3_384</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_DSA_SHA3_512</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.2.1 Definitions

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

Mechanisms:

- CKM_DSA_KEY_PAIR_GEN
- CKM_DSA_SHA1
- CKM_DSA_SHA224
- CKM_DSA_SHA256
- CKM_DSA_SHA384
- CKM_DSA_SHA512
- CKM_DSA_SHA3_224
- CKM_DSA_SHA3_256
- CKM_DSA_SHA3_384
- CKM_DSA_SHA3_512
- CKM_DSA_SHA3_96
- CKM_DSA_SHA3_192
- CKM_DSA_SHA3_256
- CKM_DSA_SHA3_384
- CKM_DSA_SHA3_512
CKM_DSA_SHA3_512
CKM_DSA_PARAMETER_GEN
CKM_DSA_PROBABLISTIC_PARAMETER_GEN
CKM_DSA_SHAWE_TAYLOR_PARAMETER_GEN
CKM_DSA_FIPS_G_GEN

♦ CK_DSA_PARAMETER_GEN_PARAM

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

typedef struct CK_DSA_PARAMETER_GEN_PARAM {
    CK_MECHANISM_TYPE hash;
    CK_BYTE_PTR pSeed;
    CK_ULONG ulSeedLen;
    CK_ULONG ulIndex;
} CK_DSA_PARAMETER_GEN_PARAM;

The fields of the structure have the following meanings:

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

2.2.2 DSA public key objects

DSA public key objects (object class CKO_PUBLIC_KEY, key type CKK_DSA) hold DSA public keys.

The following table defines the DSA public key object attributes, in addition to the common attributes defined for this object class:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_PRIME^1,3</td>
<td>Big integer</td>
<td>Prime p (512 to 3072 bits, in steps of 64 bits)</td>
</tr>
<tr>
<td>CKA_SUBPRIME^1,3</td>
<td>Big integer</td>
<td>Subprime q (160, 224 bits, or 256 bits)</td>
</tr>
<tr>
<td>CKA_BASE^1,3</td>
<td>Big integer</td>
<td>Base g</td>
</tr>
<tr>
<td>CKA_VALUE^1,4</td>
<td>Big integer</td>
<td>Public value y</td>
</tr>
</tbody>
</table>

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

The CKA_PRIME, CKA_SUBPRIME and CKA_BASE attribute values are collectively the “DSA domain parameters”. See FIPS PUB 186-4 for more information on DSA keys.
The following is a sample template for creating a DSA public key object:

```c
CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_DSA;
CK_UTF8CHAR label[] = "A DSA public key object";
CK_BYTE prime[] = {...};
CK_BYTE subprime[] = {...};
CK_BYTE base[] = {...};
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label) - 1},
    {CKA_PRIME, prime, sizeof(prime)},
    {CKA_SUBPRIME, subprime, sizeof(subprime)},
    {CKA_BASE, base, sizeof(base)},
    {CKA_VALUE, value, sizeof(value)}
};
```

### 2.2.3 DSA Key Restrictions

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

- Prime: 1024 bits, Subprime: 160
- Prime: 2048 bits, Subprime: 224
- Prime: 2048 bits, Subprime: 256
- Prime: 3072 bits, Subprime: 256

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

### 2.2.4 DSA private key objects

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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_PRIME</td>
<td>Big integer</td>
<td>Prime ( p ) (512 to 1024 bits, in steps of 64 bits)</td>
</tr>
<tr>
<td>CKA_SUBPRIME</td>
<td>Big integer</td>
<td>Subprime ( q ) (160 bits, 224 bits, or 256 bits)</td>
</tr>
<tr>
<td>CKA_BASE</td>
<td>Big integer</td>
<td>Base ( g )</td>
</tr>
<tr>
<td>CKA_VALUE</td>
<td>Big integer</td>
<td>Private value ( x )</td>
</tr>
</tbody>
</table>

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

The **CKA_PRIME**, **CKA_SUBPRIME** and **CKA_BASE** attribute values are collectively the “DSA domain parameters”. See FIPS PUB 186-4 for more information on DSA keys.
Note that when generating a DSA private key, the DSA domain parameters are not specified in the key's template. This is because DSA private keys are only generated as part of a DSA key pair, and the DSA domain parameters for the pair are specified in the template for the DSA public key.

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

```c
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_DSA;
CK_UTF8CHAR label[] = “A DSA private key object”;
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE prime[] = {...};
CK_BYTE subprime[] = {...};
CK_BYTE base[] = {...};
CK_BYTE value[] = {...};

CK_BBOOL true = CK_TRUE;

CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_SUBJECT, subject, sizeof(subject)},
    {CKA_ID, id, sizeof(id)},
    {CKA_SENSITIVE, &true, sizeof(true)},
    {CKA_SIGN, &true, sizeof(true)},
    {CKA_PRIME, prime, sizeof(prime)},
    {CKA_SUBPRIME, subprime, sizeof(subprime)},
    {CKA_BASE, base, sizeof(base)},
    {CKA_VALUE, value, sizeof(value)}
};
```

### 2.2.5 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>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_PRIME</td>
<td>Big integer</td>
<td>Prime $p$ (512 to 1024 bits, in steps of 64 bits)</td>
</tr>
<tr>
<td>CKA_SUBPRIME</td>
<td>Big integer</td>
<td>Subprime $q$ (160 bits, 224 bits, or 256 bits)</td>
</tr>
<tr>
<td>CKA_BASE</td>
<td>Big integer</td>
<td>Base $g$</td>
</tr>
<tr>
<td>CKA_PRIME_BITS</td>
<td>CK_ULONG</td>
<td>Length of the prime value.</td>
</tr>
</tbody>
</table>

* Refer to [PKCS11-Base] 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:

```c
CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
CK_KEY_TYPE keyType = CKK_DSA;
CK_UTF8CHAR label[] = "A DSA domain parameter object";
CK_BYTE prime[] = {...};
CK_BYTE subprime[] = {...};
CK_BYTE base[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_PRIME, prime, sizeof(prime)},
    {CKA_SUBPRIME, subprime, sizeof(subprime)},
    {CKA_BASE, base, sizeof(base)},
};
```

### 2.2.6 DSA key pair generation

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

This mechanism does not have a parameter.

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

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

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

### 2.2.7 DSA domain parameter generation

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

This mechanism does not have a parameter.

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

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

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

The DSA probabilistic domain parameter generation mechanism, denoted

\texttt{CKM_{DSA} \_PROBABLISTIC_{PARAMETER} \_GEN}, is a domain parameter generation mechanism based

on the Digital Signature Algorithm defined in FIPS PUB 186-4, section Appendix A.1.1 Generation and

Validation of Probable Primes..

This mechanism takes a \texttt{CK\_DSA\_PARAMETER\_GEN\_PARAM} which supplies the base hash and

returns the seed (pSeed) and the length (ulSeedLen).

The mechanism generates DSA the prime and subprime domain parameters with a particular prime

length in bits, as specified in the \texttt{CKA\_PRIME\_BITS} attribute of the template and the subprime length as

specified in the \texttt{CKA\_SUBPRIME\_BITS} attribute of the template.

The mechanism contributes the \texttt{CKA\_CLASS, CKA\_KEY\_TYPE, CKA\_PRIME, CKA\_SUBPRIME,}

\texttt{CKA\_PRIME\_BITS, and CKA\_SUBPRIME\_BITS} attributes to the new object. \texttt{CKA\_BASE} is not set by

this call. Other attributes supported by the DSA domain parameter types may also be specified in the

template, or else are assigned default initial values.

For this mechanism, the \texttt{ulMinKeySize} and \texttt{ulMaxKeySize} fields of the \texttt{CK\_MECHANISM\_INFO} structure

specify the supported range of DSA prime sizes, in bits.

2.2.9 DSA Shawe-Taylor domain parameter generation

The DSA Shawe-Taylor domain parameter generation mechanism, denoted

\texttt{CKM_{DSA} \_SHAWE\_TAYLOR\_PARAMETER\_GEN}, is a domain parameter generation mechanism

based on the Digital Signature Algorithm defined in FIPS PUB 186-4, section Appendix A.1.2

Construction and Validation of Provable Primes p and q.

This mechanism takes a \texttt{CK\_DSA\_PARAMETER\_GEN\_PARAM} which supplies the base hash and

returns the seed (pSeed) and the length (ulSeedLen).

The mechanism generates DSA the prime and subprime domain parameters with a particular prime

length in bits, as specified in the \texttt{CKA\_PRIME\_BITS} attribute of the template and the subprime length as

specified in the \texttt{CKA\_SUBPRIME\_BITS} attribute of the template.

The mechanism contributes the \texttt{CKA\_CLASS, CKA\_KEY\_TYPE, CKA\_PRIME, CKA\_SUBPRIME,}

\texttt{CKA\_PRIME\_BITS, and CKA\_SUBPRIME\_BITS} attributes to the new object. \texttt{CKA\_BASE} is not set by

this call. Other attributes supported by the DSA domain parameter types may also be specified in the

template, or else are assigned default initial values.

For this mechanism, the \texttt{ulMinKeySize} and \texttt{ulMaxKeySize} fields of the \texttt{CK\_MECHANISM\_INFO} structure

specify the supported range of DSA prime sizes, in bits.

2.2.10 DSA base domain parameter generation

The DSA base domain parameter generation mechanism, denoted \texttt{CKM\_DSA\_FIPS\_G\_GEN}, is a base

parameter generation mechanism based on the Digital Signature Algorithm defined in FIPS PUB 186-4,

section Appendix A.2 Generation of Generator G.

This mechanism takes a \texttt{CK\_DSA\_PARAMETER\_GEN\_PARAM} which supplies the base hash the seed

(pSeed) and the length (ulSeedLen) and the index value.

The mechanism generates the DSA base with the domain parameter specified in the \texttt{CKA\_PRIME} and

\texttt{CKA\_SUBPRIME} attributes of the template.

The mechanism contributes the \texttt{CKA\_CLASS, CKA\_KEY\_TYPE, and CKA\_BASE} attributes to the new

object. Other attributes supported by the DSA domain parameter types may also be specified in the

template, or else are assigned default initial values.

For this mechanism, the \texttt{ulMinKeySize} and \texttt{ulMaxKeySize} fields of the \texttt{CK\_MECHANISM\_INFO} structure

specify the supported range of DSA prime sizes, in bits.
### 2.2.11 DSA without hashing

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

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

It does not have a parameter.

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

**Table 23, DSA: Key And Data Length**

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign$^1$</td>
<td>DSA private key</td>
<td>20, 28, 32, 48, or 64 bits</td>
<td>$2^\ast$length of subprime</td>
</tr>
<tr>
<td>C_Verify$^1$</td>
<td>DSA public key</td>
<td>(20, 28, 32, 48, or 64 bits), $(2^\ast$length of subprime)$^2$</td>
<td>N/A</td>
</tr>
</tbody>
</table>

$^1$ Single-part operations only.

$^2$ Data length, signature length.

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

### 2.2.12 DSA with SHA-1

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

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

This mechanism does not have a parameter.

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

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>DSA private key</td>
<td>any</td>
<td>$2^\ast$subprime length</td>
</tr>
<tr>
<td>C_Verify</td>
<td>DSA public key</td>
<td>any, $2^\ast$subprime length$^2$</td>
<td>N/A</td>
</tr>
</tbody>
</table>

$^2$ Data length, signature length.

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

### 2.2.13 FIPS 186-4

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

$L = 1024$, $N = 160$
L = 2048, N = 224
L = 2048, N = 256
L = 3072, N = 256

### 2.2.14 DSA with SHA-224

The DSA with SHA-1 mechanism, denoted **CKM_DSA_SHA224**, is a mechanism for single- and multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-4.

This mechanism computes the entire DSA specification, including the hashing with SHA-224.

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

This mechanism does not have a parameter.

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

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>DSA private key</td>
<td>any</td>
<td>(2^{*}\text{subprime}) length</td>
</tr>
<tr>
<td>C_Verify</td>
<td>DSA public key</td>
<td>any, (2^{*}\text{subprime}) length(^2)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

\(^2\) Data length, signature length.

For this mechanism, the \(ulMinKeySize\) and \(ulMaxKeySize\) fields of the CK\_MECHANISM\_INFO structure specify the supported range of DSA prime sizes, in bits.

### 2.2.15 DSA with SHA-256

The DSA with SHA-1 mechanism, denoted **CKM_DSA_SHA256**, is a mechanism for single- and multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-4.

This mechanism computes the entire DSA specification, including the hashing with SHA-256.

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

This mechanism does not have a parameter.

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

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>DSA private key</td>
<td>any</td>
<td>(2^{*}\text{subprime}) length</td>
</tr>
<tr>
<td>C_Verify</td>
<td>DSA public key</td>
<td>any, (2^{*}\text{subprime}) length(^2)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

\(^2\) Data length, signature length.

### 2.2.16 DSA with SHA-384

The DSA with SHA-1 mechanism, denoted **CKM_DSA_SHA384**, is a mechanism for single- and multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-4.

This mechanism computes the entire DSA specification, including the hashing with SHA-384.
For the purposes of this mechanism, a DSA signature is a string of length $2^{\text{subprime}}$, corresponding to the concatenation of the DSA values $r$ and $s$, each represented most-significant byte first.

This mechanism does not have a parameter.

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>DSA private key</td>
<td>any</td>
<td>$2^{\text{subprime}}$ length</td>
</tr>
<tr>
<td>C_Verify</td>
<td>DSA public key</td>
<td>any, $2^{\text{subprime}}$ length$^2$</td>
<td>N/A</td>
</tr>
</tbody>
</table>

$^2$ Data length, signature length.

### 2.2.17 DSA with SHA-512

The DSA with SHA-1 mechanism, denoted CKM_DSA_SHA512, is a mechanism for single- and multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-4. This mechanism computes the entire DSA specification, including the hashing with SHA-512.

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

This mechanism does not have a parameter.

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>DSA private key</td>
<td>any</td>
<td>$2^{\text{subprime}}$ length</td>
</tr>
<tr>
<td>C_Verify</td>
<td>DSA public key</td>
<td>any, $2^{\text{subprime}}$ length$^2$</td>
<td>N/A</td>
</tr>
</tbody>
</table>

$^2$ Data length, signature length.

### 2.2.18 DSA with SHA3-224

The DSA with SHA3-224 mechanism, denoted CKM_DSA_SHA3_224, is a mechanism for single- and multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-4. This mechanism computes the entire DSA specification, including the hashing with SHA3-224.

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

This mechanism does not have a parameter.

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>DSA private key</td>
<td>any</td>
<td>$2^{\text{subprime}}$ length</td>
</tr>
<tr>
<td>C_Verify</td>
<td>DSA public key</td>
<td>any, $2^{\text{subprime}}$ length$^2$</td>
<td>N/A</td>
</tr>
</tbody>
</table>

$^2$ Data length, signature length.
For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of DSA prime sizes, in bits.

### 2.2.19 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>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>DSA private key</td>
<td>any</td>
<td>2*subprime length</td>
</tr>
<tr>
<td>C_Verify</td>
<td>DSA public key</td>
<td>any, 2*subprime length²</td>
<td>N/A</td>
</tr>
</tbody>
</table>

² Data length, signature length.

### 2.2.20 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>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>DSA private key</td>
<td>any</td>
<td>2*subprime length</td>
</tr>
<tr>
<td>C_Verify</td>
<td>DSA public key</td>
<td>any, 2*subprime length²</td>
<td>N/A</td>
</tr>
</tbody>
</table>

² Data length, signature length.

### 2.2.21 DSA with SHA3-512

The DSA with SHA3-512 mechanism, denoted **CKM_DSA_SHA3_512**, is a mechanism for single- and multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-4. This mechanism computes the entire DSA specification, including the hashing with SHA3-512.

For the purposes of this mechanism, a DSA signature is a string of length 2*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 32, DSA with SHA3-512: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>DSA private key</td>
<td>any</td>
<td>$2^*\text{subprime length}$</td>
</tr>
<tr>
<td>C_Verify</td>
<td>DSA public key</td>
<td>any, 2*subprime length$^2$</td>
<td>N/A</td>
</tr>
</tbody>
</table>

$^2$ Data length, signature length.

2.3 Elliptic Curve

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

Table 33, Elliptic Curve Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR$^1$</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_EC_KEY_PAIR_GEN</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_EC_KEY_PAIR_GEN_W_EXTRA_BITS</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_EC_EDWARDS_KEY_PAIR_GEN</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_EC_MONTGOMERY_KEY_PAIR_GEN</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ECDSA</td>
<td>✓$^2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ECDSA_SHA1</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ECDSA_SHA224</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ECDSA_SHA256</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ECDSA_SHA384</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ECDSA_SHA512</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ECDSA_SHA3_224</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ECDSA_SHA3_256</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ECDSA_SHA3_384</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ECDSA_SHA3_512</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_EDDSA</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_XEDDSA</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ECDH1_DERIVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_ECDH1_COFACTOR_DERIVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Mechanism</td>
<td>Functions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Encrypt &amp; Decrypt</td>
<td>Sign &amp; Verify</td>
<td>SR &amp; VR</td>
<td>Digest</td>
<td>Gen. Key/Key Pair</td>
<td>Wrap &amp; Unwrap</td>
<td>Derive</td>
</tr>
<tr>
<td>CKM_ECMQV_DERIVE</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ECDH_AES_KEY_WRAP</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 34, Mechanism Information Flags**

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Mechanism Information Flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKF_EC_F_P</td>
<td>0x00100000UL</td>
<td>True if the mechanism can be used with EC domain parameters over ( F_p )</td>
</tr>
<tr>
<td>CKF_EC_F_2M</td>
<td>0x00200000UL</td>
<td>True if the mechanism can be used with EC domain parameters over ( F_{2^m} )</td>
</tr>
<tr>
<td>CKF_EC_ECPARAMETERS</td>
<td>0x00400000UL</td>
<td>True if the mechanism can be used with EC domain parameters of the choice <code>ecParameters</code></td>
</tr>
<tr>
<td>CKF_EC_OID</td>
<td>0x00800000UL</td>
<td>True if the mechanism can be used with EC domain parameters of the choice <code>oId</code></td>
</tr>
<tr>
<td>CKF_EC_UNCOMPRESS</td>
<td>0x01000000UL</td>
<td>True if the mechanism can be used with elliptic curve point uncompressed</td>
</tr>
<tr>
<td>CKF_EC_COMPRESS</td>
<td>0x02000000UL</td>
<td>True if the mechanism can be used with elliptic curve point compressed</td>
</tr>
<tr>
<td>CKF_EC_CURVENAME</td>
<td>0x04000000UL</td>
<td>True if the mechanism can be used with EC domain parameters of the choice <code>curveName</code></td>
</tr>
</tbody>
</table>

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

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

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

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

In these specifications there are also four representation methods to define the domain parameters for an EC key. Only the `ecParameters`, the `oId` 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 `oId` choice, and the `CKF_EC_CURVENAME` flag identifies a Cryptoki library supporting the `curveName` choice. A Cryptoki library that can perform EC mechanisms must set the appropriate flag(s) for each EC mechanism.

In these specifications, an EC public key (i.e. EC point \( Q \)) or the base point \( G \) when the `ecParameters` choice is used can be represented as an octet string of the uncompressed form or the compressed form. The `CK_MECHANISM_INFO` structure `CKF_EC_UNCOMPRESS` flag identifies a Cryptoki library supporting the uncompressed form whereas the `CKF_EC_COMPRESS` flag identifies a Cryptoki library supporting the compressed form. A Cryptoki library that can perform EC mechanisms must set either or both of these flags for each EC mechanism.
Note that an implementation of a Cryptoki library supporting EC with only one variety, one representation
of domain parameters or one form may encounter difficulties achieving interoperability with other
implementations.

If an attempt to create, generate, derive or unwrap an EC key of an unsupported curve is made, the
attempt should fail with the error code CKR_CURVE_NOT_SUPPORTED. If an attempt to create,
generate, derive, or unwrap an EC key with invalid or of an unsupported representation of domain
parameters is made, that attempt should fail with the error code CKR_DOMAIN_PARAMS_INVALID. If
an attempt to create, generate, derive, or unwrap an EC key of an unsupported form is made, that
attempt should fail with the error code CKR_TEMPLATE_INCONSISTENT.

2.3.1 EC Signatures

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

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
length of n will be used. Any truncation is done by the token.

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

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

2.3.2 Definitions

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

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

Mechanisms:

CKM_EC_KEY_PAIR_GEN
CKM_EC_EDWARDS_KEY_PAIR_GEN
CKM_EC_MONTGOMERY_KEY_PAIR_GEN
CKM_ECDSA
CKM_ECDSA_SHA1
CKM_ECDSA_SHA224
CKM_ECDSA_SHA256
CKM_ECDSA_SHA384
CKM_ECDSA_SHA512
CKM_ECDSA_SHA3_224
2.3.3 ECDSA public key objects

EC (also related to ECDSA) public key objects (object class CKO_PUBLIC_KEY, key type CKK_EC) hold EC public keys. The following table defines the EC public key object attributes, in addition to the common attributes defined for this object class:
Table 35, Elliptic Curve Public Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_EC_PARAMS^1,3</td>
<td>Byte array</td>
<td>DER-encoding of an ANSI X9.62 Parameters value</td>
</tr>
<tr>
<td>CKA_EC_POINT^1,4</td>
<td>Byte array</td>
<td>DER-encoding of ANSI X9.62 ECPoint value Q</td>
</tr>
</tbody>
</table>

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

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

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

```plaintext
Parameters ::= CHOICE {
  ecParameters  ECPARAMETERS,
  oId           CURVES.&id({CurveNames}),
  implicitlyCA  NULL,
  curveName     PrintableString
}
```

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

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

```plaintext
CK_OBJECT_CLASS  class = CKO_PUBLIC_KEY;
CK_KEY_TYPE      keyType = CKK_EC;
CK_UTF8CHAR      label[] = "An EC public key object";
CK_BYTE          ecParams[] = {...};
CK_BYTE          ecPoint[] = {...};
CK_BBOOL         true = CK_TRUE;
CK_ATTRIBUTE     template[] = {
                    {CKA_CLASS, &class, sizeof(class)},
                    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
                    {CKA_TOKEN, &true, sizeof(true)},
                    {CKA_LABEL, label, sizeof(label)-1},
                    {CKA_EC_PARAMS, ecParams, sizeof(ecParams)},
                    {CKA_EC_POINT, ecPoint, sizeof(ecPoint)}
}
```

2.3.4 Elliptic curve private key objects

EC (also related to ECDSA) private key objects (object class CKO_PRIVATE_KEY, key type CKK_EC) hold EC private keys. See Section 2.3 for more information about EC. The following table defines the EC private key object attributes, in addition to the common attributes defined for this object class:
Table 36, Elliptic Curve Private Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_EC_PARAMS&lt;sup&gt;1,4,6&lt;/sup&gt;</td>
<td>Byte array</td>
<td>DER-encoding of an ANSI X962 Parameters value</td>
</tr>
<tr>
<td>CKA_VALUE&lt;sup&gt;1,4,6,7&lt;/sup&gt;</td>
<td>Big integer</td>
<td>ANSI X962 private value d</td>
</tr>
</tbody>
</table>

The CKA_EC_PARAMS attribute value is known as the “EC domain parameters” and is defined in ANSI X962 as a choice of three parameter representation methods with the following syntax:

```
Parameters ::= CHOICE {
  ecParameters   ECPParameters,
  oId            CURVES.&id({CurveNames}),
  implicitlyCA   NULL,
  curveName      PrintableString
}
```

This allows detailed specification of all required values using choice ecParameters, the use of oId as an object identifier substitute for a particular set of elliptic curve domain parameters, or implicitlyCA to indicate that the domain parameters are explicitly defined elsewhere, or curveName to specify a curve name as e.g. define in [ANSI X962], [BRAINPOOL], [SEC 2], [LEGIFRANCE]. The use of oId or curveName is recommended over the choice ecParameters. The choice implicitlyCA must not be used in Cryptoki. Note that when generating an EC private key, the EC domain parameters are not specified in the key’s template. This is because EC private keys are only generated as part of an EC key pair, and the EC domain parameters for the pair are specified in the template for the EC public key.

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

```
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_EC;
CK_UTF8CHAR label[] = “An EC private key object”;
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE ecParams[] = {...};
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
  {CKA_CLASS, &class, sizeof(class)},
  {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
  {CKA_TOKEN, &true, sizeof(true)},
  {CKA_LABEL, label, sizeof(label)-1},
  {CKA_SUBJECT, subject, sizeof(subject)},
  {CKA_ID, id, sizeof(id)},
  {CKA_SENSITIVE, &true, sizeof(true)},
  {CKA_DERIVE, &true, sizeof(true)},
  {CKA_EC_PARAMS, ecParams, sizeof(ecParams)},
  {CKA_VALUE, value, sizeof(value)}
};
```
2.3.5 Edwards Elliptic curve public key objects

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

Table 37, Edwards Elliptic Curve Public Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_EC_PARAMS</td>
<td>Byte array</td>
<td>DER-encoding of a Parameters value as defined above</td>
</tr>
<tr>
<td>CKA_EC_POINT</td>
<td>Byte array</td>
<td>DER-encoding of the b-bit public key value in little endian order as defined in RFC 8032</td>
</tr>
</tbody>
</table>

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

```
Parameters ::= CHOICE {
  ecParameters  ECPParameters,
  oId           CURVES.&id({CurveNames}),
  implicitlyCA  NULL,
  curveName     PrintableString
}
```

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

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

```
CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_EC;
CK_UTF8CHAR label[] = "An Edwards EC public key object";
CK_BYTE ecParams[] = {0x13, 0x0c, 0x65, 0x64, 0x77, 0x61, 0x72, 0x64, 0x73, 0x32, 0x35, 0x31, 0x39};
CK_BYTE ecPoint[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
  {CKA_CLASS, &class, sizeof(class)},
  {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
  {CKA_TOKEN, &true, sizeof(true)},
  {CKA_LABEL, label, sizeof(label)-1},
  {CKA_EC_PARAMS, ecParams, sizeof(ecParams)},
  {CKA_EC_POINT, ecPoint, sizeof(ecPoint)}
};
```

2.3.6 Edwards Elliptic curve private key objects

Edwards EC private key objects (object class CKO_PRIVATE_KEY, key type CKK_EC_EDWARDS) hold Edwards EC private keys. See Section 2.3 for more information about EC. The following table defines the Edwards EC private key object attributes, in addition to the common attributes defined for this object class:
Table 38, Edwards Elliptic Curve Private Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_EC_PARAMS(^1,4,6)</td>
<td>Byte array</td>
<td>DER-encoding of a Parameters value as defined above</td>
</tr>
<tr>
<td>CKA_VALUE(^1,4,6,7)</td>
<td>Big integer</td>
<td>b-bit private key value in little endian order as defined in RFC 8032</td>
</tr>
</tbody>
</table>

- Refer to [PKCS #11-Base] 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 4\(^{th}\) 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;  
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)}
};
```

2.3.7 Montgomery Elliptic curve public key objects

Montgomery EC public key objects (object class CKO_PUBLIC_KEY, key type CKK_EC_MONTGOMERY) hold Montgomery EC public keys. The following table defines the
Montgomery EC public key object attributes, in addition to the common attributes defined for this object class:

Table 39, Montgomery Elliptic Curve Public Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_EC_PARAMS</td>
<td>Byte array</td>
<td>DER-encoding of a Parameters value as defined above</td>
</tr>
<tr>
<td>CKA_EC_POINT</td>
<td>Byte array</td>
<td>DER-encoding of the public key value in little endian order as defined in RFC 7748</td>
</tr>
</tbody>
</table>

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

Parameters ::= CHOICE {
  ecParameters  ECPParameters,
  oId           CURVES.&id({CurveNames}),
  implicitlyCA  NULL,
  curveName     PrintableString
}

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

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

```c
CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_EC;
CK_UTF8CHAR label[] = "A Montgomery EC public key object";
CK_BYTE ecParams[] = {...};
CK_BYTE ecPoint[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
  {CKA_CLASS, &class, sizeof(class)},
  {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
  {CKA_TOKEN, &true, sizeof(true)},
  {CKA_LABEL, label, sizeof(label)-1},
  {CKA_EC_PARAMS, ecParams, sizeof(ecParams)},
  {CKA_EC_POINT, ecPoint, sizeof(ecPoint)}
};
```

### 2.3.8 Montgomery Elliptic curve private key objects

Montgomery EC private key objects (object class CKO_PRIVATE_KEY, key type CKK_EC_MONTGOMERY) hold Montgomery EC private keys. See Section 2.3 for more information about EC. The following table defines the Montgomery EC private key object attributes, in addition to the common attributes defined for this object class:
Table 40, Montgomery Elliptic Curve Private Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_EC_PARAMS¹,⁴,⁶</td>
<td>Byte array</td>
<td>DER-encoding of a Parameters value as defined above</td>
</tr>
<tr>
<td>CKA_VALUE¹,⁴,⁶,⁷</td>
<td>Big integer</td>
<td>Private key value in little endian order as defined in RFC 7748</td>
</tr>
</tbody>
</table>

¹ Refer to [PKCS #11-Base] 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 ⁴ th 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}),
implyCA     NULL,
curveName   PrintableString
}
```

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

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

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

```
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_EC;
CK_UTF8CHAR label[] = “A Montgomery EC private key object”;
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE ecParams[] = {...};
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_SUBJECT, subject, sizeof(subject)},
    {CKA_ID, id, sizeof(id)},
    {CKA_SENSITIVE, &true, sizeof(true)},
    {CKA_DERIVE, &true, sizeof(true)},
    {CKA_VALUE, value, sizeof(value)}
};
```

2.3.9 Elliptic curve key pair generation

The EC (also related to ECDSA) key pair generation mechanism, denoted CKM_EC_KEY_PAIR_GEN, is a key pair generation mechanism that uses the method defined by the ANSI X9.62 and X9.63 standards.
The EC (also related to ECDSA) key pair generation mechanism, denoted CKM_EC_KEY_PAIR_GEN_W_EXTRA_BITS, is a key pair generation mechanism that uses the method defined by FIPS 186-4 Appendix B.4.1.

These mechanisms do not have a parameter.

These mechanisms generate EC public/private key pairs with particular EC domain parameters, as specified in the CKA_EC_PARAMS attribute of the template for the public key. Note that this version of Cryptoki does not include a mechanism for generating these EC domain parameters.

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

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

2.3.10 Edwards Elliptic curve key pair generation

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

This mechanism does not have a parameter.

The mechanism can only generate EC public/private key pairs over the curves edwards25519 and edwards448 as defined in RFC 8032 or the curves id-Ed25519 and id-Ed448 as defined in RFC 8410.

These curves can only be specified in the CKA_EC_PARAMS attribute of the template for the public key using the curveName or the oldId methods. Attempts to generate keys over these curves using any other EC key pair generation mechanism will fail with CKR_CURVE_NOT_SUPPORTED.

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

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

2.3.11 Montgomery Elliptic curve key pair generation

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

This mechanism does not have a parameter.

The mechanism can only generate Montgomery EC public/private key pairs over the curves curve25519 and curve448 as defined in RFC 7748 or the curves id-X25519 and id-X448 as defined in RFC 8410.

These curves can only be specified in the CKA_EC_PARAMS attribute of the template for the public key using the curveName or old methods. Attempts to generate keys over these curves using any other EC key pair generation mechanism will fail with CKR_CURVE_NOT_SUPPORTED.

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

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the minimum and maximum supported number of bits in the field sizes, respectively. For this mechanism, the only allowed values are 255 and 448 as RFC 7748 only defines curves of these two sizes. A Cryptoki implementation may support one or both of these curves and should set the ulMinKeySize and ulMaxKeySize fields accordingly.
attributes to the new private key. Other attributes supported by the EC public and private key types (specifically, the flags indicating which functions the keys support) may also be specified in the templates for the keys, or else are assigned default initial values.

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

### 2.3.12 ECDSA without hashing

Refer section 2.3.1 for signature encoding.

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

This mechanism does not have a parameter.

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign¹</td>
<td>ECDSA private key</td>
<td>any³</td>
<td>2nLen</td>
</tr>
<tr>
<td>C_Verify¹</td>
<td>ECDSA public key</td>
<td>any³, ≤2nLen²</td>
<td>N/A</td>
</tr>
</tbody>
</table>

¹ Single-part operations only.
² Data length, signature length.
³ Input the entire raw digest. Internally, this will be truncated to the appropriate number of bits.

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

### 2.3.13 ECDSA with hashing

Refer to section 2.3.1 for signature encoding.

The ECDSA with SHA-1, SHA-224, SHA-384, SHA-512, SHA3-224, SHA3-256, SHA3-384, SHA3-512 mechanism, denoted CKM_ECDSA_[SHA1|SHA224|SHA384|SHA512|SHA3 _224|SHA3 _256|SHA3 _384|SHA3 _512] respectively, is a mechanism for single- and multiple-part signatures and verification for ECDSA. This mechanism computes the entire ECDSA specification, including the hashing with SHA-1, SHA-224, SHA-384, SHA-512, SHA3-224, SHA3-256, SHA3-384, SHA3-512 respectively. This mechanism does not have a parameter.

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>ECDSA private key</td>
<td>any</td>
<td>2nLen</td>
</tr>
<tr>
<td>C_Verify</td>
<td>ECDSA public key</td>
<td>any, ≤2nLen²</td>
<td>N/A</td>
</tr>
</tbody>
</table>

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

### 2.3.14 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. Table 32 enumerates the five signature schemes defined in RFC 8032 and all supported permutations of the mechanism parameter and its content.

<table>
<thead>
<tr>
<th>Signature Scheme</th>
<th>Mechanism Param</th>
<th>phFlag</th>
<th>Context Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ed25519</td>
<td>Not Required</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Ed25519ctx</td>
<td>Required</td>
<td>False</td>
<td>Optional</td>
</tr>
<tr>
<td>Ed25519ph</td>
<td>Required</td>
<td>True</td>
<td>Optional</td>
</tr>
<tr>
<td>Ed448</td>
<td>Required</td>
<td>False</td>
<td>Optional</td>
</tr>
<tr>
<td>Ed448ph</td>
<td>Required</td>
<td>True</td>
<td>Optional</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>CKK_EC_EDWARDS private key</td>
<td>any</td>
<td>2bLen</td>
</tr>
<tr>
<td>C_Verify</td>
<td>CKK_EC_EDWARDS public key</td>
<td>any, ≤2bLen 2</td>
<td>N/A</td>
</tr>
</tbody>
</table>

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 CKM_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.

### 2.3.15 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 45, XEdDSA: Key and Data Length**

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign¹</td>
<td>CKK_EC_MONTGOMERY private</td>
<td>any³</td>
<td>2b</td>
</tr>
<tr>
<td>C_Verify¹</td>
<td>CKK_EC_MONTGOMERY public</td>
<td>any³, ≤2b²</td>
<td>N/A</td>
</tr>
</tbody>
</table>

² Data length, signature length.

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

### 2.3.16 EC mechanism parameters

* **CK_EDDSA_PARAMS, CK_EDDSA_PARAMS_PTR**

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

  ```c
  typedef struct CK_EDDSA_PARAMS {
    CK_BBOOL   phFlag;
    CK_ULONG   ulContextDataLen;
    CK_BYTE_PTR pContextData;
  } CK_EDDSA_PARAMS;
  ```

  The fields of the structure have the following meanings:

  - *phFlag* a Boolean value which indicates if Prehashed variant of EdDSA should used
  - *ulContextDataLen* the length in bytes of the context data where 0 <= *ulContextDataLen* <= 255.
  - *pContextData* context data shared between the signer and verifier

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

* **CK_XEDDSA_PARAMS, CK_XEDDSA_PARAMS_PTR**

  **CK_XEDDSA_PARAMS** is a structure that provides the parameters for the **CKM_XEDDSA** signature
  mechanism. The structure is defined as follows:
typedef struct CK_XEDDSA_PARAMS {
    CK_XEDDSA_HASH_TYPE hash;
} CK_XEDDSA_PARAMS;

The fields of the structure have the following meanings:

hash     a Hash mechanism to be used by the mechanism.

CK_XEDDSA_PARAMS_PTR is a pointer to a CK_XEDDSA_PARAMS.

CK_XEDDSA_HASH_TYPE_PTR is a pointer to a CK_XEDDSA_HASH_TYPE.

CK_XEDDSA_HASH_TYPE is used to indicate the hash function used in XEDDSA. It is defined as follows:

typedef CK_ULONG CK_XEDDSA_HASH_TYPE;

The following table lists the defined functions.

Table 46, EC: Key Derivation Functions

<table>
<thead>
<tr>
<th>Source Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_BLAKE2B_256</td>
</tr>
<tr>
<td>CKM_BLAKE2B_512</td>
</tr>
<tr>
<td>CKM_SHA3_256</td>
</tr>
<tr>
<td>CKM_SHA3_512</td>
</tr>
<tr>
<td>CKM_SHA256</td>
</tr>
<tr>
<td>CKM_SHA512</td>
</tr>
</tbody>
</table>

CK_XEDDSA_HASH_TYPE_PTR is a pointer to a CK_XEDDSA_HASH_TYPE.

CK_EC_KDF_TYPE_PTR is a pointer to a CK_EC_KDF_TYPE.

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

typedef CK_ULONG CK_EC_KDF_TYPE;

The following table lists the defined functions.

Table 47, EC: Key Derivation Functions

<table>
<thead>
<tr>
<th>Source Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKD_NULL</td>
</tr>
<tr>
<td>CKD_SHA1_KDF</td>
</tr>
<tr>
<td>CKD_SHA224_KDF</td>
</tr>
<tr>
<td>CKD_SHA256_KDF</td>
</tr>
<tr>
<td>CKD_SHA384_KDF</td>
</tr>
<tr>
<td>CKD_SHA512_KDF</td>
</tr>
<tr>
<td>CKD_SHA3_224_KDF</td>
</tr>
</tbody>
</table>
The key derivation function **CKD_NULL** produces a raw shared secret value without applying any key derivation function.

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

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

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

**CK_ECDH1_DERIVE_PARAMS, CK_ECDH1_DERIVE_PARAMS_PTR**

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

```c
typedef struct CK_ECDH1_DERIVE_PARAMS {
    CK_EC_KDF_TYPE kdf;
    CK_ULONG ulSharedDataLen;
    CK_BYTE_PTR pSharedData;
    CK_ULONG ulPublicDataLen;
    CK_BYTE_PTR pPublicData;
} CK_ECDH1_DERIVE_PARAMS;
```

The fields of the structure have the following meanings:

- **kdf**  
  key derivation function used on the shared secret value
ulSharedDataLen  the length in bytes of the shared info
pSharedData    some data shared between the two parties
ulPublicDataLen the length in bytes of the other party’s EC public key

pPublicData\(^1\)  pointer to other party’s EC public key value. A token MUST be able to accept this value encoded as a raw octet string (as per section A.5.2 of [ANSI X9.62]). A token MAY, in addition, support accepting this value as a DER-encoded ECPoint (as per section E.6 of [ANSI X9.62]) i.e. the same as a CKA_EC_POINT encoding. The calling application is responsible for converting the offered public key to the compressed or uncompressed forms of these encodings if the token does not support the offered form.

With the key derivation function CKD_NULL, pSharedData must be NULL and ulSharedDataLen must be zero. With the key derivation functions CKD_[SHA1|SHA224|SHA384|SHA512|SHA3_224|SHA3_256|SHA3_384|SHA3_512]_KDF, CKD_[SHA1|SHA224|SHA384|SHA512|SHA3_224|SHA3_256|SHA3_384|SHA3_512]_KDF_SP800, an optional pSharedData may be supplied, which consists of some data shared by the two parties intending to share the shared secret. Otherwise, pSharedData must be NULL and ulSharedDataLen must be zero.

CK_ECDH1_DERIVE_PARAMS_PTR is a pointer to a CK_ECDH1_DERIVE_PARAMS.

CK_ECDH2_DERIVE_PARAMS is a structure that provides the parameters to the CKM_ECMQV_DERIVE key derivation mechanism, where each party contributes two key pairs. The structure is defined as follows:

```c
typedef struct CK_ECDH2_DERIVE_PARAMS {
    CK_KDF_TYPE kdf;
    CK_ULONG ulSharedDataLen;
    CK_BYTE_PTR pSharedData;
    CK_ULONG ulPublicDataLen;
    CK_BYTE_PTR pPublicData;
    CK_ULONG ulPrivateDataLen;
    CK_OBJECT_HANDLE hPrivateData;
    CK_ULONG ulPublicDataLen2;
    CK_BYTE_PTR pPublicData2;
} CK_ECDH2_DERIVE_PARAMS;
```

The fields of the structure have the following meanings:

- **kdf**  key derivation function used on the shared secret value
- **ulSharedDataLen**  the length in bytes of the shared info
- **pSharedData**  some data shared between the two parties
- **ulPublicDataLen**  the length in bytes of the other party’s first EC public 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.
pPublicData  pointer to other party's first EC public key value. Encoding rules are as per pPublicData of CK_ECDH1_DERIVE_PARAMS

ulPrivateDataLen  the length in bytes of the second EC private key

hPrivateData  key handle for second EC private key value

ulPublicDataLen2  the length in bytes of the other party's second EC public key

pPublicData2  pointer to other party's second EC public key value. Encoding rules are as per pPublicData of CK_ECDH1_DERIVE_PARAMS

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.

CK_ECDH2_DERIVE_PARAMS_PTR is a pointer to a CK_ECDH2_DERIVE_PARAMS.

CK_ECMQV_DERIVE_PARAMS, CK_ECMQV_DERIVE_PARAMS_PTR

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

typedef struct CK_ECMQV_DERIVE_PARAMS {
    CK_EC_KDF_TYPE  kdf;
    CK_ULONG  ulSharedDataLen;
    CK_BYTE_PTR  pSharedData;
    CK_ULONG  ulPublicDataLen;
    CK_BYTE_PTR  pPublicData;
    CK_ULONG  ulPrivateDataLen;
    CK_OBJECT_HANDLE  hPrivateData;
    CK_ULONG  ulPublicDataLen2;
    CK_BYTE_PTR  pPublicData2;
    CK_OBJECT_HANDLE  publicKey;
}  CK_ECMQV_DERIVE_PARAMS;

The fields of the structure have the following meanings:

kdf  key derivation function used on the shared secret value

ulSharedDataLen  the length in bytes of the shared info

pSharedData  some data shared between the two parties

ulPublicDataLen  the length in bytes of the other party's first EC public key

pPublicData  pointer to other party's first EC public key value. Encoding rules are as per pPublicData of CK_ECDH1_DERIVE_PARAMS

ulPrivateDataLen  the length in bytes of the second EC private key

hPrivateData  key handle for second EC private key value
The elliptic curve Diffie-Hellman (ECDH) key derivation mechanism, denoted `CKM_ECDH1_DERIVE`, is a mechanism for key derivation based on the Diffie-Hellman version of the elliptic curve key agreement scheme, as defined in ANSI X9.63, where each party contributes one key pair all using the same EC domain parameters.

It has a parameter, a `CK_ECDH1_DERIVE_PARAMS` structure.

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

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

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

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

Constraints on key types are summarized in the following table:
### 2.3.18 Elliptic curve Diffie-Hellman with cofactor key derivation

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

It has a parameter, a CK_ECDH1_DERIVE_PARAMS structure.

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

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

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

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

Constraints on key types are summarized in the following table:

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Derive</td>
<td>CKK_EC or CKK_EC_MONTGOMERY</td>
</tr>
</tbody>
</table>

### 2.3.19 Elliptic curve Menezes-Qu-Vanstone key derivation

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

It has a parameter, a CK_ECMQV_DERIVE_PARAMS structure.

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

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

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

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

Constraints on key types are summarized in the following table:

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Derive</td>
<td>CKK_EC</td>
</tr>
</tbody>
</table>

### 2.3.20 ECDH AES KEY WRAP

The ECDH AES KEY WRAP mechanism, denoted CKM_ECDH_AES_KEY_WRAP, is a mechanism based on elliptic curve public-key crypto-system and the AES key wrap mechanism. It supports single-part key wrapping; and key unwrapping.

It has a parameter, a CK_ECDH_AES_KEY_WRAP_PARAMS structure.

The mechanism can wrap and unwrap an asymmetric target key of any length and type using an EC key.

- A temporary AES key is derived from a temporary EC key and the wrapping EC key using the CKM_ECDH1_DERIVE mechanism.
- The derived AES key is used for wrapping the target key using the CKM_AES_KEY_WRAP_KWP mechanism.

For wrapping, the mechanism -

- Generates a temporary random EC key (transport key) having the same parameters as the wrapping EC key (and domain parameters). Saves the transport key public key material.
- Performs ECDH operation using CKM_ECDH1_DERIVE with parameters of kdf, ulSharedDataLen and pSharedData using the private key of the transport EC key and the public key of wrapping EC key and gets the first ulAESKeyBits bits of the derived key to be the temporary AES key.
- Wraps the target key with the temporary AES key using CKM_AES_KEY_WRAP_KWP ([AES KEYWRAP] section 6.3).
- Zeros the temporary AES key and EC transport private key.
• Concatenates public key material of the transport key and output the concatenated blob. The first part is the public key material of the transport key and the second part is the wrapped target key.

The recommended format for an asymmetric target key being wrapped is as a PKCS8 PrivateKeyInfo.

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

For unwrapping, the mechanism -

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

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

• Performs ECDH operation using CKM_ECDH1_DERIVE with parameters of kdf, ulSharedDataLen and pSharedData using the private part of unwrapping EC key and the public part of the transport EC key and gets first ulAESKeyBits bits of the derived key to be the temporary AES key.

• Un-wraps the target key from the second part with the temporary AES key using CKM_AES_KEY_WRAP_KWP ([AES KEYWRAP] section 6.3).

• Zeroizes the temporary AES key.

Table 51, CKM_ECDH_AES_KEY_WRAP Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR^1</th>
<th>Digest</th>
<th>Gen. Key/ Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_ECDH_AES_KEY_WRAP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

^1SR = SignRecover, VR = VerifyRecover

Constraints on key types are summarized in the following table:

Table 52: ECDH AES Key Wrap: Allowed Key Types

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Derive</td>
<td>CKK_EC or CKK_EC_MONTGOMERY</td>
</tr>
</tbody>
</table>

2.3.21 ECDH AES KEY WRAP mechanism parameters

- CK_ECDH_AES_KEY_WRAP_PARAMS; CK_ECDH_AES_KEY_WRAP_PARAMS_PTR

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

typedef struct CK_ECDH_AES_KEY_WRAP_PARAMS {
    CK ULONG ulAESKeyBits;
}
CK_EC_KDF_TYPE  kdf;
CK_ULONG  ulSharedDataLen;
CK_BYTE_PTR  pSharedData;
}  CK_ECDH_AES_KEY_WRAP_PARAMS;

The fields of the structure have the following meanings:

- *ulAESKeyBits*  length of the temporary AES key in bits. Can be only 128, 192 or 256.
- *kdf*  key derivation function used on the shared secret value to generate AES key.
- *ulSharedDataLen*  the length in bytes of the shared info
- *pSharedData*  Some data shared between the two parties

**CK_ECDH_AES_KEY_WRAP_PARAMS_PTR** is a pointer to a **CK_ECDH_AES_KEY_WRAP_PARAMS**.

### 2.3.22 FIPS 186-4

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

- P-192, P-224, P-256, P-384, P-521
- K-163, B-163, K-233, B-233
- K-283, B-283, K-409, B-409
- K-571, B-571

### 2.4 Diffie-Hellman

Table 53, Diffie-Hellman Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Encrypt &amp; Decrypt</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>CKM_DH_PKCS_KEY_PAIR_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DH_PKCS_PARAMETER_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DH_PKCS_DERIVE</td>
<td></td>
</tr>
<tr>
<td>CKM_X9_42_DH_KEY_PAIR_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_X9_42_DH_PKCS_PARAMETER_GEN</td>
<td>✓</td>
</tr>
</tbody>
</table>
### 2.4.1 Definitions

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

**Mechanisms:**
- CKM_DH_PKCS_KEY_PAIR_GEN
- CKM_X9_42_DH_KEY_PAIR_GEN
- CKM_X9_42_DH_DERIVE
- CKM_X9_42_DH_HYBRID_DERIVE
- CKM_X9_42_MQV_DERIVE
- CKM_DH_PKCS_PARAMETER_GEN
- CKM_X9_42_DH_PARAMETER_GEN

### 2.4.2 Diffie-Hellman public key objects

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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_PRIME</td>
<td>Big integer</td>
<td>Prime $p$</td>
</tr>
<tr>
<td>CKA_BASE</td>
<td>Big integer</td>
<td>Base $g$</td>
</tr>
<tr>
<td>CKA_VALUE$^{1,4}$</td>
<td>Big integer</td>
<td>Public value $y$</td>
</tr>
</tbody>
</table>

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

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

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

```c
CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_DH;
CK_UTF8CHAR label[] = "A Diffie-Hellman public key object";
CK_BYTE prime[] = {...};
CK_BYTE base[] = {...};
CK_BYTE value[] = {...};
```
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_PRIME, prime, sizeof(prime)},
    {CKA_BASE, base, sizeof(base)},
    {CKA_VALUE, value, sizeof(value)}
};

2.4.3 X9.42 Diffie-Hellman public key objects

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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_PRIME1,3</td>
<td>Big integer</td>
<td>Prime p (≥ 1024 bits, in steps of 256 bits)</td>
</tr>
<tr>
<td>CKA_BASE1,3</td>
<td>Big integer</td>
<td>Base g</td>
</tr>
<tr>
<td>CKA_SUBPRIME1,3</td>
<td>Big integer</td>
<td>Subprime q (≥ 160 bits)</td>
</tr>
<tr>
<td>CKA_VALUE1,4</td>
<td>Big integer</td>
<td>Public value y</td>
</tr>
</tbody>
</table>

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

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

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

```c
CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_X9_42_DH;
CK_UTF8CHAR label[] = "A X9.42 Diffie-Hellman public key object";
CK_BYTE prime[] = {...};
CK_BYTE base[] = {...};
CK_BYTE subprime[] = {...};
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_PRIME, prime, sizeof(prime)},
    {CKA_BASE, base, sizeof(base)},
    {CKA_SUBPRIME, subprime, sizeof(subprime)},
    {CKA_VALUE, value, sizeof(value)}
};
```
2.4.4 Diffie-Hellman private key objects

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

Table 56, Diffie-Hellman Private Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_PRIME</td>
<td>Big integer</td>
<td>Prime p</td>
</tr>
<tr>
<td>CKA_BASE</td>
<td>Big integer</td>
<td>Base g</td>
</tr>
<tr>
<td>CKA_VALUE</td>
<td>Big integer</td>
<td>Private value x</td>
</tr>
<tr>
<td>CKA_VALUE_BITS</td>
<td>CK_ULONG</td>
<td>Length in bits of private value x</td>
</tr>
</tbody>
</table>

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

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

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

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

```c
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_DH;
CK_UTF8CHAR label[] = “A Diffie-Hellman private key object”;
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE prime[] = {...};
CK_BYTE base[] = {...};
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_SUBJECT, subject, sizeof(subject)},
    {CKA_ID, id, sizeof(id)},
    {CKA_SENSITIVE, &true, sizeof(true)},
    {CKA_DERIVE, &true, sizeof(true)},
    {CKA_PRIME, prime, sizeof(prime)},
    {CKA_BASE, base, sizeof(base)},
    {CKA_VALUE, value, sizeof(value)}
};
```

2.4.5 X9.42 Diffie-Hellman private key objects

X9.42 Diffie-Hellman private key objects (object class `CKO_PRIVATE_KEY`, key type `CKK_X9_42_DH`) hold X9.42 Diffie-Hellman private keys. The following table defines the X9.42 Diffie-Hellman private key object attributes, in addition to the common attributes defined for this object class:
Table 57, X9.42 Diffie-Hellman Private Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_PRIME</td>
<td>Big integer</td>
<td>Prime $p \geq 1024$ bits, in steps of 256 bits</td>
</tr>
<tr>
<td>CKA_BASE</td>
<td>Big integer</td>
<td>Base $g$</td>
</tr>
<tr>
<td>CKA_SUBPRIME</td>
<td>Big integer</td>
<td>Subprime $q \geq 160$ bits</td>
</tr>
<tr>
<td>CKA_VALUE</td>
<td>Big integer</td>
<td>Private value $x$</td>
</tr>
</tbody>
</table>

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

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

Note that when generating a X9.42 Diffie-Hellman private key, the X9.42 Diffie-Hellman domain parameters are not specified in the key’s template. This is because X9.42 Diffie-Hellman private keys are only generated as part of a X9.42 Diffie-Hellman key pair, and the X9.42 Diffie-Hellman domain parameters for the pair are specified in the template for the X9.42 Diffie-Hellman public key.

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

```c
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_X9_42_DH;
CK_UTF8CHAR label[] = “A X9.42 Diffie-Hellman private key object”;
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE prime[] = {...};
CK_BYTE base[] = {...};
CK_BYTE subprime[] = {...};
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_SUBJECT, subject, sizeof(subject)},
    {CKA_ID, id, sizeof(id)},
    {CKA_SENSITIVE, &true, sizeof(true)},
    {CKA_DERIVE, &true, sizeof(true)},
    {CKA_PRIME, prime, sizeof(prime)},
    {CKA_BASE, base, sizeof(base)},
    {CKA_SUBPRIME, subprime, sizeof(subprime)},
    {CKA_VALUE, value, sizeof(value)}
};
```

2.4.6 Diffie-Hellman domain parameter objects

Diffie-Hellman domain parameter objects (object class CKO_DOMAIN_PARAMETERS, key type CKK_DH) hold Diffie-Hellman domain parameters. The following table defines the Diffie-Hellman domain parameter object attributes, in addition to the common attributes defined for this object class:
Table 58, Diffie-Hellman Domain Parameter Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_PRIME&lt;sup&gt;1,4&lt;/sup&gt;</td>
<td>Big integer</td>
<td>Prime $p$ ($\geq 1024$ bits, in steps of 256 bits)</td>
</tr>
<tr>
<td>CKA_BASE&lt;sup&gt;1,4&lt;/sup&gt;</td>
<td>Big integer</td>
<td>Base $g$</td>
</tr>
<tr>
<td>CKA_PRIME_BITS&lt;sup&gt;2,3&lt;/sup&gt;</td>
<td>CK_ULONG</td>
<td>Length of the prime value.</td>
</tr>
</tbody>
</table>

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

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

```c
CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
CK_KEY_TYPE keyType = CKK_DH;
CK_UTF8CHAR label[] = "A Diffie-Hellman domain parameters object";
CK_BYTE prime[] = {...};
CK_BYTE base[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
   {CKA_CLASS, &class, sizeof(class)},
   {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   {CKA_TOKEN, &true, sizeof(true)},
   {CKA_LABEL, label, sizeof(label) - 1},
   {CKA_PRIME, prime, sizeof(prime)},
   {CKA_BASE, base, sizeof(base)},
};
```

2.4.7 X9.42 Diffie-Hellman domain parameters objects

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

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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_PRIME&lt;sup&gt;1,4&lt;/sup&gt;</td>
<td>Big integer</td>
<td>Prime $p$ ($\geq 1024$ bits, in steps of 256 bits)</td>
</tr>
<tr>
<td>CKA_BASE&lt;sup&gt;1,4&lt;/sup&gt;</td>
<td>Big integer</td>
<td>Base $g$</td>
</tr>
<tr>
<td>CKA_SUBPRIME&lt;sup&gt;1,4&lt;/sup&gt;</td>
<td>Big integer</td>
<td>Subprime $q$ ($\geq 160$ bits)</td>
</tr>
<tr>
<td>CKA_PRIME_BITS&lt;sup&gt;2,3&lt;/sup&gt;</td>
<td>CK_ULONG</td>
<td>Length of the prime value.</td>
</tr>
<tr>
<td>CKA_SUBPRIME_BITS&lt;sup&gt;2,3&lt;/sup&gt;</td>
<td>CK_ULONG</td>
<td>Length of the subprime value.</td>
</tr>
</tbody>
</table>

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

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

```c
CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
CK_KEY_TYPE keyType = CKK_X9_42_DH;
```
CK_UTF8CHAR label[] = "A X9.42 Diffie-Hellman domain parameters object";
CK_BYTE prime[] = {...};
CK_BYTE base[] = {...};
CK_BYTE subprime[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_PRIME, prime, sizeof(prime)},
    {CKA_BASE, base, sizeof(base)},
    {CKA_SUBPRIME, subprime, sizeof(subprime)}
};

2.4.8 PKCS #3 Diffie-Hellman key pair generation

The PKCS #3 Diffie-Hellman key pair generation mechanism, denoted CKM_DH_PKCS_KEY_PAIR_GEN, is a key pair generation mechanism based on Diffie-Hellman key agreement, as defined in PKCS #3. This is what PKCS #3 calls “phase I”. It does not have a parameter.

The mechanism generates Diffie-Hellman public/private key pairs with a particular prime and base, as specified in the CKA_PRIME and CKA_BASE attributes of the template for the public key. If the CKA_VALUE_BITS attribute of the private key is specified, the mechanism limits the length in bits of the private value, as described in PKCS #3.

The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, and CKA_VALUE attributes to the new public key and the CKA_CLASS, CKA_KEY_TYPE, CKA_PRIME, CKA_BASE, and CKA_VALUE (and the CKA_VALUE_BITS attribute, if it is not already provided in the template) attributes to the new private key; other attributes required by the Diffie-Hellman public and private key types must be specified in the templates.

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

2.4.9 PKCS #3 Diffie-Hellman domain parameter generation

The PKCS #3 Diffie-Hellman domain parameter generation mechanism, denoted CKM_DH_PKCS_PARAMETER_GEN, is a domain parameter generation mechanism based on Diffie-Hellman key agreement, as defined in PKCS #3.

It does not have a parameter.

The mechanism generates Diffie-Hellman domain parameters with a particular prime length in bits, as specified in the CKA_PRIME_BITS attribute of the template.

The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, CKA_PRIME, CKA_BASE, and CKA_PRIME_BITS attributes to the new object. Other attributes supported by the Diffie-Hellman domain parameter types may also be specified in the template, or else are assigned default initial values.

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

2.4.10 PKCS #3 Diffie-Hellman key derivation

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

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

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

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

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

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

### 2.4.11 X9.42 Diffie-Hellman mechanism parameters

**CK_X9_42_DH_KDF_TYPE, CK_X9_42_DH_KDF_TYPE_PTR**

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

```c
typedef CK_ULONG CK_X9_42_DH_KDF_TYPE;
```

The following table lists the defined functions.

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

<table>
<thead>
<tr>
<th>Source Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKD_NULL</td>
</tr>
<tr>
<td>CKD_SHA1_KDF_ASN1</td>
</tr>
<tr>
<td>CKD_SHA1_KDF_CONCATENATE</td>
</tr>
</tbody>
</table>

The key derivation function CKD_NULL produces a raw shared secret value without applying any key derivation function whereas the key derivation functions CKD_SHA1_KDF_ASN1 and CKD_SHA1_KDF_CONCATENATE, which are both based on SHA-1, derive keying data from the shared secret value as defined in the ANSI X9.42 standard.

CK_X9_42_DH_KDF_TYPE_PTR is a pointer to a CK_X9_42_DH_KDF_TYPE.

---

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

```c
typedef struct CK_X9_42_DH1_DERIVE_PARAMS {
    CK_X9_42_DH_KDF_TYPE kdf;
    CK_ULONG ulOtherInfoLen;
    CK_BYTE_PTR pOtherInfo;
    CK_ULONG ulPublicDataLen;
    CK_BYTE_PTR pPublicData;
} CK_X9_42_DH1_DERIVE_PARAMS;
```

The fields of the structure have the following meanings:

- `kdf`: key derivation function used on the shared secret value
- `ulOtherInfoLen`: the length in bytes of the other info
- `pOtherInfo`: some data shared between the two parties
- `ulPublicDataLen`: the length in bytes of the other party’s X9.42 Diffie-Hellman public key
- `pPublicData`: pointer to other party’s X9.42 Diffie-Hellman public key value

With the key derivation function `CKD_NULL`, `pOtherInfo` must be NULL and `ulOtherInfoLen` must be zero.
With the key derivation function `CKD_SHA1_KDF_ASN1`, `pOtherInfo` must be supplied, which contains
an octet string, specified in ASN.1 DER encoding, consisting of mandatory and optional data shared by
the two parties intending to share the shared secret. With the key derivation function
`CKD_SHA1_KDF_CONCATENATE`, an optional `pOtherInfo` may be supplied, which consists of some
data shared by the two parties intending to share the shared secret. Otherwise, `pOtherInfo` must be
NULL and `ulOtherInfoLen` must be zero.

CK_X9_42_DH1_DERIVE_PARAMS_PTR is a pointer to a CK_X9_42_DH1_DERIVE_PARAMS.

CK_X9_42_DH2_DERIVE_PARAMS, CK_X9_42_DH2_DERIVE_PARAMS_PTR

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

```c
typedef struct CK_X9_42_DH2_DERIVE_PARAMS {
    CK_X9_42_DH_KDF_TYPE kdf;
    CK_ULONG ulOtherInfoLen;
    CK_BYTE_PTR pOtherInfo;
    CK_ULONG ulPublicDataLen;
    CK_BYTE_PTR pPublicData;
    CK_ULONG ulPrivateDataLen;
    CK_OBJECT_HANDLE hPrivateData;
    CK_ULONG ulPublicDataLen2;
    CK_BYTE_PTR pPublicData2;
} CK_X9_42_DH2_DERIVE_PARAMS;
```
The fields of the structure have the following meanings:

- **kdf**: key derivation function used on the shared secret value
- **ulOtherInfoLen**: the length in bytes of the other info
- **pOtherInfo**: some data shared between the two parties
- **ulPublicDataLen**: the length in bytes of the other party’s first X9.42 Diffie-Hellman public key
- **pPublicData**: pointer to other party’s first X9.42 Diffie-Hellman public key value
- **ulPrivateDataLen**: the length in bytes of the second X9.42 Diffie-Hellman private key
- **hPrivateData**: key handle for second X9.42 Diffie-Hellman private key value
- **ulPublicDataLen2**: the length in bytes of the other party’s second X9.42 Diffie-Hellman public key
- **pPublicData2**: pointer to other party’s second X9.42 Diffie-Hellman public key value

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

With the key derivation function **CKD_SHA1_KDF_ASN1**, `pOtherInfo` must be supplied, which contains an octet string, specified in ASN.1 DER encoding, consisting of mandatory and optional data shared by the two parties intending to share the shared secret. With the key derivation function **CKD_SHA1_KDF_CONCATENATE**, an optional `pOtherInfo` may be supplied, which consists of some data shared by the two parties intending to share the shared secret. Otherwise, `pOtherInfo` must be NULL and `ulOtherInfoLen` must be zero.

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

**CK_X9_42_MQV_DERIVE_PARAMS** is a structure that provides the parameters to the **CKM_X9_42_MQV_DERIVE** key derivation mechanism, where each party contributes two key pairs. The structure is defined as follows:

```c
typedef struct CK_X9_42_MQV_DERIVE_PARAMS {
    CK_X9_42_DH_KDF_TYPE kdf;
    CK_ULONG ulOtherInfoLen;
    CK_BYTE_PTR pOtherInfo;
    CK_ULONG ulPublicDataLen;
    CK_BYTE_PTR pPublicData;
    CK_ULONG ulPrivateDataLen;
    CK_OBJECT_HANDLE hPrivateData;
    CK_ULONG ulPublicDataLen2;
    CK_OBJECT_HANDLE publicKey;
    CK_BYTE_PTR pPublicData2;
    CK_OBJECT_HANDLE hPrivateData2;
} CK_X9_42_MQV_DERIVE_PARAMS;
```
The fields of the structure have the following meanings:

- **kdf**: key derivation function used on the shared secret value
- **ulOtherInfoLen**: the length in bytes of the other info
- **pOtherInfo**: some data shared between the two parties
- **ulPublicDataLen**: the length in bytes of the other party’s first X9.42 Diffie-Hellman public key
- **pPublicData**: pointer to other party’s first X9.42 Diffie-Hellman public key value
- **ulPrivateDataLen**: the length in bytes of the second X9.42 Diffie-Hellman private key
- **hPrivateData**: key handle for second X9.42 Diffie-Hellman private key value
- **ulPublicDataLen2**: the length in bytes of the other party’s second X9.42 Diffie-Hellman public key
- **pPublicData2**: pointer to other party’s second X9.42 Diffie-Hellman public key value
- **publicKey**: Handle to the first party’s ephemeral public key

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

With the key derivation function **CKD_SHA1_KDF_ASN1**, `pOtherInfo` must be supplied, which contains an octet string, specified in ASN.1 DER encoding, consisting of mandatory and optional data shared by the two parties intending to share the shared secret. With the key derivation function **CKD_SHA1_KDF_CONCATENATE**, an optional `pOtherInfo` may be supplied, which consists of some data shared by the two parties intending to share the shared secret. Otherwise, `pOtherInfo` must be NULL and `ulOtherInfoLen` must be zero.

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

### 2.4.12 X9.42 Diffie-Hellman key pair generation

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

It does not have a parameter.

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

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

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

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

It does not have a parameter.

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

The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, CKA_PRIME, CKA_BASE, CKA_SUBPRIME, CKA_PRIME_BITS and CKA_SUBPRIME_BITS attributes to the new object. Other attributes supported by the X9.42 Diffie-Hellman domain parameter types may also be specified in the template for the domain parameters, 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 X9.42 Diffie-Hellman prime sizes, in bits.

2.4.14 X9.42 Diffie-Hellman key derivation

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

It has a parameter, a CK_X9_42_DH1_DERIVE_PARAMS structure.

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

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

- The CKA_SENSITIVE and CKA_EXTRACTABLE attributes in the template for the new key can both be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some default value.

- If the base key has its CKA_ALWAYS_SENSITIVE attribute set to CK_FALSE, then the derived key will as well. If the base key has its CKA_ALWAYS_SENSITIVE attribute set to CK_TRUE, then the derived key has its CKA_ALWAYS_SENSITIVE attribute set to the same value as its CKA_SENSITIVE attribute.

- Similarly, if the base key has its CKA_NEVER_EXTRACTABLE attribute set to CK_FALSE, then the derived key will, too. If the base key has its CKA_NEVER_EXTRACTABLE attribute set to CK_TRUE, then the derived key has its CKA_NEVER_EXTRACTABLE attribute set to the opposite value from its CKA_EXTRACTABLE attribute.

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

2.4.15 X9.42 Diffie-Hellman hybrid key derivation

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

It has a parameter, a CK_X9_42_DH2_DERIVE_PARAMS structure.
This mechanism derives a secret value, and truncates the result according to the **CKA_KEY_TYPE** attribute of the template and, if it has one and the key type supports it, the **CKA_VALUE_LEN** attribute of the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism contributes the result as the **CKA_VALUE** attribute of the new key; other attributes required by the key type must be specified in the template. Note that in order to validate this mechanism it may be required to use the **CKA_VALUE** attribute as the key of a general-length MAC mechanism (e.g. **CK_SHA_1_HMAC_GENERAL**) over some test data.

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

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

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

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

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

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

This mechanism derives a secret value, and truncates the result according to the **CKA_KEY_TYPE** attribute of the template and, if it has one and the key type supports it, the **CKA_VALUE_LEN** attribute of the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism contributes the result as the **CKA_VALUE** attribute of the new key; other attributes required by the key type must be specified in the template. Note that in order to validate this mechanism it may be required to use the **CKA_VALUE** attribute as the key of a general-length MAC mechanism (e.g. **CK_SHA_1_HMAC_GENERAL**) over some test data.

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

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

For this mechanism, the **ulMinKeySize** and **ulMaxKeySize** fields of the **CK_MECHANISM_INFO** structure specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits, for the **CKA_PRIME** attribute.
2.5 Extended Triple Diffie-Hellman (x3dh)

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

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

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Encrypt &amp; Decrypt</td>
</tr>
<tr>
<td>CKM_X3DH_INITIATE</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_X3DH_RESPOND</td>
<td></td>
</tr>
</tbody>
</table>

2.5.1 Definitions

Mechanisms:
- CKM_X3DH_INITIATE
- CKM_X3DH_RESPOND

2.5.2 Extended Triple Diffie-Hellman key objects

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

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

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

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

2.5.3 Initiating an Extended Triple Diffie-Hellman key exchange

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

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

- the initiators identity key
- the initiators ephemeral key (a fresh, one-time CKK_EC_MONTGOMERY type key)

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

```c
typedef struct CK_X3DH_INITIATE_PARAMS {
    CK_X3DH_KDF_TYPE kdf;
    CK_OBJECT_HANDLE pPeer_identity;
};
```
2.5.4 Responding to an Extended Triple Diffie-Hellman key exchange

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

```c
typedef struct CK_X3DH_RESPOND_PARAMS {
    CK_X3DH_KDF_TYPE kdf;
    CK_BYTE_PTR pIdentity_id;
    CK_BYTE_PTR pPrekey_id;
    CK_BYTE_PTR pOnetime_id;
    CK_OBJECT_HANDLE pInitiator_identity;
    CK_BYTE_PTR pInitiator_ephemeral;
} CK_X3DH_RESPOND_PARAMS;
```

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>kdf</td>
<td>CK_X3DH_KDF_TYPE</td>
<td>Key derivation function</td>
</tr>
<tr>
<td>pIdentity_id</td>
<td>Byte array</td>
<td>Peers public Identity key identifier (from the prekey-bundle)</td>
</tr>
<tr>
<td>pPrekey_id</td>
<td>Byte array</td>
<td>Peers public prekey identifier (from the prekey-bundle)</td>
</tr>
<tr>
<td>pOnetime_id</td>
<td>Byte array</td>
<td>Optional one-time public prekey of peer (from the prekey-bundle)</td>
</tr>
<tr>
<td>pInitiator_identity</td>
<td>Key handle</td>
<td>Initiators Identity key</td>
</tr>
<tr>
<td>pInitiator_ephemeral</td>
<td>Byte array</td>
<td>Initiators ephemeral key</td>
</tr>
</tbody>
</table>
Where the *id fields are identifiers marking which key has been used from the prekey-bundle, these identifiers could be the keys themselves.

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

1. The CKA_SENSITIVE and CKA_EXTRACTABLE attributes in the template for the new key can both be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some default value.

2. If the base key has its CKA_ALWAYS_SENSITIVE attribute set to CK_FALSE, then the derived key will as well. If the base key has its CKA_ALWAYS_SENSITIVE attribute set to CK_TRUE, then the derived key has its CKA_ALWAYS_SENSITIVE attribute set to the same value as its CKA_SENSITIVE attribute.

3. Similarly, if the base key has its CKA_NEVER_EXTRACTABLE attribute set to CK_FALSE, then the derived key will, too. If the base key has its CKA_NEVER_EXTRACTABLE attribute set to CK_TRUE, then the derived key has its CKA_NEVER_EXTRACTABLE attribute set to the opposite value from its CKA_EXTRACTABLE attribute.

2.5.5 Extended Triple Diffie-Hellman parameters

- CK_X3DH_KDF_TYPE, CK_X3DH_KDF_TYPE_PTR

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

```c
typedef CK_ULONG CK_X3DH_KDF_TYPE;
```

The following table lists the defined functions.

<table>
<thead>
<tr>
<th>Source Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKD_NULL</td>
</tr>
<tr>
<td>CKD_BLAKE2B_256_KDF</td>
</tr>
<tr>
<td>CKD_BLAKE2B_512_KDF</td>
</tr>
<tr>
<td>CKD_SHA3_256_KDF</td>
</tr>
<tr>
<td>CKD_SHA256_KDF</td>
</tr>
<tr>
<td>CKD_SHA3_512_KDF</td>
</tr>
<tr>
<td>CKD_SHA512_KDF</td>
</tr>
</tbody>
</table>

2.6 Double Ratchet

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

Table 65, Double Ratchet Mechanisms vs. Functions

---

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.
### 2.6.1 Definitions

This section defines the key type “CKK_X2RATCHET” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

#### Mechanisms:

- CKM_X2RATCHET_INITIALIZE
- CKM_X2RATCHET_RESPOND
- CKM_X2RATCHET_ENCRYPT
- CKM_X2RATCHET_DECRYPT

### 2.6.2 Double Ratchet secret key objects

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

#### Table 66, Double Ratchet Secret Key Object Attributes

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_X2RATCHET_INITIALIZE</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_X2RATCHET_RESPOND</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_X2RATCHET_ENCRYPT</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_X2RATCHET_DECRYPT</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Attribute</td>
<td>Data type</td>
<td>Meaning</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------</td>
<td>----------------</td>
<td>------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKA_X2RATCHET_RK</td>
<td>Byte array</td>
<td>Root key</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKA_X2RATCHET_HKS</td>
<td>Byte array</td>
<td>Sender Header key</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKA_X2RATCHET_HKR</td>
<td>Byte array</td>
<td>Receiver Header key</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKA_X2RATCHET_NHKR</td>
<td>Byte array</td>
<td>Next Sender Header Key</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKA_X2RATCHET_NHR</td>
<td>Byte array</td>
<td>Next Receiver Header Key</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKA_X2RATCHET_CKS</td>
<td>Byte array</td>
<td>Sender Chain key</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKA_X2RATCHET_CKR</td>
<td>Byte array</td>
<td>Receiver Chain key</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKA_X2RATCHET_DHS</td>
<td>Byte array</td>
<td>Sender DH secret key</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKA_X2RATCHET_DHP</td>
<td>Byte array</td>
<td>Sender DH public key</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKA_X2RATCHET_DHR</td>
<td>Byte array</td>
<td>Receiver DH public key</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKA_X2RATCHET_NS</td>
<td>ULONG</td>
<td>Message number send</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKA_X2RATCHET_NR</td>
<td>ULONG</td>
<td>Message number receive</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKA_X2RATCHET_PNS</td>
<td>ULONG</td>
<td>Previous message number send</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKA_X2RATCHET_BOBS1STMSG</td>
<td>BOOL</td>
<td>Is this bob and has he ever sent a message?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKA_X2RATCHET_ISALICE</td>
<td>BOOL</td>
<td>Is this Alice?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKA_X2RATCHET_BAGSIZE</td>
<td>ULONG</td>
<td>How many out-of-order keys do we store</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKA_X2RATCHET_BAG</td>
<td>Byte array</td>
<td>Out-of-order keys</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 2.6.3 Double Ratchet key derivation

The Double Ratchet key derivation mechanisms depend on who is the initiating party, and who the receiving, denoted `CKM_X2RATCHET_INITIALIZE` and `CKM_X2RATCHET_RESPOND`, are the key derivation mechanisms for the Double Ratchet. Usually the keys are derived from a shared secret by executing a X3DH key exchange.

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

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

- **CK_X2RATCHET_INITIALIZE_PARAMS**;
  - **CK_X2RATCHET_INITIALIZE_PARAMS_PTR**

`CK_X2RATCHET_INITIALIZE_PARAMS` provides the parameters to the `CKM_X2RATCHET_INITIALIZE` mechanism. It is defined as follows:

```c
typedef struct CK_X2RATCHET_INITIALIZE_PARAMS {
    CK_BYTE_PTR sk;
    CK_OBJECT_HANDLE peer_public_prekey;
    CK_OBJECT_HANDLE peer_public_identity;
    CK_OBJECT_HANDLE own_public_identity;
    CK_BBOOL bEncryptedHeader;
    CK_UULONG eCurve;
    CK_MECHANISM_TYPE aeadMechanism;
    CK_X2RATCHET_KDF_TYPE kdfMechanism;
} CK_X2RATCHET_INITIALIZE_PARAMS;
```
The fields of the structure have the following meanings:

- **sk**: the shared secret with peer (derived using X3DH)
- **peers_public_prekey**: Peers public prekey which the Initiator used in the X3DH
- **peers_public_identity**: Peers public identity which the Initiator used in the X3DH
- **own_public_identity**: Initiators public identity as used in the X3DH
- **bEncryptedHeader**: whether the headers are encrypted
- **eCurve**: 255 for curve 25519 or 448 for curve 448
- **aeadMechanism**: a mechanism supporting AEAD encryption, e.g. CKM_XCHACHA20
- **kdfMechanism**: a Key Derivation Mechanism, such as CKD_BLAKE2B_512_KDF

- **CK_X2RATCHET_RESPOND_PARAMS**;
  - **CKM_X2RATCHET_RESPOND_PARAMS_PTR**

CKM_X2RATCHET_RESPOND_PARAMS provides the parameters to the CKM_X2RATCHET_RESPOND mechanism. It is defined as follows:

```c
typedef struct CK_X2RATCHET_RESPOND_PARAMS {
    CK_BYTE_PTR sk;
    CK_OBJECT_HANDLE own_prekey;
    CK_OBJECT_HANDLE initiator_identity;
    CK_OBJECT_HANDLE own_public_identity;
    CK_BBOOL bEncryptedHeader;
    CK_ULONG eCurve;
    CK_MECHANISM_TYPE aeadMechanism;
    CK_X2RATCHET_KDF_TYPE kdfMechanism;
} CK_X2RATCHET_RESPOND_PARAMS;
```

The fields of the structure have the following meanings:

- **sk**: shared secret with the Initiator
- **own_prekey**: Own Prekey pair that the Initiator used
- **initiator_identity**: Initiators public identity key used
- **own_public_identity**: as used in the prekey bundle by the initiator in the X3DH
- **bEncryptedHeader**: whether the headers are encrypted
- **eCurve**: 255 for curve 25519 or 448 for curve 448
aeadMechanism a mechanism supporting AEAD encryption, e.g.
CKM_XCHACHA20

kdfMechanism a Key Derivation Mechanism, such as
CKD_BLAKE2B_512_KDF

2.6.4 Double Ratchet Encryption mechanism

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

2.6.5 Double Ratchet parameters

- CK_X2RATCHET_KDF_TYPE, CK_X2RATCHET_KDF_TYPE_PTR

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

typedef CK_ULONG CK_X2RATCHET_KDF_TYPE;

The following table lists the defined functions.

Table 67, X2RATCHET: Key Derivation Functions

<table>
<thead>
<tr>
<th>Source Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKD_NULL</td>
</tr>
<tr>
<td>CKD_BLAKE2B_256_KDF</td>
</tr>
<tr>
<td>CKD_BLAKE2B_512_KDF</td>
</tr>
<tr>
<td>CKD_SHA3_256_KDF</td>
</tr>
<tr>
<td>CKD_SHA256_KDF</td>
</tr>
<tr>
<td>CKD_SHA3_512_KDF</td>
</tr>
<tr>
<td>CKD_SHA512_KDF</td>
</tr>
</tbody>
</table>

2.7 Wrapping/unwrapping private keys

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

rsaEncryption OBJECT IDENTIFIER ::= { pkcs-1 1 }

dhKeyAgreement OBJECT IDENTIFIER ::= { pkcs-3 1 }

dhpublicnumber OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) ansi-x942(10046) number-type(2) 1 }

id-ecPublicKey OBJECT IDENTIFIER ::= { iso(1) member-body(2)
id-dsa OBJECT IDENTIFIER ::= {
  iso(1) member-body(2) us(840) x9-57(10040) x9cm(4) 1 }

where

pkcs-1 OBJECT IDENTIFIER ::= {
  iso(1) member-body(2) US(840) rsadsi(113549) pkcs(1) 1 }

pkcs-3 OBJECT IDENTIFIER ::= {
  iso(1) member-body(2) US(840) rsadsi(113549) pkcs(1) 3 }

These parameters for the algorithm identifiers have the following types, respectively:

NULL

DHParameter ::= SEQUENCE {
  prime INTEGER, -- p
  base INTEGER, -- g
  privateValueLength INTEGER OPTIONAL
}

DomainParameters ::= SEQUENCE {
  prime INTEGER, -- p
  base INTEGER, -- g
  subprime INTEGER, -- q
  cofactor INTEGER OPTIONAL, -- j
  validationParms ValidationParms OPTIONAL
}

ValidationParms ::= SEQUENCE {
  Seed BIT STRING, -- seed
  PGenCounter INTEGER -- parameter verification
}

Parameters ::= CHOICE {
  ecParameters ECParameters,
  namedCurve CURVES.&id({CurveNames}),
  implicitlyCA NULL
}

Dss-Parms ::= SEQUENCE {
  p INTEGER,
  q INTEGER,
  g INTEGER
}
For the X9.42 Diffie-Hellman domain parameters, the `cofactor` and the `validationParms` optional fields should not be used when wrapping or unwrapping X9.42 Diffie-Hellman private keys since their values are not stored within the token.

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

Within the `PrivateKeyInfo` type:

- RSA private keys are BER-encoded according to PKCS #1's `RSAPrivateKey` ASN.1 type. This type requires values to be present for all the attributes specific to Cryptoki's RSA private key objects. In other words, if a Cryptoki library does not have values for an RSA private key's `CKA_MODULUS`, `CKA_PUBLIC_EXPONENT`, `CKA_PRIVATE_EXPONENT`, `CKA_PRIME_1`, `CKA_PRIME_2`, `CKA_EXPONENT_1`, `CKA_EXPONENT2`, and `CKA_COEFFICIENT` values, it must not create an `RSAPrivateKey` BER-encoding of the key, and so it must not prepare it for wrapping.

- Diffie-Hellman private keys are represented as BER-encoded ASN.1 type `INTEGER`.

- X9.42 Diffie-Hellman private keys are represented as BER-encoded ASN.1 type `INTEGER`.

- EC (also related with ECDSA) private keys are BER-encoded according to SECG SEC 1 ECPrivateKey ASN.1 type:

  ```
  ECPrivateKey ::= SEQUENCE {
    Version           INTEGER { ecPrivkeyVer1(1) } (ecPrivkeyVer1),
    privateKey        OCTET STRING,
    parameters [0]    Parameters OPTIONAL,
    publicKey [1]    BIT STRING OPTIONAL
  }
  ```

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

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

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

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

Earlier drafts of PKCS #11 Version 2.0 and Version 2.0.1 used the object identifier

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

with associated parameters

```
DSAParameters ::= SEQUENCE {
  prime1 INTEGER, -- modulus p
  prime2 INTEGER, -- modulus q
```
for wrapping DSA private keys. Note that although the two structures for holding DSA domain parameters appear identical when instances of them are encoded, the two corresponding object identifiers are different.

### 2.8 Generic secret key

#### 2.8.1 Definitions

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

Mechanisms:

- CKM_GENERIC_SECRET_KEY_GEN

#### 2.8.2 Generic secret key objects

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

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

#### Table 69, Generic Secret Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE⁴,⁶,⁷</td>
<td>Byte array</td>
<td>Key value (arbitrary length)</td>
</tr>
<tr>
<td>CKA_VALUE_LEN²,³</td>
<td>CK_ULONG</td>
<td>Length in bytes of key value</td>
</tr>
</tbody>
</table>

- Refer to [PKCS11-Base] 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.

2.8.3 Generic secret key generation

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

It does not have a parameter.

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

CK_OBJECT_CLASS = CKO_SECRET_KEY;
CK_KEY_TYPE = CKK_GENERIC_SECRET;

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

2.9 HMAC mechanisms

Refer to RFC2104 and FIPS 198 for HMAC algorithm description. The HMAC secret key shall correspond
to the PKCS11 generic secret key type or the mechanism specific key types (see mechanism definition).
Such keys, for use with HMAC operations can be created using C_CreateObject or C_GenerateKey.
The RFC also specifies test vectors for the various hash function based HMAC mechanisms described in
the respective hash mechanism descriptions. The RFC should be consulted to obtain these test vectors.

2.9.1 General block cipher mechanism parameters

• CK_MAC_GENERAL_PARAMS; CK_MAC_GENERAL_PARAMS_PTR

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

typedef CK_ULONG CK_MAC_GENERAL_PARAMS;

CK_MAC_GENERAL_PARAMS_PTR is a pointer to a CK_MAC_GENERAL_PARAMS.

2.10 AES

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

Table 70, AES Mechanisms vs. Functions
### Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_AES_KEY_GEN</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_AES_ECB</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_AES_CBC</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_AES_CBC_PAD</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_AES_MAC_GENERAL</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_AES_MAC</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_AES_OFB</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_AES_CFB64</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_AES_CFB8</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_AES_CFB128</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_AES_CFB1</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_AES_XCBC_MAC</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_AES_XCBC_MAC_96</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 2.10.1 Definitions

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

Mechanisms:

- CKM_AES_KEY_GEN
- CKM_AES_ECB
- CKM_AES_CBC
- CKM_AES_MAC
- CKM_AES_MAC_GENERAL
- CKM_AES_CBC_PAD
- CKM_AES_OFB
- CKM_AES_CFB64
- CKM_AES_CFB8
- CKM_AES_CFB128
- CKM_AES_CFB1
- CKM_AES_XCBC_MAC
- CKM_AES_XCBC_MAC_96

#### 2.10.2 AES secret key objects

AES secret key objects (object class CKO_SECRET_KEY, key type CKK_AES) hold AES keys. The following table defines the AES secret key object attributes, in addition to the common attributes defined for this object class:
Table 71, AES Secret Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE&lt;sup&gt;1,4,6,7&lt;/sup&gt;</td>
<td>Byte array</td>
<td>Key value (16, 24, or 32 bytes)</td>
</tr>
<tr>
<td>CKA_VALUE_LEN&lt;sup&gt;2,3,6&lt;/sup&gt;</td>
<td>CK_ULONG</td>
<td>Length in bytes of key value</td>
</tr>
</tbody>
</table>

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

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

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_AES;
CK_UTF8CHAR label[] = "An AES secret key object";
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_ENCRYPT, &true, sizeof(true)},
    {CKA_VALUE, value, sizeof(value)}
};
```

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.

2.10.3 AES key generation

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

It does not have a parameter.

The mechanism generates AES 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 AES 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 key sizes, in bytes.

2.10.4 AES-ECB

AES-ECB, denoted **CKM_AES_ECB**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on NIST Advanced Encryption Standard 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...
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>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>AES</td>
<td>multiple of block size</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>AES</td>
<td>multiple of block size</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>AES</td>
<td>any</td>
<td>input length rounded up to multiple of block size</td>
<td></td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>AES</td>
<td>multiple of block size</td>
<td>determined by type of key being unwrapped or CKA_VALUE_LEN</td>
<td></td>
</tr>
</tbody>
</table>

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

2.10.5 AES-CBC

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

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

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

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

Constraints on key types and the length of data are summarized in the following table:
Table 73, AES-CBC: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>AES</td>
<td>multiple of block size</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>AES</td>
<td>multiple of block size</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>AES</td>
<td>any</td>
<td>input length rounded up to</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>multiple of the block size</td>
<td></td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>AES</td>
<td>multiple of block size</td>
<td>determined by type of key</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>being unwrapped or</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CKA_VALUE_LEN</td>
<td></td>
</tr>
</tbody>
</table>

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

2.10.6 AES-CBC with PKCS padding

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

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

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

In addition to being able to wrap and unwrap secret keys, this mechanism can wrap and unwrap RSA, Diffie-Hellman, X9.42 Diffie-Hellman, EC (also related to ECDSA) and DSA private keys (see Section 2.7 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.

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

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>AES</td>
<td>any</td>
<td>input length rounded up to</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>multiple of the block size</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>AES</td>
<td>multiple of block size</td>
<td>between 1 and block size</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bytes shorter than input</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>AES</td>
<td>any</td>
<td>input length rounded up to</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>multiple of the block size</td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>AES</td>
<td>multiple of block size</td>
<td>between 1 and block length</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bytes shorter than input</td>
</tr>
</tbody>
</table>

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

2.10.7 AES-OFB

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

It has a parameter, an initialization vector for this mode. The initialization vector has the same length as the block size.
Constraints on key types and the length of data are summarized in the following table:

Table 75, AES-OFB: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>AES</td>
<td>any</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>AES</td>
<td>any</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
</tbody>
</table>

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

2.10.8 AES-CFB

Cipher AES has a cipher feedback mode, AES-CFB, denoted CKM_AES_CFB8, CKM_AES_CFB64, and CKM_AES_CFB128. It is a mechanism for single and multiple-part encryption and decryption with AES.

AES-OFB mode is described [NIST sp800-38a].

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

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

Table 76, AES-CFB: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>AES</td>
<td>any</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>AES</td>
<td>any</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
</tbody>
</table>

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

2.10.9 General-length AES-MAC

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

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

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

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

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>AES</td>
<td>any</td>
<td>1-block size, as specified in parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>AES</td>
<td>any</td>
<td>1-block size, as specified in parameters</td>
</tr>
</tbody>
</table>

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

2.10.10 AES-MAC

AES-MAC, denoted by CKM_AES_MAC, is a special case of the general-length AES-MAC mechanism.

AES-MAC always produces and verifies MACs that are half the block size in length.

It does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:
Table 78, AES-MAC: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>AES</td>
<td>Any</td>
<td>½ block size (8 bytes)</td>
</tr>
<tr>
<td>C_Verify</td>
<td>AES</td>
<td>Any</td>
<td>½ block size (8 bytes)</td>
</tr>
</tbody>
</table>

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

2.10.11 AES-XCBC-MAC

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

It does not have a parameter.

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

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>AES</td>
<td>Any</td>
<td>16 bytes</td>
</tr>
<tr>
<td>C_Verify</td>
<td>AES</td>
<td>Any</td>
<td>16 bytes</td>
</tr>
</tbody>
</table>

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

2.10.12 AES-XCBC-MAC-96

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

It does not have a parameter.

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

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>AES</td>
<td>Any</td>
<td>12 bytes</td>
</tr>
<tr>
<td>C_Verify</td>
<td>AES</td>
<td>Any</td>
<td>12 bytes</td>
</tr>
</tbody>
</table>

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

2.11 AES with Counter

Table 81, AES with Counter Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Encrypt/Decrypt</td>
</tr>
<tr>
<td>CKM_AES_CTR</td>
<td>✓</td>
</tr>
</tbody>
</table>

2.11.1 Definitions

Mechanisms:

CKM_AES_CTR
2.11.2 AES with Counter mechanism parameters

*CK_AES_CTR_PARAMS; CK_AES_CTR_PARAMS_PTR*

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

```c
typedef struct CK_AES_CTR_PARAMS {
    CK_ULONG ulCounterBits;
    CK_BYTE cb[16];
} CK_AES_CTR_PARAMS;
```

ulCounterBits specifies the number of bits in the counter block (cb) that shall be incremented. This number shall be such that 0 < ulCounterBits <= 128. For any values outside this range the mechanism shall return CKR_MECHANISM_PARAM_INVALID.

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

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

```
0                   1
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
```

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

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

2.11.3 AES with Counter Encryption / Decryption

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

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

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

2.12 AES CBC with Cipher Text Stealing CTS

Ref [NIST AES CTS]
This mode allows unpadded data that has length that is not a multiple of the block size to be encrypted to the same length of cipher text.

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

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR¹</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_AES_CTS</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

2.12.1 Definitions

Mechanisms:

- CKM_AES_CTS

2.12.2 AES CTS mechanism parameters

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

Table 83, AES-CTS: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>AES</td>
<td>Any, ≥ block size (16 bytes)</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>AES</td>
<td>any, ≥ block size (16 bytes)</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
</tbody>
</table>

2.13 Additional AES Mechanisms

Table 84, Additional AES Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR¹</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_AES_GCM</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_AES_CCM</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_AES_GMAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

2.13.1 Definitions

Mechanisms:

- CKM_AES_GCM
- CKM_AES_CCM
- CKM_AES_GMAC

Generator Functions:

- CKG_NO_GENERATE
2.13.2 AES-GCM Authenticated Encryption / Decryption

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

Encrypt:

- Set the IV length \( ulIvLen \) in the parameter block.
- Set the IV data \( plv \) in the parameter block.
- Set the AAD data \( pAAD \) and size \( ulAADLen \) in the parameter block. \( pAAD \) may be NULL if \( ulAADLen \) is 0.
- Set the tag length \( ulTagBits \) in the parameter block.
- Call C_EncryptInit() for CKM_AES_GCM mechanism with parameters and key \( K \).
- Call C_Encrypt(), or C_EncryptUpdate()\(^*\) C_EncryptFinal(), for the plaintext obtaining ciphertext and authentication tag output.

Decrypt:

- Set the IV length \( ullvLen \) in the parameter block.
- Set the IV data \( plv \) in the parameter block.
- Set the AAD data \( pAAD \) and size \( ulAADLen \) in the parameter block. \( pAAD \) may be NULL if \( ulAADLen \) is 0.
- Set the tag length \( ulTagBits \) in the parameter block.
- Call C_DecryptInit() for CKM_AES_GCM mechanism with parameters and key \( K \).
- Call C_Decrypt(), or C_DecryptUpdate()\(^*\) C_DecryptFinal(), for the ciphertext, including the appended tag, obtaining plaintext output. Note: since CKM_AES_GCM is an AEAD cipher, no data should be returned until C_Decrypt() or C_DecryptFinal() is called.

MessageEncrypt:

- Set the IV length \( ullvLen \) in the parameter block.
- Set \( plv \) to hold the IV data returned from C_EncryptMessage() and C_EncryptMessageBegin(). If \( ullvFixedBits \) is not zero, then the most significant bits of \( plv \) contain the fixed IV. If \( ivGenerator \) is set to CKG_NO_GENERATE, \( plv \) is an input parameter with the full IV.
- Set the \( ullvFixedBits \) and \( ivGenerator \) fields in the parameter block.
- Set the tag length \( ulTagBits \) in the parameter block.
- Set \( pTag \) to hold the tag data returned from C_EncryptMessage() or the final C_EncryptMessageNext().
- Call C_MessageEncryptInit() for CKM_AES_GCM mechanism key \( K \).

---

\( ^* \) indicates 0 or more calls may be made as required
Call C_EncryptMessage(), or C_EncryptMessageBegin() followed by C_EncryptMessageNext()*. The mechanism parameter is passed to all three of these functions.

- Call C_MessageEncryptFinal() to close the message encryption.

**MessageDecrypt:**

- Set the IV length *ulIvLen* in the parameter block.
- Set the IV data *pIv* in the parameter block.
- The *ulIvFixedBits* and *ivGenerator* fields are ignored.
- Set the tag length *ulTagBits* in the parameter block.
- Set the tag data *pTag* in the parameter block before C_DecryptMessage() or the final C_DecryptMessageNext().
- Call C_MessageDecryptInit() for CKM_AES_GCM mechanism key *K*.
- Call C_DecryptMessage(), or C_DecryptMessageBegin followed by C_DecryptMessageNext()*. The mechanism parameter is passed to all three of these functions.
- Call C_MessageDecryptFinal() to close the message decryption.

In *pIv* the least significant bit of the initialization vector is the rightmost bit. *ulIvLen* is the length of the initialization vector in bytes.

On MessageEncrypt, the meaning of *ivGenerator* is as follows: CKG_NO_GENERATE means the IV is passed in on MessageEncrypt and no internal IV generation is done. CKG_GENERATE means that the non-fixed portion of the IV is generated by the module internally. The generation method is not defined.

CKG_GENERATE_COUNTER means that the non-fixed portion of the IV is generated by the module internally by use of an incrementing counter. CKG_GENERATE_RANDOM means that the non-fixed portion of the IV is generated by the module internally using a PRNG. In any case the entire IV, including the fixed portion, is returned in *pIV*.

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

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

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

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

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

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

**Encrypt:**

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

---

5 *"* indicates 0 or more calls may be made as required
6 "*" indicates 0 or more calls may be made as required
• Set the nonce length `ulNonceLen` and the nonce data `pNonce` in the parameter block.
• Set the AAD data `pAAD` and size `ulAADLen` in the parameter block. `pAAD` may be NULL if `ulAADLen` is 0.
• Set the MAC length `ulMACLen` in the parameter block.
• Call `C_EncryptInit()` for `CKM_AES_CCM` mechanism with parameters and key `K`.
• Call `C_Encrypt()`, `C_EncryptUpdate()`, or `C_EncryptFinal()`, for the plaintext obtaining the final ciphertext output and the MAC. The total length of data processed must be `ulDataLen`. The output length will be `ulDataLen + ulMACLen`.

Decrypt:
• Set the message/data length `ulDataLen` in the parameter block. This length must not include the length of the MAC that is appended to the cipher text.
• Set the nonce length `ulNonceLen` and the nonce data `pNonce` in the parameter block.
• Set the AAD data `pAAD` and size `ulAADLen` in the parameter block. `pAAD` may be NULL if `ulAADLen` is 0.
• Set the MAC length `ulMACLen` in the parameter block.
• Call `C_DecryptInit()` for `CKM_AES_CCM` mechanism with parameters and key `K`.
• Call `C_Decrypt()`, `C_DecryptUpdate()`, or `C_DecryptFinal()`, for the ciphertext, including the appended MAC, obtaining plaintext output. The total length of data processed must be `ulDataLen + ulMACLen`. Note: since `CKM_AES_CCM` is an AEAD cipher, no data should be returned until `C_Decrypt()` or `C_DecryptFinal()`.

MessageEncrypt:
• Set the message/data length `ulDataLen` in the parameter block.
• Set the nonce length `ulNonceLen`.
• Set `pNonce` to hold the nonce data returned from `C_EncryptMessage()` and `C_EncryptMessageBegin()`. If `ulNonceFixedBits` is not zero, then the most significant bits of `pNonce` contain the fixed nonce. If `nonceGenerator` is set to CKG_NO_GENERATE, `pNonce` is an input parameter with the full nonce.
• Set the `ulNonceFixedBits` and `nonceGenerator` fields in the parameter block.
• Set the MAC length `ulMACLen` in the parameter block.
• Set `pMAC` to hold the MAC data returned from `C_EncryptMessage()` or the final `C_EncryptMessageNext()`.
• Call `C_MessageEncryptInit()` for `CKM_AES_CCM` mechanism key `K`.
• Call `C_EncryptMessage()`, or `C_EncryptMessageBegin()` followed by `C_EncryptMessageNext()`*: The mechanism parameter is passed to all three functions.
• Call `C_MessageEncryptFinal()` to close the message encryption.
• The MAC is returned in `pMac` of the `CK_CCM_MESSAGE_PARAMS` structure.

MessageDecrypt:
• Set the message/data length `ulDataLen` in the parameter block.

---

7 **"** indicates 0 or more calls may be made as required
• Set the nonce length \textit{ulNonceLen} and the nonce data \textit{pNonce} in the parameter block.

• The \textit{ulNonceFixedBits} and \textit{nonceGenerator} fields in the parameter block are ignored.

• Set the MAC length \textit{ulMACLen} in the parameter block.

• Set the MAC data \textit{pMAC} in the parameter block before \texttt{C_DecryptMessage()} or the final \texttt{C_DecryptMessageNext()}.

• Call \texttt{C_MessageDecryptInit()} for \textbf{CKM_AES_CCM} mechanism key \(K\).

• Call \texttt{C_DecryptMessage()}, or \texttt{C_DecryptMessageBegin()} followed by \texttt{C_DecryptMessageNext()}{*}. The mechanism parameter is passed to all three functions.

• Call \texttt{C_MessageDecryptFinal()} to close the message decryption.


\textit{pNonce} the least significant bit of the nonce is the rightmost bit. \textit{ulNonceLen} is the length of the nonce in bytes.

On \texttt{MessageEncrypt}, the meaning of \textit{nonceGenerator} is as follows: \texttt{CKG\_NO\_GENERATE} means the nonce is passed in on \texttt{MessageEncrypt} and no internal MAC generation is done. \texttt{CKG\_GENERATE} means that the non-fixed portion of the nonce is generated by the module internally. The generation method is not defined. \texttt{CKG\_GENERATE\_COUNTER} means that the non-fixed portion of the nonce is generated by the module internally by use of an incrementing counter. \texttt{CKG\_GENERATE\_RANDOM} means that the non-fixed portion of the nonce is generated by the module internally using a PRNG. In any case the entire nonce, including the fixed portion, is returned in \textit{pNonce}.

Modules must implement \texttt{CKG\_GENERATE}. Modules may also reject \textit{ulNonceFixedBits} values which are too large. Zero is always an acceptable value for \textit{ulNonceFixedBits}.

In \texttt{Encrypt} and \texttt{Decrypt} the MAC is appended to the cipher text and the least significant byte of the MAC is the rightmost byte and the MAC bytes are the rightmost \textit{ulMACLen} bytes. In \texttt{MessageEncrypt} the MAC is returned in the \textit{pMAC} field of \texttt{CK\_CCM\_MESSAGE\_PARAMS}. In \texttt{MessageDecrypt} the MAC is provided by the \textit{pMAC} field of \texttt{CK\_CCM\_MESSAGE\_PARAMS}.

The key type for \(K\) must be compatible with \textbf{CKM\_AES\_ECB} and the \texttt{C\_EncryptInit()}/\texttt{C\_DecryptInit()}/\texttt{C\_MessageEncryptInit()}/\texttt{C\_MessageDecryptInit()} calls shall behave, with respect to \(K\), as if they were called directly with \textbf{CKM\_AES\_ECB}, \(K\) and \texttt{NULL} parameters.

\subsection{2.13.4 AES-GMAC}

\textbf{AES-GMAC}, denoted \textbf{CKM\_AES\_GMAC}, is a mechanism for single and multiple-part signatures and verification. It is described in NIST Special Publication 800-38D [GMAC]. GMAC is a special case of GCM that authenticates only the Additional Authenticated Data (AAD) part of the GCM mechanism parameters. When GMAC is used with \texttt{C\_Sign} or \texttt{C\_Verify}, \texttt{pData} points to the AAD. GMAC does not use plaintext or ciphertext.

The signature produced by GMAC, also referred to as a Tag, the tag's length is determined by the \texttt{CK\_GMAC\_PARAMS} field \textit{ulTagBits}.

The IV length is determined by the \texttt{CK\_GMAC\_PARAMS} field \textit{ulIVLen}.

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

\begin{center}
\begin{tabular}{|c|c|c|}
\hline
Key Type & \texttt{ckmgmacInit()} & \texttt{ckmgmacFinal()} \\
\hline
\texttt{CKM\_AES\_GMAC} & \checkmark & \checkmark \\
\texttt{CKM\_AES\_GMAC} & \checkmark & \checkmark \\
\end{tabular}
\end{center}
Table 85, AES-GMAC: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>CKK_AES</td>
<td>&lt; 2^64</td>
<td>Depends on param’s ulTagBits</td>
</tr>
<tr>
<td>C_Verify</td>
<td>CKK_AES</td>
<td>&lt; 2^64</td>
<td>Depends on param’s ulTagBits</td>
</tr>
</tbody>
</table>

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

2.13.5 AES GCM and CCM Mechanism parameters

♦ CK_GENERATOR_FUNCTION

Functions to generate unique IVs and nonces.

```c
typedef CK_ULONG CK_GENERATOR_FUNCTION;
```

♦ CK_GCM_PARAMS; CK_GCM_PARAMS_PTR

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

```c
typedef struct CK_GCM_PARAMS {
    CK_BYTE_PTR pIv;
    CK_ULONG ulIvLen;
    CK_BYTE_PTR pAAD;
    CK_ULONG ulAADLen;
    CK_ULONG ulTagBits;
} CK_GCM_PARAMS;
```

The fields of the structure have the following meanings:

- **pIv** pointer to initialization vector
- **ulIvLen** 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.
- **pAAD** pointer to additional authentication data. This data is authenticated but not encrypted.
- **ulAADLen** length of pAAD in bytes. The length of the AAD can be any number between 0 and \((2^{32}) – 1\).
- **ulTagBits** length of authentication tag (output following cipher text) in bits. Can be any value between 0 and 128.

CK_GCM_PARAMS_PTR is a pointer to a CK_GCM_PARAMS.

♦ CK_GCM_MESSAGE_PARAMS; CK_GCM_MESSAGE_PARAMS_PTR

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:
typedef struct CK_GCM_MESSAGE_PARAMS {
    CK_BYTE_PTR pIv;
    CK_ULONG ulIvLen;
    CK_ULONG ulIvFixedBits;
    CK_GENERATOR_FUNCTION ivGenerator;
    CK_BYTE_PTR pTag;
    CK_ULONG ulTagBits;
} CK_GCM_MESSAGE_PARAMS;

The fields of the structure have the following meanings:

- **pIv** pointer to initialization vector
- **ulIvLen** 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.
- **ulIvFixedBits** number of bits of the original IV to preserve when generating an new IV. These bits are counted from the Most significant bits (to the right).
- **ivGenerator** Function used to generate a new IV. Each IV must be unique for a given session.
- **pTag** location of the authentication tag which is returned on MessageEncrypt, and provided on MessageDecrypt.
- **ulTagBits** length of authentication tag in bits. Can be any value between 0 and 128.

CK_GCM_MESSAGE_PARAMS_PTR is a pointer to a CK_GCM_MESSAGE_PARAMS.

**CK_CCM_PARAMS; CK_CCM_PARAMS_PTR**

CK_CCM_PARAMS is a structure that provides the parameters to the CKM_AES_CCM mechanism when used for Encrypt or Decrypt. It is defined as follows:

typedef struct CK_CCM_PARAMS {
    CK_ULONG ulDataLen; /*plaintext or ciphertext*/
    CK_BYTE_PTR pNonce;
    CK_ULONG ulNonceLen;
    CK_BYTE_PTR pAAD;
    CK_ULONG ulAADLen;
    CK_ULONG ulMACLen;
} CK_CCM_PARAMS;

The fields of the structure have the following meanings, where L is the size in bytes of the data length’s length (2 <= L <= 8):

- **ulDataLen** length of the data where 0 <= ulDataLen < \(2^8(L)\).
- **pNonce** the nonce.

pAAD Additional authentication data. This data is authenticated but not encrypted.

ulAADLen length of pAAD in bytes where 0 <= ulAADLen <= (2^32) - 1.

ulMACLen length of the MAC (output following cipher text) in bytes. Valid values are 4, 6, 8, 10, 12, 14, and 16.

CK_CCM_PARAMS_PTR is a pointer to a CK_CCM_PARAMS.

CK_CCM_PARAMS is a structure that provides the parameters to the CKM_AES_CCM mechanism when used for MessageEncrypt or MessageDecrypt. It is defined as follows:

typedef struct CK_CCM_MESSAGE_PARAMS {
    CK_ULONG ulDataLen; /*plaintext or ciphertext*/
    CK_BYTE_PTR pNonce;
    CK_ULONG ulNonceLen;
    CK_ULONG ulNonceFixedBits;
    CK_GENERATOR_FUNCTION nonceGenerator;
    CK_BYTE_PTR pMAC;
    CK_ULONG ulMACLen;
} CK_CCM_MESSAGE_PARAMS;

The fields of the structure have the following meanings, where L is the size in bytes of the data length's length (2 <= L <= 8):

ulDataLen length of the data where 0 <= ulDataLen < 2^(8L).

pNonce the nonce.


ulNonceFixedBits number of bits of the original nonce to preserve when generating a new nonce. These bits are counted from the Most significant bits (to the right).

nonceGenerator Function used to generate a new nonce. Each nonce must be unique for a given session.

pMAC location of the CCM MAC returned on MessageEncrypt, provided on MessageDecrypt.

ulMACLen length of the MAC (output following cipher text) in bytes. Valid values are 4, 6, 8, 10, 12, 14, and 16.

CK_CCM_MESSAGE_PARAMS_PTR is a pointer to a CK_CCM_MESSAGE_PARAMS.
2.14 AES CMAC

Table 86, Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR¹</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_AES_CMAC_GENERAL</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_AES_CMAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

¹ SR = SignRecover, VR = VerifyRecover.

2.14.1 Definitions

Mechanisms:
- CKM_AES_CMAC_GENERAL
- CKM_AES_CMAC

2.14.2 Mechanism parameters

CKM_AES_CMAC_GENERAL uses the existing CK_MAC_GENERAL_PARAMS structure.
CKM_AES_CMAC does not use a mechanism parameter.

2.14.3 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 87, General-length AES-CMAC: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>CKK_AES</td>
<td>any</td>
<td>1-block size, as specified in parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>CKK_AES</td>
<td>any</td>
<td>1-block size, as specified in parameters</td>
</tr>
</tbody>
</table>

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.

2.14.4 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 88, AES-CMAC: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>CKK_AES</td>
<td>any</td>
<td>Block size (16 bytes)</td>
</tr>
<tr>
<td>C_Verify</td>
<td>CKK_AES</td>
<td>any</td>
<td>Block size (16 bytes)</td>
</tr>
</tbody>
</table>

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.

2.15 AES XTS

Table 89, Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Encrypt &amp; Decrypt</td>
</tr>
<tr>
<td>CKM_AES_XTS</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_AES_XTS_KEY_GEN</td>
<td></td>
</tr>
</tbody>
</table>

2.15.1 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_CTS_KEY_GEN

2.15.2 AES-XTS secret key objects

Table 90, AES-XTS Secret Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE1,4,6,7</td>
<td>Byte array</td>
<td>Key value (32 or 64 bytes)</td>
</tr>
<tr>
<td>CKA_VALUE_LEN2,3,8</td>
<td>CK_UULONG</td>
<td>Length in bytes of key value</td>
</tr>
</tbody>
</table>

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

2.15.3 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.
2.15.4 AES-XTS

AES-XTS (XEX-based Tweaked CodeBook mode with CipherText Stealing), denoted CKM_AES_XTS, is a mechanism for single- and multiple-part encryption and decryption. It is specified in NIST SP800-38E.

Its single parameter is a Data Unit Sequence Number 16 bytes long. Supported key lengths are 32 and 64 bytes. Keys are internally split into half-length sub-keys of 16 and 32 bytes respectively. Constraints on key types and the length of data are summarized in the following table:

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>CKK_AES_XTS</td>
<td>Any, ≥ block size (16 bytes)</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>CKK_AES_XTS</td>
<td>Any, ≥ block size (16 bytes)</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
</tbody>
</table>

Table 91, AES-XTS: Key And Data Length

2.16 AES Key Wrap

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR¹</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_AES_KEY_WRAP</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_AES_KEY_WRAP_PAD</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_AES_KEY_WRAP_KWP</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

¹SR = SignRecover, VR = VerifyRecover

2.16.1 Definitions

Mechanisms:
- CKM_AES_KEY_WRAP
- CKM_AES_KEY_WRAP_PAD
- CKM_AES_KEY_WRAP_KWP

2.16.2 AES Key Wrap Mechanism parameters

The mechanisms will accept an optional mechanism parameter as the Initialization vector which, if present, must be a fixed size array of 8 bytes for CKM_AES_KEY_WRAP and CKM_AES_KEY_WRAP_PAD, resp. 4 bytes for CKM_AES_KEY_WRAP_KWP; and, if NULL, will use the default initial value defined in Section 4.3 resp. 6.2 / 6.3 of [AES KEYWRAP].

The type of this parameter is CK_BYTE_PTR and the pointer points to the array of bytes to be used as the initial value. The length shall be either 0 and the pointer NULL; or 8 for CKM_AES_KEY_WRAP / CKM_AES_KEY_WRAP_PAD, resp. 4 for CKM_AES_KEY_WRAP_KWP, and the pointer non-NULL.

2.16.3 AES Key Wrap

The mechanisms support only single-part operations, single part wrapping and unwrapping, and single-part encryption and decryption.
The CKM_AES_KEY_WRAP mechanism can only wrap a key resp. encrypt a block of data whose size is an exact multiple of the AES Key Wrap algorithm block size. Wrapping / encryption is done as defined in Section 6.2 of [AES KEYWRAP].

The CKM_AES_KEY_WRAP_PAD mechanism can wrap a key or encrypt a block of data of any length. It does the padding detailed in PKCS #7 of inputs (keys or data blocks), always producing wrapped output that is larger than the input key/data to be wrapped. This padding is done by the token before being passed to the AES key wrap algorithm, which then wraps / encrypts the padded block of data as defined in Section 6.2 of [AES KEYWRAP].

The CKM_AES_KEY_WRAP_KWP mechanism can wrap a key or encrypt block of data of any length. The input is padded and wrapped / encrypted as defined in Section 6.3 of [AES KEYWRAP], which produces same results as RFC 5649.

### 2.17 Key derivation by data encryption – DES & AES

These mechanisms allow derivation of keys using the result of an encryption operation as the key value. They are for use with the C_DeriveKey function.

**Table 93, Key derivation by data encryption Mechanisms vs. Functions**

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/ Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_DES_ECB_ENCRYPT_DATA</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DES_CBC_ENCRYPT_DATA</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DES3_ECB_ENCRYPT_DATA</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DES3_CBC_ENCRYPT_DATA</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_AES_ECB_ENCRYPT_DATA</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_AES_CBC_ENCRYPT_DATA</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

#### 2.17.1 Definitions

Mechanisms:

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

typedef struct CK_DES_CBC_ENCRYPT_DATA_PARAMS {
    CK_BYTE iv[8];
    CK_BYTE_PTR pData;
    CK_ULONG length;
} CK_DES_CBC_ENCRYPT_DATA_PARAMS;

typedef CK_DES_CBC_ENCRYPT_DATA_PARAMS CK_PTR
       CK_DES_CBC_ENCRYPT_DATA_PARAMS_PTR;
typedef struct CK_AES_CBC_ENCRYPT_DATA_PARAMS {
    CK_BYTE iv[16];
    CK_BYTE_PTR pData;
    CK_ULONG length;
} CK_AES_CBC_ENCRYPT_DATA_PARAMS;

typedef CK_AES_CBC_ENCRYPT_DATA_PARAMS CK_PTR CK_AES_CBC_ENCRYPT_DATA_PARAMS_PTR;

### 2.17.2 Mechanism Parameters

Uses CK_KEY_DERIVATION_STRING_DATA as defined in section 2.43.2

Table 94, Mechanism Parameters

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_DES_ECB_ENCRYPT_DATA</td>
<td>Uses CK_KEY_DERIVATION_STRING_DATA structure. Parameter is the data to be encrypted and must be a multiple of 8 bytes long.</td>
</tr>
<tr>
<td>CKM_DES3_ECB_ENCRYPT_DATA</td>
<td>Uses CK_KEY_DERIVATION_STRING_DATA structure. Parameter is the data to be encrypted and must be a multiple of 16 long.</td>
</tr>
<tr>
<td>CKM_AES_ECB_ENCRYPT_DATA</td>
<td>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.</td>
</tr>
<tr>
<td>CKM_DES_CBC_ENCRYPT_DATA</td>
<td>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.</td>
</tr>
</tbody>
</table>

### 2.17.3 Mechanism Description

The mechanisms will function by performing the encryption over the data provided using the base key. The resulting cipher text shall be used to create the key value of the resulting key. If not all the cipher text is used then the part discarded will be from the trailing end (least significant bytes) of the cipher text data. The derived key shall be defined by the attribute template supplied but constrained by the length of cipher text available for the key value and other normal PKCS11 derivation constraints.

Attribute template handling, attribute defaulting and key value preparation will operate as per the SHA-1 Key Derivation mechanism in section 2.20.5.

If the data is too short to make the requested key then the mechanism returns CKR_DATA_LEN_RANGE.

### 2.18 Double and Triple-length DES

Table 95, Double and Triple-Length DES Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR¹</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_DES2_KEY_GEN</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DES3_KEY_GEN</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DES3_ECB</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_DES3_CBC</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/ Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_DES3_CBC_PAD</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_DES3_MAC_GENERAL</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_DES3_MAC</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.18.1 Definitions

This section defines the key type “CKK_DES2” and “CKK_DES3” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:
- CKM_DES2_KEY_GEN
- CKM_DES3_KEY_GEN
- CKM_DES3_ECB
- CKM_DES3_CBC
- CKM_DES3_MAC
- CKM_DES3_MAC_GENERAL
- CKM_DES3_CBC_PAD

2.18.2 DES2 secret key objects

DES2 secret key objects (object class CKO_SECRET_KEY, key type CKK_DES2) hold double-length DES keys. The following table defines the DES2 secret key object attributes, in addition to the common attributes defined for this object class:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE1,4,6,7</td>
<td>Byte array</td>
<td>Key value (always 16 bytes long)</td>
</tr>
</tbody>
</table>

DES2 keys must always have their parity bits properly set as described in FIPS PUB 46-3 (i.e., each of the DES keys comprising a DES2 key must have its parity bits properly set). Attempting to create or unwrap a DES2 key with incorrect parity will return an error.

The following is a sample template for creating a double-length DES secret key object:

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_DES2;
CK_UTF8CHAR label[] = "A DES2 secret key object";
CK_BYTE value[16] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_VALUE, &true, sizeof(value)}
};
```
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.

### 2.18.3 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>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE</td>
<td>Byte array</td>
<td>Key value (always 24 bytes long)</td>
</tr>
</tbody>
</table>

- Refer to [PKCS11-Base] 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:

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_DES3;
CK_UTF8CHAR label[] = "A DES3 secret key object";
CK_BYTE value[24] = {...};
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_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 ECB encryption of a single block of null (0x00) bytes, using the default cipher associated with the key type of the secret key object.

### 2.18.4 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

---

#### Table 97, DES3 Secret Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE</td>
<td>Byte array</td>
<td>Key value (always 24 bytes long)</td>
</tr>
</tbody>
</table>
**2.18.5 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:

\[
\begin{align*}
\text{DES3-E} & (\{K_1, K_2, K_3\}, P) = E(K_3, D(K_2, E(K_1, P))) \\
\text{DES3-D} & (\{K_1, K_2, K_3\}, C) = D(K_1, E(K_2, D(K_3, P)))
\end{align*}
\]

**2.18.6 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:

\[
\begin{align*}
\text{DES3-CBC-E} & (\{K_1, K_2, K_3\}, P) = E(K_3, D(K_2, E(K_1, P + I))) \\
\text{DES3-CBC-D} & (\{K_1, K_2, K_3\}, C) = D(K_1, E(K_2, D(K_3, P))) + I
\end{align*}
\]

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 used is addition modulo-2 (XOR).

**2.18.7 DES and Triple length DES in OFB Mode**

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/ Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_DES_OFB64</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_DES_OFB8</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_DES_OFB64</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_DES_CFB64</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cipher DES has a output feedback mode, DES-OFB, denoted **CKM_DES_OFB8** and **CKM_DES_OFB64**. It is a mechanism for single and multiple-part encryption and decryption with DES. It has a parameter, an initialization vector for this mode. The initialization vector has the same length as the block size.

Constraints on key types and the length of data are summarized in the following table:
Table 99, OFB: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>CKK_DES, CKK_DES2, CKK_DES3</td>
<td>any</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>CKK_DES, CKK_DES2, CKK_DES3</td>
<td>any</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
</tbody>
</table>

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

2.18.8 DES and Triple length DES in CFB Mode

Cipher DES has a cipher feedback mode, DES-CFB, denoted **CKM_DES_CFB8** and **CKM_DES_CFB64**. It is a mechanism for single and multiple-part encryption and decryption with DES. It has a parameter, an initialization vector for this mode. The initialization vector has the same length as the block size.

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

Table 100, CFB: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>CKK_DES, CKK_DES2, CKK_DES3</td>
<td>any</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>CKK_DES, CKK_DES2, CKK_DES3</td>
<td>any</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
</tbody>
</table>

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

2.19 Double and Triple-length DES CMAC

Table 101, Double and Triple-length DES CMAC Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR¹</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_DES3_CMAC_GENERAL</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_DES3_CMAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

¹ SR = SignRecover, VR = VerifyRecover.

The following additional DES3 mechanisms have been added.

2.19.1 Definitions

Mechanisms:

CKM_DES3_CMAC_GENERAL

CKM_DES3_CMAC
2.19.2 Mechanism parameters

CKM_DES3_CMAC_GENERAL uses the existing CK_MAC_GENERAL_PARAMS structure. CKM_DES3_CMAC does not use a mechanism parameter.

2.19.3 General-length DES3-MAC

General-length DES3-CMAC, denoted CKM_DES3_CMAC_GENERAL, is a mechanism for single- and multiple-part signatures and verification with DES3 or DES2 keys, based on [NIST sp800-38b].

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

The output bytes from this mechanism are taken from the start of the final DES3 cipher block produced in the MACing process.

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

Table 102, General-length DES3-CMAC: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>CKK_DES3</td>
<td>any</td>
<td>1-block size, as specified in parameters</td>
</tr>
<tr>
<td></td>
<td>CKK_DES2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C_Verify</td>
<td>CKK_DES3</td>
<td>any</td>
<td>1-block size, as specified in parameters</td>
</tr>
<tr>
<td></td>
<td>CKK_DES2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Reference [NIST sp800-38b] recommends that the output MAC is not truncated to less than 64 bits (which means using the entire block for DES). The MAC length must be specified before the communication starts, and must not be changed during the lifetime of the key. It is the caller’s responsibility to follow these rules.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure are not used.

2.19.4 DES3-CMAC

DES3-CMAC, denoted CKM_DES3_CMAC, is a special case of the general-length DES3-CMAC mechanism. DES3-MAC always produces and verifies MACs that are a full block size in length, since the DES3 block length is the minimum output length recommended by [NIST sp800-38b].

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

Table 103, DES3-CMAC: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>CKK_DES3</td>
<td>any</td>
<td>Block size (8 bytes)</td>
</tr>
<tr>
<td></td>
<td>CKK_DES2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C_Verify</td>
<td>CKK_DES3</td>
<td>any</td>
<td>Block size (8 bytes)</td>
</tr>
<tr>
<td></td>
<td>CKK_DES2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure are not used.

2.20 SHA-1

Table 104, SHA-1 Mechanisms vs. Functions
### 2.20.1 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

---

### 2.20.2 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 105, SHA-1: Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>20</td>
</tr>
</tbody>
</table>

---

### 2.20.3 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 106, General-length SHA-1-HMAC: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret</td>
<td>any</td>
<td>1-20, depending on parameters</td>
</tr>
<tr>
<td></td>
<td>CKK_SHA_1_HMAC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret</td>
<td>any</td>
<td>1-20, depending on parameters</td>
</tr>
<tr>
<td></td>
<td>CKK_SHA_1_HMAC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.20.4 SHA-1-HMAC

The SHA-1-HMAC mechanism, denoted CKM_SHA_1_HMAC, is a special case of the general-length SHA-1-HMAC mechanism in Section 2.20.3.

It has no parameter, and always produces an output of length 20.

2.20.5 SHA-1 key derivation

SHA-1 key derivation, denoted CKM_SHA1_KEY_DERIVATION, is a mechanism which provides the capability of deriving a secret key by digesting the value of another secret key with SHA-1.

The value of the base key is digested once, and the result is used to make the value of derived secret key.

- If no length or key type is provided in the template, then the key produced by this mechanism will be a generic secret key. Its length will be 20 bytes (the output size of SHA-1).
- If no key type is provided in the template, but a length is, then the key produced by this mechanism will be a generic secret key of the specified length.
- If no length was provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn't, an error will be returned.
- If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism will be of the specified type and length.

If a DES, DES2, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more than 20 bytes, such as DES3, an error is generated.

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

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

2.20.6 SHA-1 HMAC key generation

The SHA-1-HMAC key generation mechanism, denoted CKM_SHA_1_KEY_GEN, is a key generation mechanism for NIST’s SHA-1-HMAC.

It does not have a parameter.
The mechanism generates SHA-1-HMAC keys with a particular length in bytes, as specified in the
\texttt{CKA\_VALUE\_LEN} attribute of the template for the key.

The mechanism contributes the \texttt{CKA\_CLASS}, \texttt{CKA\_KEY\_TYPE}, and \texttt{CKA\_VALUE} attributes to the new
key. Other attributes supported by the SHA-1-HMAC key type (specifically, the flags indicating which
functions the key supports) may be specified in the template for the key, or else are assigned default
initial values.

For this mechanism, the \texttt{ulMinKeySize} and \texttt{ulMaxKeySize} fields of the \texttt{CK\_MECHANISM\_INFO} structure
specify the supported range of \texttt{CKM\_SHA\_1\_HMAC} key sizes, in bytes.

## 2.21 SHA-224

\textit{Table 107, SHA-224 Mechanisms vs. Functions}

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR\textsuperscript{1}</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_SHA224</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA224_HMAC</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA224_HMAC_GENERAL</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA224_RSA_PKCS</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA224_RSA_PKCS_PSS</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA224_KEY_DERIVATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA224_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

### 2.21.1 Definitions

This section defines the key type "CKK\_SHA224\_HMAC" for type \texttt{CK\_KEY\_TYPE} as used in the
\texttt{CKA\_KEY\_TYPE} attribute of key objects.

Mechanisms:

- CKM\_SHA224
- CKM\_SHA224\_HMAC
- CKM\_SHA224\_HMAC\_GENERAL
- CKM\_SHA224\_KEY\_DERIVATION
- CKM\_SHA224\_KEY\_GEN

### 2.21.2 SHA-224 digest

The SHA-224 mechanism, denoted \texttt{CKM\_SHA224}, is a mechanism for message digesting, following the
Secure Hash Algorithm with a 224-bit message digest defined in 0.

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 108, SHA-224: Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>28</td>
</tr>
</tbody>
</table>

2.21.3 General-length SHA-224-HMAC

The general-length SHA-224-HMAC mechanism, denoted **CKM_SHA224_HMAC_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism except that it uses the HMAC construction based on the SHA-224 hash function and length of the output should be in the range 1-28. The keys it uses are generic secret keys and **CKK_SHA224_HMAC**. FIPS-198 compliant tokens may require the key length to be at least 14 bytes; that is, half the size of the SHA-224 hash output.

It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired output. This length should be in the range 1-28 (the output size of SHA-224 is 28 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 14 (half the maximum length).

Signatures (MACs) produced by this mechanism will be taken from the start of the full 28-byte HMAC output.

Table 109, General-length SHA-224-HMAC: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret</td>
<td>Any</td>
<td>1-28, depending on parameters</td>
</tr>
<tr>
<td></td>
<td>C_KK_SHA224_HMAC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret</td>
<td>Any</td>
<td>1-28, depending on parameters</td>
</tr>
<tr>
<td></td>
<td>C_KK_SHA224_HMAC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.21.4 SHA-224-HMAC

The SHA-224-HMAC mechanism, denoted **CKM_SHA224_HMAC**, is a special case of the general-length SHA-224-HMAC mechanism.

It has no parameter, and always produces an output of length 28.

2.21.5 SHA-224 key derivation

SHA-224 key derivation, denoted **CKM_SHA224_KEY_DERIVATION**, is the same as the SHA-1 key derivation mechanism in Section 12.21.5 except that it uses the SHA-224 hash function and the relevant length is 28 bytes.

2.21.6 SHA-224 HMAC key generation

The SHA-224-HMAC key generation mechanism, denoted **CKM_SHA224_KEY_GEN**, is a key generation mechanism for NIST’s SHA224-HMAC.

It does not have a parameter.

The mechanism generates SHA224-HMAC keys with a particular length in bytes, as specified in the **CKA_VALUE_LEN** attribute of the template for the key.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the SHA224-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the **ulMinKeySize** and **ulMaxKeySize** fields of the **CK_MECHANISM_INFO** structure specify the supported range of **CKM_SHA224_HMAC** key sizes, in bytes.
2.22 SHA-256

Table 110, SHA-256 Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR¹</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_SHA256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA256_HMAC_GENERAL</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA256_HMAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA256_KEY_DERIVATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA256_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

2.22.1 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

2.22.2 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 111, SHA-256: Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>32</td>
</tr>
</tbody>
</table>

2.22.3 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 2.20.3, 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 112, General-length SHA-256-HMAC: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret, CKK_SHA256_HMAC</td>
<td>Any</td>
<td>1-32, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret, CKK_SHA256_HMAC</td>
<td>Any</td>
<td>1-32, depending on parameters</td>
</tr>
</tbody>
</table>

2.22.4 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 2.22.3.

It has no parameter, and always produces an output of length 32.

2.22.5 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 2.20.5, except that it uses the SHA-256 hash function and the relevant length is 32 bytes.

2.22.6 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.

2.23 SHA-384

Table 113, SHA-384 Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR¹</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_SHA384</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA384_HMAC_GENERAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA384_HMAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA384_KEY_DERIVATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA384_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.23.1 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.

CKM_SHA384
CKM_SHA384_HMAC
CKM_SHA384_HMAC_GENERAL
CKM_SHA384_KEY_DERIVATION
CKM_SHA384_KEY_GEN

2.23.2 SHA-384 digest
The SHA-384 mechanism, denoted CKM_SHA384, is a mechanism for message digesting, following the Secure Hash Algorithm with a 384-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 114, SHA-384: Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>48</td>
</tr>
</tbody>
</table>

2.23.3 General-length SHA-384-HMAC
The general-length SHA-384-HMAC mechanism, denoted CKM_SHA384_HMAC_GENERAL, is the same as the general-length SHA-1-HMAC mechanism in Section 2.20.3, except that it uses the HMAC construction based on the SHA-384 hash function and length of the output should be in the range 1-48.

The keys it uses are generic secret keys and CKK_SHA384_HMAC. FIPS-198 compliant tokens may require the key length to be at least 24 bytes; that is, half the size of the SHA-384 hash output.

It has a parameter, a CK_MAC_GENERAL_PARAMS, which holds the length in bytes of the desired output. This length should be in the range 0-48 (the output size of SHA-384 is 48 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 24 (half the maximum length). Signatures (MACs) produced by this mechanism will be taken from the start of the full 48-byte HMAC output.

Table 115, General-length SHA-384-HMAC: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret, CKK_SHA384_HMAC</td>
<td>Any</td>
<td>1-48, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret, CKK_SHA384_HMAC</td>
<td>Any</td>
<td>1-48, depending on parameters</td>
</tr>
</tbody>
</table>

2.23.4 SHA-384-HMAC
The SHA-384-HMAC mechanism, denoted CKM_SHA384_HMAC, is a special case of the general-length SHA-384-HMAC mechanism.

It has no parameter, and always produces an output of length 48.
2.23.5 SHA-384 key derivation

SHA-384 key derivation, denoted CKM_SHA384_KEY_DERIVATION, is the same as the SHA-1 key derivation mechanism in Section 2.20.5, except that it uses the SHA-384 hash function and the relevant length is 48 bytes.

2.23.6 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.

2.24 SHA-512

Table 116, SHA-512 Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_SHA512</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA512_HMAC_GENERAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA512_HMAC</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA512_KEY_DERIVATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_SHA512_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

2.24.1 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

2.24.2 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>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>64</td>
</tr>
</tbody>
</table>

### 2.24.3 General-length SHA-512-HMAC

The general-length SHA-512-HMAC mechanism, denoted **CKM_SHA512_HMAC_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in Section 2.20.3, except that it uses the HMAC construction based on the SHA-512 hash function and length of the output should be in the range 1-64.

The keys it uses are generic secret keys and **CKK_SHA512_HMAC**. FIPS-198 compliant tokens may require the key length to be at least 32 bytes; that is, half the size of the SHA-512 hash output.

It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired output. This length should be in the range 0-64 (the output size of SHA-512 is 64 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 32 (half the maximum length). Signatures (MACs) produced by this mechanism will be taken from the start of the full 64-byte HMAC output.

### 2.24.4 SHA-512-HMAC

The SHA-512-HMAC mechanism, denoted **CKM_SHA512_HMAC**, is a special case of the general-length SHA-512-HMAC mechanism.

It has no parameter, and always produces an output of length 64.

### 2.24.5 SHA-512 key derivation

SHA-512 key derivation, denoted **CKM_SHA512_KEY_DERIVATION**, is the same as the SHA-1 key derivation mechanism in Section 2.20.5, except that it uses the SHA-512 hash function and the relevant length is 64 bytes.

### 2.24.6 SHA-512 HMAC key generation

The SHA-512-HMAC key generation mechanism, denoted **CKM_SHA512_KEY_GEN**, is a key generation mechanism for NIST’s SHA512-HMAC.

It does not have a parameter.

The mechanism generates SHA512-HMAC keys with a particular length in bytes, as specified in the **CKA_VALUE_LEN** attribute of the template for the key.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the SHA512-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.
For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure specify the supported range of `CKM_SHA512_HMAC` key sizes, in bytes.

### 2.25 SHA-512/224

*Table 119, SHA-512/224 Mechanisms vs. Functions*

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Encrypt &amp; Decrypt</td>
</tr>
<tr>
<td>CKM_SHA512_224</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA512_224_HMAC_GENERAL</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA512_224_HMAC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA512_224_KEY_DERIVATION</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA512_224_KEY_GEN</td>
<td>✓</td>
</tr>
</tbody>
</table>

#### 2.25.1 Definitions

This section defines the key type "CKK_SHA512_224_HMAC" for type `CK_KEY_TYPE` as used in the `CKA_KEY_TYPE` attribute of key objects.

Mechanisms:

- CKM_SHA512_224
- CKM_SHA512_224_HMAC
- CKM_SHA512_224_HMAC_GENERAL
- CKM_SHA512_224_KEY_DERIVATION
- CKM_SHA512_224_KEY_GEN

#### 2.25.2 SHA-512/224 digest

The SHA-512/224 mechanism, denoted `CKM_SHA512_224`, is a mechanism for message digesting, following the Secure Hash Algorithm defined in FIPS PUB 180-4, section 5.3.6. It is based on a 512-bit message digest with a distinct initial hash value and truncated to 224 bits. `CKM_SHA512_224` is the same as `CKM_SHA512_T` with a parameter value of 224.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

*Table 120, SHA-512/224: Data Length*

<table>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>28</td>
</tr>
</tbody>
</table>

#### 2.25.3 General-length SHA-512/224-HMAC

The general-length SHA-512/224-HMAC mechanism, denoted `CKM_SHA512_224_HMAC_GENERAL`, is the same as the general-length SHA-1-HMAC mechanism in Section 2.20.3, except that it uses the HMAC construction based on the SHA-512/224 hash function and length of the output should be in the range 1-28. The keys it uses are generic secret keys and `CKK_SHA512_224_HMAC`. FIPS-198
compliant tokens may require the key length to be at least 14 bytes; that is, half the size of the SHA-512/224 hash output.

It has a parameter, a CK_MAC_GENERAL_PARAMS, which holds the length in bytes of the desired output. This length should be in the range 0-28 (the output size of SHA-512/224 is 28 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 14 (half the maximum length). Signatures (MACs) produced by this mechanism will be taken from the start of the full 28-byte HMAC output.

Table 121, General-length SHA-384-HMAC: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret, CKK_SHA512_224_HMAC</td>
<td>Any</td>
<td>1-28, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret, CKK_SHA512_224_HMAC</td>
<td>Any</td>
<td>1-28, depending on parameters</td>
</tr>
</tbody>
</table>

2.25.4 SHA-512/224-HMAC

The SHA-512-HMAC mechanism, denoted CKM_SHA512_224_HMAC, is a special case of the general-length SHA-512/224-HMAC mechanism.

It has no parameter, and always produces an output of length 28.

2.25.5 SHA-512/224 key derivation

The SHA-512/224 key derivation, denoted CKM_SHA512_224_KEY_DERIVATION, is the same as the SHA-512 key derivation mechanism in section 2.25.5, except that it uses the SHA-512/224 hash function and the relevant length is 28 bytes.

2.25.6 SHA-512/224 HMAC key generation

The SHA-512/224-HMAC key generation mechanism, denoted CKM_SHA512_224_KEY_GEN, is a key generation mechanism for NIST’s SHA512/224-HMAC.

It does not have a parameter.

The mechanism generates SHA512/224-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 SHA512/224-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of CKM_SHA512_224_HMAC key sizes, in bytes.

2.26 SHA-512/256

Table 122, SHA-512/256 Mechanisms vs. Functions
### Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encryp &amp; Decryp</th>
<th>Sign &amp; Verif</th>
<th>SR &amp; VR</th>
<th>Diges t</th>
<th>Gen. Key/ Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_SHA512_256</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA512_256_HMAC_GENERAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA512_256_HMAC</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA512_256_KEY_DERIVATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA512_256_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 2.26.1 Definitions

This section defines the key type “CKK_SHA512_256_HMAC” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

- CKM_SHA512_256
- CKM_SHA512_256_HMAC
- CKM_SHA512_256_HMAC_GENERAL
- CKM_SHA512_256_KEY_DERIVATION
- CKM_SHA512_256_KEY_GEN

#### 2.26.2 SHA-512/256 digest

The SHA-512/256 mechanism, denoted CKM_SHA512_256, is a mechanism for message digesting, following the Secure Hash Algorithm defined in FIPS PUB 180-4, section 5.3.6. It is based on a 512-bit message digest with a distinct initial hash value and truncated to 256 bits. CKM_SHA512_256 is the same as CKM_SHA512_T with a parameter value of 256.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

<table>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>32</td>
</tr>
</tbody>
</table>

#### 2.26.3 General-length SHA-512/256-HMAC

The general-length SHA-512/256-HMAC mechanism, denoted CKM_SHA512_256_HMAC_GENERAL, is the same as the general-length SHA-1-HMAC mechanism in Section 2.20.3, except that it uses the HMAC construction based on the SHA-512/256 hash function and length of the output should be in the range 1-32. The keys it uses are generic secret keys and CKK_SHA512_256_HMAC. FIPS-198 compliant tokens may require the key length to be at least 16 bytes; that is, half the size of the SHA-512/256 hash output.

It has a parameter, a CK_MAC_GENERAL_PARAMS, which holds the length in bytes of the desired output. This length should be in the range 1-32 (the output size of SHA-512/256 is 32 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 16 (half the maximum length).
Signatures (MACs) produced by this mechanism will be taken from the start of the full 32-byte HMAC output.

Table 124, General-length SHA-384-HMAC: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret, CKK_SHA512_256_HMAC</td>
<td>Any</td>
<td>1-32, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret, CKK_SHA512_256_HMAC</td>
<td>Any</td>
<td>1-32, depending on parameters</td>
</tr>
</tbody>
</table>

2.26.4 SHA-512/256-HMAC

The SHA-512-HMAC mechanism, denoted **CKM_SHA512_256_HMAC**, is a special case of the general-length SHA-512/256-HMAC mechanism. It has no parameter, and always produces an output of length 32.

2.26.5 SHA-512/256 key derivation

The SHA-512/256 key derivation, denoted **CKM_SHA512_256_KEY_DERIVATION**, is the same as the SHA-512 key derivation mechanism in section 2.25.5, except that it uses the SHA-512/256 hash function and the relevant length is 32 bytes.

2.26.6 SHA-512/256 HMAC key generation

The SHA-512/256-HMAC key generation mechanism, denoted **CKM_SHA512_256_KEY_GEN**, is a key generation mechanism for NIST’s SHA512/256-HMAC. It does not have a parameter. The mechanism generates SHA512/256-HMAC keys with a particular length in bytes, as specified in the **CKA_VALUE_LEN** attribute of the template for the key. The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the SHA512/256-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values. For this mechanism, the **ulMinKeySize** and **ulMaxKeySize** fields of the **CK_MECHANISM_INFO** structure specify the supported range of **CKM_SHA512_256_HMAC** key sizes, in bytes.

2.27 SHA-512/t

Table 125, SHA-512 / t Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR†</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_SHA512_T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA512_T_HMAC_GENERAL</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA512_T_HMAC</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA512_T_KEY_DERIVATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
2.27.1 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
- CKK_SHA512_T_KEY_GEN

2.27.2 SHA-512/t digest

The SHA-512/t mechanism, denoted CKM_SHA512_T, is a mechanism for message digesting, following
the Secure Hash Algorithm defined in FIPS PUB 180-4, section 5.3.6. It is based on a 512-bit message
digest with a distinct initial hash value and truncated to t bits.

It has a parameter, a CK_MAC_GENERAL_PARAMS, which holds the value of t in bits. The length in
bytes of the desired output should be in the range of 0 \leq t < 512, and t 
eq 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>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>(\lceil t/8 \rceil), where 0 &lt; t &lt; 512, and t \neq 384</td>
</tr>
</tbody>
</table>

2.27.3 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 2.20.3, except that it uses the HMAC
construction based on the SHA-512/t hash function and length of the output should be in the range 0 \leq \(\lceil t/8 \rceil\), where 0 < t < 512, and t \neq 384.

2.27.4 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_MACGENERAL_PARAMS, which holds the value of t in bits. The length in
bytes of the desired output should be in the range of 0 \leq \(\lceil t/8 \rceil\), where 0 < t < 512, and t \neq 384.
2.27.5 SHA-512/t key derivation

The SHA-512/t key derivation, denoted \texttt{CKM\_SHA512\_T\_KEY\_DERIVATION}, is the same as the SHA-512 key derivation mechanism in section 2.25.5, except that it uses the SHA-512 hash function and the relevant length is \(\lceil t/8 \rceil\) bytes, where \(0 < t < 512\), and \(t <> 384\).

2.27.6 SHA-512/t HMAC key generation

The SHA-512/t-HMAC key generation mechanism, denoted \texttt{CKM\_SHA512\_T\_KEY\_GEN}, is a key generation mechanism for NIST’s SHA512/t-HMAC.

It does not have a parameter.

The mechanism generates SHA512/t-HMAC keys with a particular length in bytes, as specified in the \texttt{CKA\_VALUE\_LEN} attribute of the template for the key.

The mechanism contributes the \texttt{CKA\_CLASS}, \texttt{CKA\_KEY\_TYPE}, and \texttt{CKA\_VALUE} attributes to the new key. Other attributes supported by the SHA512/t-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the \texttt{ulMinKeySize} and \texttt{ulMaxKeySize} fields of the \texttt{CK\_MECHANISM\_INFO} structure specify the supported range of \texttt{CKM\_SHA512\_T\_HMAC} key sizes, in bytes.

2.28 SHA3-224

Table 127, SHA-224 Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_SHA3_224</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA3_224_HMAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA3_224_HMAC_GENERAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA3_224_KEY_DERIVATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA3_224_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

2.28.1 Definitions

Mechanisms:

- CKM_SHA3_224
- CKM_SHA3_224_HMAC
- CKM_SHA3_224_HMAC_GENERAL
- CKM_SHA3_224_KEY_DERIVATION
- CKM_SHA3_224_KEY_GEN
- CKK_SHA3_224_HMAC
2.28.2 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>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>28</td>
</tr>
</tbody>
</table>

2.28.3 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 2.20.4 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>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret or CKK_SHA3_224_HMAC</td>
<td>Any</td>
<td>1-28, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret or CKK_SHA3_224_HMAC</td>
<td>Any</td>
<td>1-28, depending on parameters</td>
</tr>
</tbody>
</table>

2.28.4 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.

2.28.5 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 2.20.5 except that it uses the SHA3-224 hash function and the relevant length is 28 bytes.

2.28.6 SHA3-224 HMAC key generation

The SHA3-224-HMAC key generation mechanism, denoted **CKM_SHA3_224_KEY_GEN**, is a key generation mechanism for NIST's SHA3-224-HMAC.

It does not have a parameter.

The mechanism generates SHA3-224-HMAC keys with a particular length in bytes, as specified in the **CKA_VALUE_LEN** attribute of the template for the key.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the SHA3-224-HMAC key type (specifically, the flags indicating which
functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of CKM_SHA3_224_HMAC key sizes, in bytes.

### 2.29 SHA3-256

**Table 130, SHA3-256 Mechanisms vs. Functions**

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR(^1)</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_SHA3_256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA3_256_HMAC_GENERAL</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA3_256_HMAC</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA3_256_KEY_DERIVATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_SHA3_256_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

#### 2.29.1 Definitions

Mechanisms:
- CKM_SHA3_256
- CKM_SHA3_256_HMAC
- CKM_SHA3_256_HMAC_GENERAL
- CKM_SHA3_256_KEY_DERIVATION
- CKM_SHA3_256_KEY_GEN
- CKK_SHA3_256_HMAC

#### 2.29.2 SHA3-256 digest

The SHA3-256 mechanism, denoted **CKM_SHA3_256**, is a mechanism for message digesting, following the Secure Hash 3 Algorithm with a 256-bit message digest defined in FIPS PUB 202.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

**Table 131, SHA3-256: Data Length**

<table>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>32</td>
</tr>
</tbody>
</table>

#### 2.29.3 General-length SHA3-256-HMAC

The general-length SHA3-256-HMAC mechanism, denoted **CKM_SHA3_256_HMAC_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in Section 2.20.4, except that it uses the HMAC construction based on the SHA3-256 hash function and length of the output should be in the range 1-32.

The keys it uses are generic secret keys and CKK_SHA3_256_HMAC. FIPS-198 compliant tokens may require the key length to be at least 16 bytes; that is, half the size of the SHA3-256 hash output.
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 SHA3-256 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 shall be taken from the start of the full 32-byte HMAC output.

Table 132, General-length SHA3-256-HMAC: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret or CKK_SHA3_256_HMAC</td>
<td>Any</td>
<td>1-32, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret or CKK_SHA3_256_HMAC</td>
<td>Any</td>
<td>1-32, depending on parameters</td>
</tr>
</tbody>
</table>

2.29.4 SHA3-256-HMAC

The SHA-256-HMAC mechanism, denoted CKM_SHA3_256_HMAC, is a special case of the general-length SHA-256-HMAC mechanism in Section 2.22.3. It has no parameter, and always produces an output of length 32.

2.29.5 SHA3-256 key derivation

SHA-256 key derivation, denoted CKM_SHA3_256_KEY_DERIVATION, is the same as the SHA-1 key derivation mechanism in Section 2.20.5, except that it uses the SHA3-256 hash function and the relevant length is 32 bytes.

2.29.6 SHA3-256 HMAC key generation

The SHA3-256-HMAC key generation mechanism, denoted CKM_SHA3_256_KEY_GEN, is a key generation mechanism for NIST’s SHA3-256-HMAC. It does not have a parameter. The mechanism generates SHA3-256-HMAC keys with a particular length in bytes, as specified in the CKA_VALUE_LEN attribute of the template for the key.

The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, and CKA_VALUE attributes to the new key. Other attributes supported by the SHA3-256-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of CKM_SHA3_256_HMAC key sizes, in bytes.

2.30 SHA3-384

Table 133, SHA3-384 Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/ Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_SHA3_384</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA3_384_HMAC_GENERAL</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA3_384_HMAC</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_SHA3_384_KEY_DERIVATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA3_384_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

#### 2.30.1 Definitions

- **CKM_SHA3_384**
- **CKM_SHA3_384_HMAC**
- **CKM_SHA3_384_HMAC_GENERAL**
- **CKM_SHA3_384_KEY_DERIVATION**
- **CKM_SHA3_384_KEY_GEN**
- **CKK_SHA3_384_HMAC**

#### 2.30.2 SHA3-384 digest

The SHA3-384 mechanism, denoted **CKM_SHA3_384**, is a mechanism for message digesting, following the Secure Hash 3 Algorithm with a 384-bit message digest defined in FIPS PUB 202.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

<table>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>48</td>
</tr>
</tbody>
</table>

#### 2.30.3 General-length SHA3-384-HMAC

The general-length SHA3-384-HMAC mechanism, denoted **CKM_SHA3_384_HMAC_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in Section 2.20.4, except that it uses the HMAC construction based on the SHA-384 hash function and length of the output should be in the range 1-48. The keys it uses are generic secret keys and **CKK_SHA3_384_HMAC**. FIPS-198 compliant tokens may require the key length to be at least 24 bytes; that is, half the size of the SHA3-384 hash output.

It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired output. This length should be in the range 1-48 (the output size of SHA3-384 is 48 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 24 (half the maximum length).

Signatures (MACs) produced by this mechanism shall be taken from the start of the full 48-byte HMAC output.
Table 135, General-length SHA3-384-HMAC: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret or</td>
<td>Any</td>
<td>1-48, depending on parameters</td>
</tr>
<tr>
<td></td>
<td>CKK_SHA3_384_HMAC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret or</td>
<td>Any</td>
<td>1-48, depending on parameters</td>
</tr>
<tr>
<td></td>
<td>CKK_SHA3_384_HMAC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.30.4 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.

2.30.5 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 2.20.5, except that it uses the SHA-384 hash function and the relevant length is 48 bytes.

2.30.6 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.

2.31 SHA3-512

Table 136, SHA-512 Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Encrypt &amp; Decrypt</td>
</tr>
<tr>
<td>CKM_SHA3_512</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA3_512_HMAC_GENERAL</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA3_512_HMAC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA3_512_KEY_DERIVATION</td>
<td></td>
</tr>
<tr>
<td>CKM_SHA3_512_KEY_GEN</td>
<td></td>
</tr>
</tbody>
</table>
2.31.1 Definitions

CKM_SHA3_512
CKM_SHA3_512_HMAC
CKM_SHA3_512_HMAC_GENERAL
CKM_SHA3_512_KEY_DERIVATION
CKM_SHA3_512_KEY_GEN

2.31.2 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>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>64</td>
</tr>
</tbody>
</table>

2.31.3 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 2.20.4, 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>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret or CKK_SHA3_512-HMAC</td>
<td>Any</td>
<td>1-64, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret or CKK_SHA3_512-HMAC</td>
<td>Any</td>
<td>1-64, depending on parameters</td>
</tr>
</tbody>
</table>

2.31.4 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.
2.31.5 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 2.20.5, except that it uses the SHA-512 hash function and the relevant
length is 64 bytes.

2.31.6 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.

The mechanism contributes the `CKA_CLASS`, `CKA_KEY_TYPE`, and `CKA_VALUE` attributes to the new
key. Other attributes supported by the SHA3-512-HMAC key type (specifically, the flags indicating which
functions the key supports) may be specified in the template for the key, or else are assigned default
initial values.

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure
specify the supported range of `CKM_SHA3_512_HMAC` key sizes, in bytes.

2.32 SHAKE

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_SHAKE_128_KEY_DERIVATION</td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHAKE_256_KEY_DERIVATION</td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.32.1 Definitions

- `CKM_SHAKE_128_KEY_DERIVATION`
- `CKM_SHAKE_256_KEY_DERIVATION`

2.32.2 SHAKE Key Derivation

 SHAKE-128 and SHAKE-256 key derivation, denoted `CKM_SHAKE_128_KEY_DERIVATION` and
`CKM_SHAKE_256_KEY_DERIVATION`, implements the SHAKE expansion function defined in FIPS 202
on the input key.

- If no length or key type is provided in the template a `CKR_INVALID_TEMPLATE` error is generated.
- If no key type is provided in the template, but a length is, then the key produced by this mechanism
shall be a generic secret key of the specified length.
- If no length was provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism shall be of the type specified in
the template. If it doesn't, an error shall be returned.
- If both a key type and a length are provided in the template, the length must be compatible with that
key type. The key produced by this mechanism shall be of the specified type and length.
- If a DES, DES2, or CDMF key is derived with this mechanism, the parity bits of the key shall be set
properly.
This mechanism has the following rules about key sensitivity and extractability:

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

### 2.33 Blake2b-160

#### Table 140, Blake2b-160 Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/ Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_BLAKE2B_160</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_BLAKE2B_160_HMAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_BLAKE2B_160_HMAC_GENERAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_BLAKE2B_160_KEY_DERIVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_BLAKE2B_160_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

#### 2.33.1 Definitions

Mechanisms:

- CKM_BLAKE2B_160
- CKM_BLAKE2B_160_HMAC
- CKM_BLAKE2B_160_HMAC_GENERAL
- CKM_BLAKE2B_160_KEY_DERIVE
- CKM_BLAKE2B_160_KEY_GEN
- CKK_BLAKE2B_160_HMAC

#### 2.33.2 BLAKE2B-160 digest

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. 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.
### 2.33.3 General-length BLAKE2B-160-HMAC

The general-length BLAKE2B-160-HMAC mechanism, denoted **CKM_BLAKE2B_160_HMAC_GENERAL**, is the keyed variant of BLAKE2b-160 and length of the output should be in the range 1-20. The keys it uses are generic secret keys and **CKK_BLAKE2B_160_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 BLAKE2B-160 is 20 bytes). Signatures (MACs) produced by this mechanism shall be taken from the start of the full 20-byte HMAC output.

### Table 142, General-length BLAKE2B-160-HMAC: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret or CKK_BLAKE2B_160_HMAC</td>
<td>Any</td>
<td>1-20, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret or CKK_BLAKE2B_160_HMAC</td>
<td>Any</td>
<td>1-20, depending on parameters</td>
</tr>
</tbody>
</table>

### 2.33.4 BLAKE2B-160-HMAC

The BLAKE2B-160-HMAC mechanism, denoted **CKM_BLAKE2B_160_HMAC**, is a special case of the general-length BLAKE2B-160-HMAC mechanism. It has no parameter, and always produces an output of length 20.

### 2.33.5 BLAKE2B-160 key derivation

BLAKE2B-160 key derivation, denoted **CKM_BLAKE2B_160_KEY_DERIVE**, is the same as the SHA-1 key derivation mechanism in Section 2.20.5 except that it uses the BLAKE2B-160 hash function and the relevant length is 20 bytes.

### 2.33.6 BLAKE2B-160 HMAC key generation

The BLAKE2B-160-HMAC key generation mechanism, denoted **CKM_BLAKE2B_160_KEY_GEN**, is a key generation mechanism for BLAKE2B-160-HMAC. It does not have a parameter. The mechanism generates BLAKE2B-160-HMAC keys with a particular length in bytes, as specified in the **CKA_VALUE_LEN** attribute of the template for the key. The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the BLAKE2B-160-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values. For this mechanism, the **ulMinKeySize** and **ulMaxKeySize** fields of the **CK_MECHANISM_INFO** structure specify the supported range of **CKM_BLAKE2B_160_HMAC** key sizes, in bytes.

### 2.34 BLAKE2B-256

**Table 143, BLAKE2B-256 Mechanisms vs. Functions**
## Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digests</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_BLAKE2B_256</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_BLAKE2B_256_HMAC_GENERAL</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_BLAKE2B_256_HMAC</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_BLAKE2B_256_KEY_DERIVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_BLAKE2B_256_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

### 2.34.1 Definitions

#### Mechanisms:

- CKM_BLAKE2B_256
- CKM_BLAKE2B_256_HMAC
- CKM_BLAKE2B_256_HMAC_GENERAL
- CKM_BLAKE2B_256_KEY_DERIVE
- CKM_BLAKE2B_256_KEY_GEN
- CKK_BLAKE2B_256_HMAC

### 2.34.2 BLAKE2B-256 digest

The BLAKE2B-256 mechanism, denoted **CKM_BLAKE2B_256**, is a mechanism for message digesting, following the Blake2b Algorithm with a 256-bit message digest without a key as defined in RFC 7693.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

<table>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>32</td>
</tr>
</tbody>
</table>

### 2.34.3 General-length BLAKE2B-256-HMAC

The general-length BLAKE2B-256-HMAC mechanism, denoted **CKM_BLAKE2B_256_HMAC_GENERAL**, is the keyed variant of Blake2b-256 and length of the output should be in the range 1-32. The keys it uses are generic secret keys and **CKK_BLAKE2B_256_HMAC**.

It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired output. This length should be in the range 1-32 (the output size of BLAKE2B-256 is 32 bytes). Signatures (MACs) produced by this mechanism shall be taken from the start of the full 32-byte HMAC output.
Table 145, General-length BLAKE2B-256-HMAC: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret or</td>
<td>Any</td>
<td>1-32, depending on parameters</td>
</tr>
<tr>
<td></td>
<td>CKK_BLAKE2B_256_HMAC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret or</td>
<td>Any</td>
<td>1-32, depending on parameters</td>
</tr>
<tr>
<td></td>
<td>CKK_BLAKE2B_256_HMAC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.34.4 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 2.22.3. It has no parameter, and always produces an output of length 32.

2.34.5 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 2.20.5, except that it uses the BLAKE2B-256 hash function and the relevant length is 32 bytes.

2.34.6 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.

2.35 BLAKE2B-384

Table 146, BLAKE2B-384 Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_BLAKE2B_384</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_BLAKE2B_384_HMAC_GERAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_BLAKE2B_384_HMAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_BLAKE2B_384_KEY_DERIVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_BLAKE2B_384_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.35.1 Definitions

CKM_BLAKE2B_384
CKM_BLAKE2B_384_HMAC
CKM_BLAKE2B_384_HMAC_GENERAL
CKM_BLAKE2B_384_KEY_DERIVE
CKM_BLAKE2B_384_KEY_GEN
CKK_BLAKE2B_384_HMAC

2.35.2 BLAKE2B-384 digest

The BLAKE2B-384 mechanism, denoted CKM_BLAKE2B_384, is a mechanism for message digesting, following the Blake2b Algorithm with a 384-bit message digest without a key as defined in RFC 7693. It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

<table>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>48</td>
</tr>
</tbody>
</table>

2.35.3 General-length BLAKE2B-384-HMAC

The general-length BLAKE2B-384-HMAC mechanism, denoted CKM_BLAKE2B_384_HMAC_GENERAL, is the keyed variant of the Blake2b-384 hash function and length of the output should be in the range 1-48. The keys it uses are generic secret keys and CKK_BLAKE2B_384_HMAC.

It has a parameter, a CK_MAC_GENERAL_PARAMS, which holds the length in bytes of the desired output. This length should be in the range 1-48 (the output size of BLAKE2B-384 is 48 bytes). Signatures (MACs) produced by this mechanism shall be taken from the start of the full 48-byte HMAC output.

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret or CKK_BLAKE2B_384_HMAC</td>
<td>Any</td>
<td>1-48, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret or CKK_BLAKE2B_384_HMAC</td>
<td>Any</td>
<td>1-48, depending on parameters</td>
</tr>
</tbody>
</table>

2.35.4 BLAKE2B-384-HMAC

The BLAKE2B-384-HMAC mechanism, denoted CKM_BLAKE2B_384_HMAC, is a special case of the general-length BLAKE2B-384-HMAC mechanism.

It has no parameter, and always produces an output of length 48.
2.35.5 BLAKE2B-384 key derivation

BLAKE2B-384 key derivation, denoted \texttt{CKM\_BLAKE2B\_384\_KEY\_DERIVE}, is the same as the SHA-1 key derivation mechanism in Section 2.20.5, except that it uses the SHA-384 hash function and the relevant length is 48 bytes.

2.35.6 BLAKE2B-384 HMAC key generation

The BLAKE2B-384-HMAC key generation mechanism, denoted \texttt{CKM\_BLAKE2B\_384\_KEY\_GEN}, is a key generation mechanism for NIST’s BLAKE2B-384-HMAC.

It does not have a parameter.

The mechanism generates BLAKE2B-384-HMAC keys with a particular length in bytes, as specified in the \texttt{CKA\_VALUE\_LEN} attribute of the template for the key.

The mechanism contributes the \texttt{CKA\_CLASS}, \texttt{CKA\_KEY\_TYPE}, and \texttt{CKA\_VALUE} attributes to the new key. Other attributes supported by the BLAKE2B-384-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the \texttt{ulMinKeySize} and \texttt{ulMaxKeySize} fields of the \texttt{CK\_MECHANISM\_INFO} structure specify the supported range of \texttt{CKM\_BLAKE2B\_384\_HMAC} key sizes, in bytes.

2.36 BLAKE2B-512

\textit{Table 149, SHA-512 Mechanisms vs. Functions}

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR¹</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_BLAKE2B_512</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_BLAKE2B_512_HMAC_GENERAL</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_BLAKE2B_512_HMAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_BLAKE2B_512_KEY_DERIVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_BLAKE2B_512_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.36.1 Definitions

- CKM\_BLAKE2B\_512
- CKM\_BLAKE2B\_512\_HMAC
- CKM\_BLAKE2B\_512\_HMAC\_GENERAL
- CKM\_BLAKE2B\_512\_KEY\_DERIVE
- CKM\_BLAKE2B\_512\_KEY\_GEN
- CKK\_BLAKE2B\_512\_HMAC

2.36.2 BLAKE2B-512 digest

The BLAKE2B-512 mechanism, denoted \texttt{CKM\_BLAKE2B\_512}, is a mechanism for message digesting, following the Blake2b Algorithm with a 512-bit message digest defined in RFC 7693.

It does not have a parameter.
Constraints on the length of input and output data are summarized in the following table. For single-part digests, the data and the digest may begin at the same location in memory.

<table>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>64</td>
</tr>
</tbody>
</table>

### 2.36.3 General-length BLAKE2B-512-HMAC

The general-length BLAKE2B-512-HMAC mechanism, denoted `CKM_BLAKE2B_512_HMAC_GENERAL`, is the keyed variant of the BLAKE2B-512 hash function and length of the output should be in the range 1-64. The keys it uses are generic secret keys and `CKK_BLAKE2B_512_HMAC`.

It has a parameter, a `CK_MAC_GENERAL_PARAMS`, which holds the length in bytes of the desired output. This length should be in the range 1-64 (the output size of BLAKE2B-512 is 64 bytes). Signatures (MACs) produced by this mechanism shall be taken from the start of the full 64-byte HMAC output.

### Table 151, General-length BLAKE2B-512-HMAC: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret or <code>CKK_BLAKE2B_512_HMAC</code></td>
<td>Any</td>
<td>1-64, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret or <code>CKK_BLAKE2B_512_HMAC</code></td>
<td>Any</td>
<td>1-64, depending on parameters</td>
</tr>
</tbody>
</table>

### 2.36.4 BLAKE2B-512-HMAC

The BLAKE2B-512-HMAC mechanism, denoted `CKM_BLAKE2B_512_HMAC`, is a special case of the general-length BLAKE2B-512-HMAC mechanism.

It has no parameter, and always produces an output of length 64.

### 2.36.5 BLAKE2B-512 key derivation

BLAKE2B-512 key derivation, denoted `CKM_BLAKE2B_512_KEY_DERIVE`, is the same as the SHA-1 key derivation mechanism in Section2.20.5, except that it uses the Blake2b-512 hash function and the relevant length is 64 bytes.

### 2.36.6 BLAKE2B-512 HMAC key generation

The BLAKE2B-512-HMAC key generation mechanism, denoted `CKM_BLAKE2B_512_KEY_GEN`, is a key generation mechanism for NIST’s BLAKE2B-512-HMAC.

It does not have a parameter.

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.

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.

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.
2.37 PKCS #5 and PKCS #5-style password-based encryption (PBE)

The mechanisms in this section are for generating keys and IVs for performing password-based encryption. The method used to generate keys and IVs is specified in PKCS #5.

Table 152, PKCS 5 Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_PBE_SHA1_DES3_EDE_CBC</td>
<td>Encrypt &amp; Decrypt, Sign &amp; Verify, Gen. Key/ Key Pair, Wrap &amp; Unwrap, Derive</td>
</tr>
<tr>
<td>CKM_PBE_SHA1_DES2_EDE_CBC</td>
<td></td>
</tr>
<tr>
<td>CKM_PBA_SHA1_WITH_SHA1_HMAC</td>
<td></td>
</tr>
<tr>
<td>CKM_PKCS5_PBKD2</td>
<td></td>
</tr>
</tbody>
</table>

2.37.1 Definitions

Mechanisms:

- CKM_PBE_SHA1_DES3_EDE_CBC
- CKM_PBE_SHA1_DES2_EDE_CBC
- CKM_PKCS5_PBKD2
- CKM_PBA_SHA1_WITH_SHA1_HMAC

2.37.2 Password-based encryption/authentication mechanism parameters

**CK_PBE_PARAMS; CK_PBE_PARAMS_PTR**

**CK_PBE_PARAMS** is a structure which provides all of the necessary information required by the CKM_PBE mechanisms (see PKCS #5 and PKCS #12 for information on the PBE generation mechanisms) and the CKM_PBA_SHA1_WITH_SHA1_HMAC mechanism. It is defined as follows:

```c
typedef struct CK_PBE_PARAMS {
    CK_BYTE_PTR pInitVector;
    CK_UTF8CHAR_PTR pPassword;
    CK_ULONG ulPasswordLen;
    CK_BYTE_PTR pSalt;
    CK_ULONG ulSaltLen;
    CK_ULONG ulIteration;
} CK_PBE_PARAMS;
```

The fields of the structure have the following meanings:

- `pInitVector` pointer to the location that receives the 8-byte initialization vector (IV), if an IV is required;
- `pPassword` points to the password to be used in the PBE key generation;
- `ulPasswordLen` length in bytes of the password information;
pSalt points to the salt to be used in the PBE key generation;

ulSaltLen length in bytes of the salt information;

ullIteration number of iterations required for the generation.

CK_PBE_PARAMS_PTR is a pointer to a CK_PBE_PARAMS.

### 2.37.3 PKCS #5 PBKDF2 key generation mechanism parameters

- **CK_PKCS5_PBKD2_PSEUDORANDOM_FUNCTION_TYPE;**
- **CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE_PTR**

**CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE** is used to indicate the Pseudo-Random Function (PRF) used to generate key bits using PKCS #5 PBKDF2. It is defined as follows:

```c
typedef CK_ULONG CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE;
```

The following PRFs are defined in PKCS #5 v2.1. The following table lists the defined functions.

<table>
<thead>
<tr>
<th>PRF Identifier</th>
<th>Value</th>
<th>Parameter Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CK_PKCS5_PBKD2_HMAC_SHA1</td>
<td>0x00000001UL</td>
<td>No Parameter. pPrfData must be NULL and ulPrfDataLen must be zero.</td>
</tr>
<tr>
<td>CK_PKCS5_PBKD2_HMAC_GOSTR3411</td>
<td>0x00000002UL</td>
<td>This PRF uses GOST R34.11-94 hash to produce secret key value. pPrfData should point to DER-encoded OID, indicating GOSTR34.11-94 parameters. ulPrfDataLen holds encoded OID length in bytes. If pPrfData is set to NULL_PTR, then id-GostR3411-94-CryptoProParamSet parameters will be used (RFC 4357, 11.2), and ulPrfDataLen must be 0.</td>
</tr>
<tr>
<td>CK_PKCS5_PBKD2_HMAC_SHA224</td>
<td>0x00000003UL</td>
<td>No Parameter. pPrfData must be NULL and ulPrfDataLen must be zero.</td>
</tr>
<tr>
<td>CK_PKCS5_PBKD2_HMAC_SHA256</td>
<td>0x00000004UL</td>
<td>No Parameter. pPrfData must be NULL and ulPrfDataLen must be zero.</td>
</tr>
<tr>
<td>CK_PKCS5_PBKD2_HMAC_SHA384</td>
<td>0x00000005UL</td>
<td>No Parameter. pPrfData must be NULL and ulPrfDataLen must be zero.</td>
</tr>
<tr>
<td>CK_PKCS5_PBKD2_HMAC_SHA512</td>
<td>0x00000006UL</td>
<td>No Parameter. pPrfData must be NULL and ulPrfDataLen must be zero.</td>
</tr>
<tr>
<td>CK_PKCS5_PBKD2_HMAC_SHA512_224</td>
<td>0x00000007UL</td>
<td>No Parameter. pPrfData must be NULL and ulPrfDataLen must be zero.</td>
</tr>
</tbody>
</table>
CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE_PTR is a pointer to a
CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE.

♣ CK_PKCS5_PBKD2_SALT_SOURCE_TYPE;
CK_PKCS5_PBKD2_SALT_SOURCE_TYPE_PTR

CK_PKCS5_PBKD2_SALT_SOURCE_TYPE is used to indicate the source of the salt value when
deriving a key using PKCS #5 PBKDF2. It is defined as follows:

typedef CK_ULONG CK_PKCS5_PBKD2_SALT_SOURCE_TYPE;

The following salt value sources are defined in PKCS #5 v2.1. The following table lists the defined
sources along with the corresponding data type for the pSaltSourceData field in the
CK_PKCS5_PBKD2_PARAM structure defined below.

Table 154, PKCS #5 PBKDF2 Key Generation: Salt sources

<table>
<thead>
<tr>
<th>Source Identifier</th>
<th>Value</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKZ_SALT_SPECIFIED</td>
<td>0x00000001</td>
<td>Array of CK_BYTE containing the value of the salt value.</td>
</tr>
</tbody>
</table>

CK_PKCS5_PBKD2_SALT_SOURCE_TYPE_PTR is a pointer to a
CK_PKCS5_PBKD2_SALT_SOURCE_TYPE.

♣ CK_PKCS5_PBKD2_PARAMS; CK_PKCS5_PBKD2_PARAMS_PTR

CK_PKCS5_PBKD2_PARAMS is a structure that provides the parameters to the CKM_PKCS5_PBKD2
mechanism. The structure is defined as follows:

typedef struct CK_PKCS5_PBKD2_PARAMS {
    CK_PKCS5_PBKD2_SALT_SOURCE_TYPE_ptr saltSource;
    CK_VOID_PTR pSaltSourceData;
    CK_ULONG ulSaltSourceDataLen;
    CK_ULONG iterations;
    CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE prf;
    CK_VOID_PTR pPrfData;
    CK_ULONG ulPrfDataLen;
    CK_UTF8CHAR_PTR pPassword;
    CK_ULONG_PTR ulPasswordLen;
} CK_PKCS5_PBKD2_PARAMS;

The fields of the structure have the following meanings:

saltSource source of the salt value
pSaltSourceData data used as the input for the salt source
ulSaltSourceDataLen length of the salt source input
iterations  number of iterations to perform when generating each block of random data

prf  pseudo-random function used to generate the key

pPrfData  data used as the input for PRF in addition to the salt value

ulPrfDataLen  length of the input data for the PRF

pPassword  points to the password to be used in the PBE key generation

ulPasswordLen  length in bytes of the password information

CK_PKCS5_PBKD2_PARAMS_PTR  is a pointer to a CK_PKCS5_PBKD2_PARAMS.

2.37.4 PKCS #5 PBKDF2 key generation

PKCS #5 PBKDF2 key generation, denoted CKM_PKCS5_PBKD2, is a mechanism used for generating a secret key from a password and a salt value. This functionality is defined in PKCS#5 as PBKDF2.

It has a parameter, a CK_PKCS5_PBKD2_PARAMS structure. The parameter specifies the salt value source, pseudo-random function, and iteration count used to generate the new key.

Since this mechanism can be used to generate any type of secret key, new key templates must contain the CKA_KEY_TYPE and CKA_VALUE_LEN attributes. If the key type has a fixed length the CKA_VALUE_LEN attribute may be omitted.

2.38 PKCS #12 password-based encryption/authentication mechanisms

The mechanisms in this section are for generating keys and IVs for performing password-based encryption or authentication. The method used to generate keys and IVs is based on a method that was specified in PKCS #12.

We specify here a general method for producing various types of pseudo-random bits from a password, p; a string of salt bits, s; and an iteration count, c. The “type” of pseudo-random bits to be produced is identified by an identification byte, ID, the meaning of which will be discussed later.

Let H be a hash function built around a compression function $f : \mathbb{Z}_2^u \times \mathbb{Z}_2^v \rightarrow \mathbb{Z}_2^u$ (that is, H has a chaining variable and output of length $u$ bits, and the message input to the compression function of H is $v$ bits).

For MD2 and MD5, $u=128$ and $v=512$; for SHA-1, $u=160$ and $v=512$.

We assume here that $u$ and $v$ are both multiples of 8, as are the lengths in bits of the password and salt strings and the number $n$ of pseudo-random bits required. In addition, $u$ and $v$ are of course nonzero.

1. Construct a string, D (the “diversifier”), by concatenating $v/8$ copies of ID.
2. Concatenate copies of the salt together to create a string S of length $v\lceil s/v \rceil$ bits (the final copy of the salt may be truncated to create S). Note that if the salt is the empty string, then so is S.
3. Concatenate copies of the password together to create a string P of length $v\lceil p/v \rceil$ bits (the final copy of the password may be truncated to create P). Note that if the password is the empty string, then so is P.
4. Set $I=S||P$ to be the concatenation of S and P.
5. Set $j=\lceil n/u \rceil$.
6. For $i=1, 2, ..., j$, do the following:
   a. Set $A_i=H^i(D||I)$, the $c^\text{th}$ hash of $D||I$. That is, compute the hash of $D||I$; compute the hash of that hash; etc.; continue in this fashion until a total of c hashes have been computed, each on the result of the previous hash.
b. Concatenate copies of $A_i$ to create a string $B$ of length $v$ bits (the final copy of $A_i$ may be truncated to create $B$).

c. Treating $l$ as a concatenation $l_0, l_1, \ldots, l_{k-1}$ of $v$-bit blocks, where $k = \lceil s/v \rceil + \lceil p/v \rceil$, modify $l$ by setting $l = (i + B + 1) \mod 2^s$ for each $j$. To perform this addition, treat each $v$-bit block as a binary number represented most-significant bit first.

7. Concatenate $A_1, A_2, \ldots, A_j$ together to form a pseudo-random bit string, $A$.

8. Use the first $n$ bits of $A$ as the output of this entire process.

When the password-based encryption mechanisms presented in this section are used to generate a key and IV (if needed) from a password, salt, and an iteration count, the above algorithm is used. To generate a key, the identifier byte ID is set to the value 1; to generate an IV, the identifier byte ID is set to the value 2.

When the password based authentication mechanism presented in this section is used to generate a key from a password, salt, and an iteration count, the above algorithm is used. The identifier byte ID is set to the value 3.

2.38.1 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.

2.38.2 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.

2.38.3 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.
2.39 SSL

Table 155, SSL Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Encrypt &amp; Decrypt</td>
</tr>
<tr>
<td>CKM_SSL3_PRE_MASTER_KEY_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SSL3_MASTER_KEY_DERIVE</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SSL3_MASTER_KEY_DERIVE_DH</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SSL3_KEY_AND_MAC_DERIVE</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SSL3_MD5_MAC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SSL3_SHA1_MAC</td>
<td>✓</td>
</tr>
</tbody>
</table>

2.39.1 Definitions

Mechanisms:
- CKM_SSL3_PRE_MASTER_KEY_GEN
- CKM_SSL3_MASTER_KEY_DERIVE
- CKM_SSL3_KEY_AND_MAC_DERIVE
- CKM_SSL3_MASTER_KEY_DERIVE_DH
- CKM_SSL3_MD5_MAC
- CKM_SSL3_SHA1_MAC

2.39.2 SSL mechanism parameters

♦ CK_SSL3_RANDOM_DATA

CK_SSL3_RANDOM_DATA is a structure which provides information about the random data of a client and a server in an SSL context. This structure is used by both the CKM_SSL3_MASTER_KEY_DERIVE and the CKM_SSL3_KEY_AND_MAC_DERIVE mechanisms. It is defined as follows:

```c
typedef struct CK_SSL3_RANDOM_DATA {
    CK_BYTE_PTR pClientRandom;
    CK_ULONG ulClientRandomLen;
    CK_BYTE_PTR pServerRandom;
    CK_ULONG ulServerRandomLen;
} CK_SSL3_RANDOM_DATA;
```

The fields of the structure have the following meanings:
- `pClientRandom` pointer to the client’s random data
- `ulClientRandomLen` length in bytes of the client’s random data
- `pServerRandom` pointer to the server’s random data
ulServerRandomLen  length in bytes of the server's random data

- **CK_SSL3_MASTER_KEY_DERIVE_PARAMS;**
  CK_SSL3_MASTER_KEY_DERIVE_PARAMS_PTR

  CK_SSL3_MASTER_KEY_DERIVE_PARAMS is a structure that provides the parameters to the
  CKM_SSL3_MASTER_KEY_DERIVE mechanism. It is defined as follows:

  ```
  typedef struct CK_SSL3_MASTER_KEY_DERIVE_PARAMS {
    CK_SSL3_RANDOM_DATA  RandomInfo;
    CK_VERSION_PTR     pVersion;
  } CK_SSL3_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.

  CK_SSL3_MASTER_KEY_DERIVE_PARAMS_PTR is a pointer to a
  CK_SSL3_MASTER_KEY_DERIVE_PARAMS.

- **CK_SSL3_KEY_MAT_OUT; CK_SSL3_KEY_MAT_OUT_PTR**

  CK_SSL3_KEY_MAT_OUT is a structure that contains the resulting key handles and initialization vectors
  after performing a C_DeriveKey function with the CKM_SSL3_KEY_AND_MAC_DERIVE mechanism. It
  is defined as follows:

  ```
  typedef struct CK_SSL3_KEY_MAT_OUT {
    CK_OBJECT_HANDLE  hClientMacSecret;
    CK_OBJECT_HANDLE  hServerMacSecret;
    CK_OBJECT_HANDLE  hClientKey;
    CK_OBJECT_HANDLE  hServerKey;
    CK_BYTE_PTR       pIVClient;
    CK_BYTE_PTR       pIVServer;
  } CK_SSL3_KEY_MAT_OUT;
  ```

  The fields of the structure have the following meanings:

  - `hClientMacSecret` key handle for the resulting Client MAC Secret key
  - `hServerMacSecret` key handle for the resulting Server MAC Secret key
  - `hClientKey`     key handle for the resulting Client Secret key
  - `hServerKey`     key handle for the resulting Server Secret key
  - `pIVClient`      pointer to a location which receives the initialization vector (IV)
                       created for the client (if any)
  - `pIVServer`      pointer to a location which receives the initialization vector (IV)
                       created for the server (if any)

  CK_SSL3_KEY_MAT_OUT_PTR is a pointer to a CK_SSL3_KEY_MAT_OUT.
CK_SSL3_KEY_MAT_PARAMS; CK_SSL3_KEY_MAT_PARAMS_PTR

CK_SSL3_KEY_MAT_PARAMS is a structure that provides the parameters to the CKM_SSL3_KEY_AND_MAC_DERIVE mechanism. It is defined as follows:

```c
typedef struct CK_SSL3_KEY_MAT_PARAMS {
    CK_ULONG ulMacSizeInBits;
    CK_ULONG ulKeySizeInBits;
    CK_ULONG ulIVSizeInBits;
    CK_BBOOL bIsExport;
    CK_SSL3_RANDOM_DATA RandomInfo;
    CK_SSL3_KEY_MAT_OUT_PTR pReturnedKeyMaterial;
} CK_SSL3_KEY_MAT_PARAMS;
```

The fields of the structure have the following meanings:

- `ulMacSizeInBits`: the length (in bits) of the MACing keys agreed upon during the protocol handshake phase.
- `ulKeySizeInBits`: the length (in bits) of the secret keys agreed upon during the protocol handshake phase.
- `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.
- `bIsExport`: a Boolean value which indicates whether the keys have to be derived for an export version of the protocol.
- `RandomInfo`: client’s and server’s random data information.
- `pReturnedKeyMaterial`: points to a CK_SSL3_KEY_MAT_OUT structures which receives the handles for the keys generated and the IVs.

CK_SSL3_KEY_MAT_PARAMS_PTR is a pointer to a CK_SSL3_KEY_MAT_PARAMS.

2.39.3 Pre-master key generation

Pre-master key generation in SSL 3.0, denoted CKM_SSL3_PRE_MASTER_KEY_GEN, is a mechanism which generates a 48-byte generic secret key. It is used to produce the “pre_master” key used in SSL version 3.0 for RSA-like cipher suites.

It has one parameter, a CK_VERSION structure, which provides the client’s SSL version number.

The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, and CKA_VALUE attributes to the new key (as well as the CKA_VALUE_LEN attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.

The template sent along with this mechanism during a C_GenerateKey call may indicate that the object class is CKO_SECRET_KEY, the key type is CKK_GENERIC_SECRET, and the CKA_VALUE_LEN attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure both indicate 48 bytes.
2.39.4 Master key derivation

Master key derivation in SSL 3.0, denoted `CKM_SSL3_MASTER_KEY_DERIVE`, is a mechanism used to derive one 48-byte generic secret key from another 48-byte generic secret key. It is used to produce the "master_secret" key used in the SSL protocol from the "pre_master" key. This mechanism returns the value of the client version, which is built into the "pre_master" key as well as a handle to the derived "master_secret" key.

It has a parameter, a `CK_SSL3_MASTER_KEY_DERIVE_PARAMS` structure, which allows for the passing of random data to the token as well as the returning of the protocol version number which is part of the pre-master key. This structure is defined in Section 2.39.

The mechanism contributes the `CKA_CLASS`, `CKA_KEY_TYPE`, and `CKA_VALUE` attributes to the new key (as well as the `CKA_VALUE_LEN` attribute, if it is not supplied in the template). Other attributes may be specified in the template; otherwise they are assigned default values.

The template sent along with this mechanism during a `C_DeriveKey` call may indicate that the object class is `CKO_SECRET_KEY`, the key type is `CKK_GENERIC_SECRET`, and the `CKA_VALUE_LEN` attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

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

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

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure both indicate 48 bytes.

Note that the `CK_VERSION` structure pointed to by the `CK_SSL3_MASTER_KEY_DERIVE_PARAMS` structure’s `pVersion` field will be modified by the `C_DeriveKey` call. In particular, when the call returns, this structure will hold the SSL version associated with the supplied pre_master key.

Note that this mechanism is only useable for cipher suites that use a 48-byte "pre_master" secret with an embedded version number. This includes the RSA cipher suites, but excludes the Diffie-Hellman cipher suites.

2.39.5 Master key derivation for Diffie-Hellman

Master key derivation for Diffie-Hellman in SSL 3.0, denoted `CKM_SSL3_MASTER_KEY_DERIVE_DH`, is a mechanism used to derive one 48-byte generic secret key from another arbitrary length generic secret key. It is used to produce the "master_secret" key used in the SSL protocol from the "pre_master" key.

It has a parameter, a `CK_SSL3_MASTER_KEY_DERIVE_PARAMS` structure, which allows for the passing of random data to the token. This structure is defined in Section 2.39. The `pVersion` field of the structure must be set to NULL_PTR since the version number is not embedded in the "pre_master" key as it is for RSA-like cipher suites.

The mechanism contributes the `CKA_CLASS`, `CKA_KEY_TYPE`, and `CKA_VALUE` attributes to the new key (as well as the `CKA_VALUE_LEN` attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.
The template sent along with this mechanism during a C_DeriveKey call may indicate that the object class is CKO_SECRET_KEY, the key type is CKK_GENERIC_SECRET, and the CKA_VALUE_LEN attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

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

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

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure both indicate 48 bytes.

Note that this mechanism is only useable for cipher suites that do not use a fixed length 48-byte "pre_master" secret with an embedded version number. This includes the Diffie-Hellman cipher suites, but excludes the RSA cipher suites.

### 2.39.6 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 2.39.

This mechanism contributes to the creation of four distinct keys on the token and returns two IVs (if IVs are requested by the caller) back to the caller. The keys are all given an object class of CKO_SECRET_KEY.

The two MACing keys ("client_write_MAC_secret" and "server_write_MAC_secret") are always given a type of CKK_GENERIC_SECRET. They are flagged as valid for signing, verification, and derivation operations.

The other two keys ("client_write_key" and "server_write_key") are typed according to information found in the template sent along with this mechanism during a C_DeriveKey function call. By default, they are flagged as valid for encryption, decryption, and derivation operations.

IVs will be generated and returned if the ulIVSizeInBits field of the CK_SSL3_KEY_MAT_PARAMS field has a nonzero value. If they are generated, their length in bits will agree with the value in the ulIVSizeInBits field.

All four keys inherit the values of the CKA_SENSITIVE, CKA_ALWAYS_SENSITIVE, CKA_EXTRACTABLE, and CKA_NEVER_EXTRACTABLE attributes from the base key. The template provided to C_DeriveKey may not specify values for any of these attributes which differ from those held by the base key.

Note that the CK_SSL3_KEY_MAT_OUT structure pointed to by the CK_SSL3_KEY_MAT_PARAMS structure's pReturnedKeyMaterial field will be modified by the C_DeriveKey call. In particular, the four key handle fields in the CK_SSL3_KEY_MAT_OUT structure will be modified to hold handles to the newly-created keys; in addition, the buffers pointed to by the CK_SSL3_KEY_MAT_OUT structure's
pIVClient and pIVServer fields will have IVs returned in them (if IVs are requested by the caller).
Therefore, these two fields must point to buffers with sufficient space to hold any IVs that will be returned.
This mechanism departs from the other key derivation mechanisms in Cryptoki in its returned information.
For most key-derivation mechanisms, C_DeriveKey returns a single key handle as a result of a successful completion. However, since the CKM_SSL3_KEY_AND_MAC_DERIVE mechanism returns all of its key handles in the CK_SSL3_KEY_MAT_OUT structure pointed to by the CK_SSL3_KEY_MAT_PARAMS structure specified as the mechanism parameter, the parameter phKey passed to C_DeriveKey is unnecessary, and should be a NULL_PTR.
If a call to C_DeriveKey with this mechanism fails, then none of the four keys will be created on the token.

2.39.7 MD5 MACing in SSL 3.0
MD5 MACing in SSL3.0, denoted CKM_SSL3_MD5_MAC, is a mechanism for single- and multiple-part signatures (data authentication) and verification using MD5, based on the SSL 3.0 protocol. This technique is very similar to the HMAC technique.
It has a parameter, a CK_MAC_GENERAL_PARAMS, which specifies the length in bytes of the signatures produced by this mechanism.
Constraints on key types and the length of input and output data are summarized in the following table:

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret</td>
<td>any</td>
<td>4-8, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret</td>
<td>any</td>
<td>4-8, depending on parameters</td>
</tr>
</tbody>
</table>

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

2.39.8 SHA-1 MACing in SSL 3.0
SHA-1 MACing in SSL3.0, denoted CKM_SSL3_SHA1_MAC, is a mechanism for single- and multiple-part signatures (data authentication) and verification using SHA-1, based on the SSL 3.0 protocol. This technique is very similar to the HMAC technique.
It has a parameter, a CK_MAC_GENERAL_PARAMS, which specifies the length in bytes of the signatures produced by this mechanism.
Constraints on key types and the length of input and output data are summarized in the following table:

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret</td>
<td>any</td>
<td>4-8, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret</td>
<td>any</td>
<td>4-8, depending on parameters</td>
</tr>
</tbody>
</table>

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

2.40 TLS 1.2 Mechanisms
Details for TLS 1.2 and its key derivation and MAC mechanisms can be found in [TLS12]. TLS 1.2 mechanisms differ from TLS 1.0 and 1.1 mechanisms in that the base hash used in the underlying TLS PRF (pseudo-random function) can be negotiated. Therefore each mechanism parameter for the TLS 1.2 mechanisms contains a new value in the parameters structure to specify the hash function.
This section also specifies CKM_TLS12_MAC which should be used in place of CKM_TLS_PRF to calculate the verify_data in the TLS “finished” message.

This section also specifies CKM_TLS_KDF that can be used in place of CKM_TLS_PRF to implement key material exporters.

Table 158, TLS 1.2 Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR¹</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_TLS12_MASTER_KEY_DERIVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_TLS12_MASTER_KEY_DERIVE_DH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_TLS12_KEY_AND_MAC_DERIVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_TLS12_KEY_SAFE_DERIVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_TLS_KDF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_TLS12_MAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_TLS12_KDF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

2.40.1 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

2.40.2 TLS 1.2 mechanism parameters

- CK_TTLS12_MASTER_KEY_DERIVE_PARAMS;
- CK_TTLS12_MASTER_KEY_DERIVE_PARAMS_PTR

CK_TTLS12_MASTER_KEY_DERIVE_PARAMS is a structure that provides the parameters to the CK_TTLS12_MASTER_KEY_DERIVE mechanism. It is defined as follows:

typedef struct CK_TTLS12_MASTER_KEY_DERIVE_PARAMS {
    CK_SSL3_RANDOM_DATA RandomInfo;
    CK_VERSION_PTR pVersion;
    CK_MECHANISM_TYPE prfHashMechanism;
} CK_TTLS12_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.

♦ CK_TLS12_KEY_MAT_PARAMS; CK_TLS12_KEY_MAT_PARAMS_PTR

CK_TLS12_KEY_MAT_PARAMS is a structure that provides the parameters to the CKM_TLS12_KEY_AND_MAC_DERIVE mechanism. It is defined as follows:

```c
typedef struct CK_TLS12_KEY_MAT_PARAMS {
    CK_ULONG ulMacSizeInBits;
    CK_ULONG ulKeySizeInBits;
    CK_ULONG ulIVSizeInBits;
    CK_BBOOL bIsExport;
    CK_SSL3_RANDOM_DATA RandomInfo;
    CK_SSL3_KEY_MAT_OUT_PTR pReturnedKeyMaterial;
    CK_MECHANISM_TYPE prfHashMechanism;
} CK_TLS12_KEY_MAT_PARAMS;
```

The fields of the structure have the following meanings:

- `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.

- `ulKeySizeInBits` the length (in bits) of the secret keys agreed upon during the protocol handshake phase.

- `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.

- `bIsExport` must be set to CK_FALSE because export cipher suites must not be used in TLS 1.1 and later.

- `RandomInfo` client’s and server’s random data information.

- `pReturnedKeyMaterial` points to a CK_SSL3_KEY_MAT_OUT_PTR structure which receives the handles for the keys generated and the IVs.

- `prfHashMechanism` base hash used in the underlying TLS1.2 PRF operation used to derive the master key.

CK_TLS12_KEY_MAT_PARAMS_PTR is a pointer to a CK_TLS12_KEY_MAT_PARAMS.

♦ CK_TLS_KDF_PARAMS; CK_TLS_KDF_PARAMS_PTR

CK_TLS_KDF_PARAMS is a structure that provides the parameters to the CKM_TLS_KDF mechanism. It is defined as follows:
typedef struct CK_TLS_KDF_PARAMS {
    CK_MECHANISM_TYPE prfMechanism;
    CK_BYTE_PTR pLabel;
    CK_ULONG ulLabelLength;
    CK_SSL3_RANDOM_DATA RandomInfo;
    CK_BYTE_PTR pContextData;
    CK_ULONG ulContextDataLength;
} CK_TLS_KDF_PARAMS;

The fields of the structure have the following meanings:

- **prfMechanism** - the hash mechanism used in the TLS1.2 PRF construct or CKM_TLS_PRF to use with the TLS1.0 and 1.1 PRF construct.
- **pLabel** - a pointer to the label for this key derivation
- **ulLabelLength** - length of the label in bytes
- **RandomInfo** - the random data for the key derivation
- **pContextData** - a pointer to the context data for this key derivation. NULL_PTR if not present
- **ulContextDataLength** - length of the context data in bytes. 0 if not present.

♦ **CK_TLS_MAC_PARAMS; CK_TLS_MAC_PARAMS_PTR**

**CK_TLS_MAC_PARAMS** is a structure that provides the parameters to the CKM_TLS_MAC mechanism. It is defined as follows:

typedef struct CK_TLS_MAC_PARAMS {
    CK_MECHANISM_TYPE prfMechanism;
    CK_ULONG ulMacLength;
    CK_ULONG ulServerOrClient;
} CK_TLS_MAC_PARAMS;

The fields of the structure have the following meanings:

- **prfMechanism** - the hash mechanism used in the TLS12 PRF construct or CKM_TLS_PRF to use with the TLS1.0 and 1.1 PRF construct.
- **ulMacLength** - the length of the MAC tag required or offered. Always 12 octets in TLS 1.0 and 1.1. Generally 12 octets, but may be negotiated to a longer value in TLS1.2.
- **ulServerOrClient** - 1 to use the label "server finished", 2 to use the label "client finished". All other values are invalid.

**CK_TLS_MAC_PARAMS_PTR** is a pointer to a **CK_TLS_MAC_PARAMS**.
CK_TLS_PRF_PARAMS is a structure, which provides the parameters to the CKM_TLS_PRF mechanism. It is defined as follows:

```c
typedef struct CK_TLS_PRF_PARAMS {
    CK_BYTE_PTR       pSeed;
    CK_ULONG          ulSeedLen;
    CK_BYTE_PTR       pLabel;
    CK_ULONG          ulLabelLen;
    CK_BYTE_PTR       pOutput;
    CK_ULONG_PTR      pulOutputLen;
} CK_TLS_PRF_PARAMS;
```

The fields of the structure have the following meanings:

- `pSeed` pointer to the input seed
- `ulSeedLen` length in bytes of the input seed
- `pLabel` pointer to the identifying label
- `ulLabelLen` length in bytes of the identifying label
- `pOutput` pointer receiving the output of the operation
- `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

CK_TLS_PRF_PARAMS_PTR is a pointer to a CK_TLS_PRF_PARAMS.

### 2.40.3 TLS MAC

The TLS MAC mechanism is used to generate integrity tags for the TLS “finished” message. It replaces the use of the CKM_TLS_PRF function for TLS1.0 and 1.1 and that mechanism is deprecated.

CKM_TLS_MAC takes a parameter of CK_TLS_MAC_PARAMS. To use this mechanism with TLS1.0 and TLS1.1, use CKM_TLS_PRF as the value for `prfMechanism` in place of a hash mechanism. Note: Although CKM_TLS_PRF is deprecated as a mechanism for C_DeriveKey, the manifest value is retained for use with this mechanism to indicate the use of the TLS1.0/1.1 pseudo-random function.

In TLS1.0 and 1.1 the “finished” message verify_data (i.e. the output signature from the MAC mechanism) is always 12 bytes. In TLS1.2 the “finished” message verify_data is a minimum of 12 bytes, defaults to 12 bytes, but may be negotiated to longer length.

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret</td>
<td>any</td>
<td>&gt;=12 bytes</td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret</td>
<td>any</td>
<td>&gt;=12 bytes</td>
</tr>
</tbody>
</table>

### 2.40.4 Master key derivation

Master key derivation in TLS 1.0, denoted CKM_TLS_MASTER_KEY_DERIVE, is a mechanism used to derive one 48-byte generic secret key from another 48-byte generic secret key. It is used to produce the
"master_secret" key used in the TLS protocol from the "pre_master" key. This mechanism returns the value of the client version, which is built into the "pre_master" key as well as a handle to the derived "master_secret" key.

It has a parameter, a CK_SSL3_MASTER_KEY_DERIVE_PARAMS structure, which allows for the passing of random data to the token as well as the returning of the protocol version number which is part of the pre-master key. This structure is defined in Section 2.39.

The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, and CKA_VALUE attributes to the new key (as well as the CKA_VALUE_LEN attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.

The mechanism also contributes the CKA_ALLOWED_MECHANISMS attribute consisting only of CKM_TLS12_KEY_SAFE_DERIVE, CKM_TLS12_KDF and CKM_TLS12_MAC.

The template sent along with this mechanism during a C_DeriveKey call may indicate that the object class is CKO_SECRET_KEY, the key type is CKK_GENERIC_SECRET, and the CKA_VALUE_LEN attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

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

- The CKA_SENSITIVE and CKA_EXTRACTABLE attributes in the template for the new key can both be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some default value.

- If the base key has its CKA_ALWAYS_SENSITIVE attribute set to CK_FALSE, then the derived key will as well. If the base key has its CKA_ALWAYS_SENSITIVE attribute set to CK_TRUE, then the derived key has its CKA_ALWAYS_SENSITIVE attribute set to the same value as its CKA_SENSITIVE attribute.

- Similarly, if the base key has its CKA_NEVER_EXTRACTABLE attribute set to CK_FALSE, then the derived key will, too. If the base key has its CKA_NEVER_EXTRACTABLE attribute set to CK_TRUE, then the derived key has its CKA_NEVER_EXTRACTABLE attribute set to the opposite value from its CKA_EXTRACTABLE attribute.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure both indicate 48 bytes.

Note that the CK_VERSION structure pointed to by the CK_SSL3_MASTER_KEY_DERIVE_PARAMS structure’s pVersion field will be modified by the C_DeriveKey call. In particular, when the call returns, this structure will hold the SSL version associated with the supplied pre_master key.

Note that this mechanism is only usable for cipher suites that use a 48-byte "pre_master" secret with an embedded version number. This includes the RSA cipher suites, but excludes the Diffie-Hellman cipher suites.

### 2.40.5 Master key derivation for Diffie-Hellman

Master key derivation for Diffie-Hellman in TLS 1.0, denoted CKM_TLS_MASTER_KEY_DERIVE_DH, is a mechanism used to derive one 48-byte generic secret key from another arbitrary length generic secret key. It is used to produce the "master_secret" key used in the TLS protocol from the "pre_master" key.

It has a parameter, a CK_SSL3_MASTER_KEY_DERIVE_PARAMS structure, which allows for the passing of random data to the token. This structure is defined in Section 2.39. The pVersion field of the structure must be set to NULL_PTR since the version number is not embedded in the "pre_master" key as it is for RSA-like cipher suites.

The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, and CKA_VALUE attributes to the new key (as well as the CKA_VALUE_LEN attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.

The mechanism also contributes the CKA_ALLOWED_MECHANISMS attribute consisting only of CKM_TLS12_KEY_AND_MAC_DERIVE, CKM_TLS12_KEY_SAFE_DERIVE, CKM_TLS12_KDF and CKM_TLS12_MAC.
The template sent along with this mechanism during a C_DeriveKey call may indicate that the object class is CKO_SECRET_KEY, the key type is CKK_GENERIC_SECRET, and the CKA_VALUE_LEN attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

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

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

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure both indicate 48 bytes.

Note that this mechanism is only useable for cipher suites that do not use a fixed length 48-byte "pre_master" secret with an embedded version number. This includes the Diffie-Hellman cipher suites, but excludes the RSA cipher suites.

2.40.6 Key and MAC derivation

Key, MAC and IV derivation in TLS 1.0, denoted CKM_TLS_KEY_AND_MAC_DERIVE, is a mechanism used to derive the appropriate cryptographic keying material used by a "CipherSuite" from the "master_secret" key and random data. This mechanism returns the key handles for the keys generated in the process, as well as the IVs created.

It has a parameter, a CK_SSL3_KEY_MAT_PARAMS structure, which allows for the passing of random data as well as the characteristic of the cryptographic material for the given CipherSuite and a pointer to a structure which receives the handles and IVs which were generated. This structure is defined in Section 2.39.

This mechanism contributes to the creation of four distinct keys on the token and returns two IVs (if IVs are requested by the caller) back to the caller. The keys are all given an object class of CKO_SECRET_KEY.

The two MACing keys ("client_write_MAC_secret" and "server_write_MAC_secret") (if present) are always given a type of CKK_GENERIC_SECRET. They are flagged as valid for signing and verification.

The other two keys ("client_write_key" and "server_write_key") are typed according to information found in the template sent along with this mechanism during a C_DeriveKey function call. By default, they are flagged as valid for encryption, decryption, and derivation operations.

For CKM_TLS12_KEY_AND_MAC_DERIVE, IVs will be generated and returned if the ulIVSizeInBits field of the CK_SSL3_KEY_MAT_PARAMS field has a nonzero value. If they are generated, their length in bits will agree with the value in the ulIVSizeInBits field.

Note Well: CKM_TLS12_KEY_AND_MAC_DERIVE produces both private (key) and public (IV) data. It is possible to "leak" private data by the simple expedient of decreasing the length of private data requested. E.g. Setting ulMacSizeInBits and ulKeySizeInBits to 0 (or other lengths less than the key size) will result in the private key data being placed in the destination designated for the IV’s. Repeated calls with the same master key and same RandomInfo but with differing lengths of the private key material will result in different data being leaked.
All four keys inherit the values of the CKA_SENSITIVE, CKA_ALWAYS_SENSITIVE,
CKA_EXTRACTABLE, and CKA_NEVER_EXTRACTABLE attributes from the base key. The template
provided to C_DeriveKey may not specify values for any of these attributes which differ from those held
by the base key.

Note that the CK_SSL3_KEY_MAT_OUT structure pointed to by the CK_SSL3_KEY_MAT_PARAMS
structure's pReturnedKeyMaterial field will be modified by the C_DeriveKey call. In particular, the four
key handle fields in the CK_SSL3_KEY_MAT_OUT structure will be modified to hold handles to the
newly-created keys; in addition, the buffers pointed to by the CK_SSL3_KEY_MAT_OUT structure's
pIVClient and pIVServer fields will have IVs returned in them (if IVs are requested by the caller).
Therefore, these two fields must point to buffers with sufficient space to hold any IVs that will be returned.

This mechanism departs from the other key derivation mechanisms in Cryptoki in its returned information.
For most key-derivation mechanisms, C_DeriveKey returns a single key handle as a result of a
successful completion. However, since the CKM_SSL3_KEY_AND_MAC_DERIVE mechanism returns
all of its key handles in the CK_SSL3_KEY_MAT_OUT structure pointed to by the
CK_SSL3_KEY_MAT_PARAMS structure specified as the mechanism parameter, the parameter phKey
passed to C_DeriveKey is unnecessary, and should be a NULL_PTR.

If a call to C_DeriveKey with this mechanism fails, then none of the four keys will be created on the
token.

2.40.7 CKM_TLS12_KEY_SAFE_DERIVE

CKM_TLS12_KEY_SAFE_DERIVE is identical to CKM_TLS12_KEY_AND_MAC_DERIVE except that it
shall never produce IV data, and the ulIVSizeInBits field of CK_TLS12_KEY_MAT_PARAMS is ignored
and treated as 0. All of the other conditions and behavior described for
CKM_TLS12_KEY_AND_MAC_DERIVE, with the exception of the black box warning, apply to this
mechanism.

CKM_TLS12_KEY_SAFE_DERIVE is provided as a separate mechanism to allow a client to control the
export of IV material (and possible leaking of key material) through the use of the
CKA_ALLOWED_MECHANISMS key attribute.

2.40.8 Generic Key Derivation using the TLS PRF

CKM_TLS_KDF is the mechanism defined in [RFC 5705]. It uses the TLS key material and TLS PRF
function to produce additional key material for protocols that want to leverage the TLS key negotiation
mechanism. CKM_TLS_KDF has a parameter of CK_TLS_KDF_PARAMS. If the protocol using this
mechanism does not use context information, the pContextData field shall be set to NULL_PTR and the
ulContextDataLength field shall be set to 0.

To use this mechanism with TLS1.0 and TLS1.1, use CKM_TLS_PRF as the value for prfMechanism in
place of a hash mechanism. Note: Although CKM_TLS_PRF is deprecated as a mechanism for
C_DeriveKey, the manifest value is retained for use with this mechanism to indicate the use of the
TLS1.0/1.1 Pseudo-random function.

This mechanism can be used to derive multiple keys (e.g. similar to
CKM_TLS12_KEY_AND_MAC_DERIVE) by first deriving the key stream as a CKK_GENERIC_SECRET
of the necessary length and doing subsequent derives against that derived key using the
CKM_EXTRACT_KEY_FROM_KEY mechanism to split the key stream into the actual operational keys.

The mechanism should not be used with the labels defined for use with TLS, but the token does not
enforce this behavior.

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

- If the original key has its CKA_SENSITIVE attribute set to CK_TRUE, so does the derived key. If not,
  then the derived key's CKA_SENSITIVE attribute is set either from the supplied template or from the
  original key.
Similarly, if the original key has its `CKA_EXTRACTABLE` attribute set to CK_FALSE, so does the derived key. If not, then the derived key’s `CKA_EXTRACTABLE` attribute is set either from the supplied template or from the original key.

The derived key’s `CKA_ALWAYS_SENSITIVE` attribute is set to CK_TRUE if and only if the original key has its `CKA_ALWAYS_SENSITIVE` attribute set to CK_TRUE.

Similarly, the derived key’s `CKA_NEVER_EXTRACTABLE` attribute is set to CK_TRUE if and only if the original key has its `CKA_NEVER_EXTRACTABLE` attribute set to CK_TRUE.

### 2.40.9 Generic Key Derivation using the TLS12 PRF

`CKM_TLS12_KDF` is the mechanism defined in [RFC 5705]. It uses the TLS key material and TLS PRF function to produce additional key material for protocols that want to leverage the TLS key negotiation mechanism. `CKM_TLS12_KDF` has a parameter of `CK_TLS_KDF_PARAMS`. If the protocol using this mechanism does not use context information, the `pContextData` field shall be set to NULL_PTR and the `ulContextDataLength` field shall be set to 0.

To use this mechanism with TLS1.0 and TLS1.1, use `CKM_TLS_PRF` as the value for `prfMechanism` in place of a hash mechanism. Note: Although `CKM_TLS_PRF` is deprecated as a mechanism for `C_DeriveKey`, the manifest value is retained for use with this mechanism to indicate the use of the TLS1.0/1.1 Pseudo-random function.

This mechanism can be used to derive multiple keys (e.g. similar to `CKM_TLS12_KEY_AND_MAC_DERIVE`) by first deriving the key stream as a `CKK_GENERIC_SECRET` of the necessary length and doing subsequent derives against that derived key stream using the `CKM_EXTRACT_KEY_FROM_KEY` mechanism to split the key stream into the actual operational keys.

The mechanism should not be used with the labels defined for use with TLS, but the token does not enforce this behavior.

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

- If the original key has its `CKA_SENSITIVE` attribute set to CK_TRUE, so does the derived key. If not, then the derived key’s `CKA_SENSITIVE` attribute is set either from the supplied template or from the original key.

- Similarly, if the original key has its `CKA_EXTRACTABLE` attribute set to CK_FALSE, so does the derived key. If not, then the derived key’s `CKA_EXTRACTABLE` attribute is set either from the supplied template or from the original key.

- The derived key’s `CKA_ALWAYS_SENSITIVE` attribute is set to CK_TRUE if and only if the original key has its `CKA_ALWAYS_SENSITIVE` attribute set to CK_TRUE.

- Similarly, the derived key’s `CKA_NEVER_EXTRACTABLE` attribute is set to CK_TRUE if and only if the original key has its `CKA_NEVER_EXTRACTABLE` attribute set to CK_TRUE.

### 2.41 WTLS

Details can be found in [WTLS].

When comparing the existing TLS mechanisms with these extensions to support WTLS one could argue that there would be no need to have distinct handling of the client and server side of the handshake. However, since in WTLS the server and client use different sequence numbers, there could be instances (e.g. when WTLS is used to protect asynchronous protocols) where sequence numbers on the client and server side differ, and hence this motivates the introduced split.

Table 160, WTLS Mechanisms vs. Functions
<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Encrypt &amp; Decrypt</td>
</tr>
<tr>
<td>CKM_WTLS_PRE_MASTER_KEY_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_WTLS_MASTER_KEY_DERIVE</td>
<td></td>
</tr>
<tr>
<td>CKM_WTLS_MASTER_KEY_DERIVE_DH_ECC</td>
<td></td>
</tr>
<tr>
<td>CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE</td>
<td></td>
</tr>
<tr>
<td>CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE</td>
<td></td>
</tr>
<tr>
<td>CKM_WTLS_PRF</td>
<td></td>
</tr>
</tbody>
</table>

### 2.41.1 Definitions

#### Mechanisms:

- CKM_WTLS_PRE_MASTER_KEY_GEN
- CKM_WTLS_MASTER_KEY_DERIVE
- CKM_WTLS_MASTER_KEY_DERIVE_DH_ECC
- CKM_WTLS_PRF
- CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE
- CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE

### 2.41.2 WTLS mechanism parameters

#### CK_WTLS_RANDOM_DATA; CK_WTLS_RANDOM_DATA_PTR

**CK_WTLS_RANDOM_DATA** is a structure, which provides information about the random data of a client and a server in a WTLS context. This structure is used by the **CKM_WTLS_MASTER_KEY_DERIVE** mechanism. It is defined as follows:

```c
typedef struct CK_WTLS_RANDOM_DATA {
    CK_BYTE_PTR pClientRandom;
    CK_ULONG ulClientRandomLen;
    CK_BYTE_PTR pServerRandom;
    CK_ULONG ulServerRandomLen;
} CK_WTLS_RANDOM_DATA;
```

The fields of the structure have the following meanings:

- `pClientRandom` pointer to the client’s random data
- `ulClientRandomLen` length in bytes of the client’s random data
- `pServerRandom` pointer to the server’s random data
ulServerRandomLen   length in bytes of the server's random data

CK_WTLS_RANDOM_DATA_PTR is a pointer to a CK_WTLS_RANDOM_DATA.

♦ CK_WTLS_MASTER_KEY_DERIVE_PARAMS;
   CK_WTLS_MASTER_KEY_DERIVE_PARAMS_PTR

CK_WTLS_MASTER_KEY_DERIVE_PARAMS is a structure, which provides the parameters to the
CKM_WTLS_MASTER_KEY_DERIVE mechanism. It is defined as follows:

typedef struct CK_WTLS_MASTER_KEY_DERIVE_PARAMS {
    CK_MECHANISM_TYPE   DigestMechanism;
    CK_WTLS_RANDOM_DATA RandomInfo;
    CK_BYTE_PTR         pVersion;
} CK_WTLS_MASTER_KEY_DERIVE_PARAMS;

The fields of the structure have the following meanings:

DigestMechanism    the mechanism type of the digest mechanism to be used (possible
types can be found in [WTLS])

RandomInfo         Client's and server's random data information

pVersion            pointer to a CK_BYTE which receives the WTLS protocol version
                     information

CK_WTLS_MASTER_KEY_DERIVE_PARAMS_PTR is a pointer to a
CK_WTLS_MASTER_KEY_DERIVE_PARAMS.

♦ CK_WTLS_PRF_PARAMS; CK_WTLS_PRF_PARAMS_PTR

CK_WTLS_PRF_PARAMS is a structure, which provides the parameters to the CKM_WTLS_PRF
mechanism. It is defined as follows:

typedef struct CK_WTLS_PRF_PARAMS {
    CK_MECHANISM_TYPE DigestMechanism;
    CK_BYTE_PTR       pSeed;
    CK_ULONG          ulSeedLen;
    CK_BYTE_PTR       pLabel;
    CK_ULONG          ulLabelLen;
    CK_BYTE_PTR       pOutput;
    CK_ULONG_PTR      pulOutputLen;
} CK_WTLS_PRF_PARAMS;

The fields of the structure have the following meanings:

Digest Mechanism    the mechanism type of the digest mechanism to be used (possible
types can be found in [WTLS])

pSeed               pointer to the input seed

ulSeedLen            length in bytes of the input seed

pLabel               pointer to the identifying label
ulLabelLen  length in bytes of the identifying label
pOutput  pointer receiving the output of the operation
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

CK_WTLS_PRF_PARAMS_PTR  is a pointer to a CK_WTLS_PRF_PARAMS.

♦  CK_WTLS_KEY_MAT_OUT; CK_WTLS_KEY_MAT_OUT_PTR

CK_WTLS_KEY_MAT_OUT  is a structure that contains the resulting key handles and initialization vectors after performing a C_DeriveKey function with the
CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE or with the
CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE mechanism. It is defined as follows:

```c
typedef struct CK_WTLS_KEY_MAT_OUT {
    CK_OBJECT_HANDLE hMacSecret;
    CK_OBJECT_HANDLE hKey;
    CK_BYTE_PTR      pIV;
} CK_WTLS_KEY_MAT_OUT;
```

The fields of the structure have the following meanings:

- **hMacSecret**  Key handle for the resulting MAC secret key
- **hKey**  Key handle for the resulting secret key
- **pIV**  Pointer to a location which receives the initialization vector (IV) created (if any)

CK_WTLS_KEY_MAT_OUT_PTR  is a pointer to a CK_WTLS_KEY_MAT_OUT.

♦  CK_WTLS_KEY_MAT_PARAMS; CK_WTLS_KEY_MAT_PARAMS_PTR

CK_WTLS_KEY_MAT_PARAMS  is a structure that provides the parameters to the
CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE and the
CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE mechanisms. It is defined as follows:

```c
typedef struct CK_WTLS_KEY_MAT_PARAMS {
    CK_MECHANISM_TYPE       DigestMechanism;
    CK_ULONG                ulMacSizeInBits;
    CK_ULONG                ulKeySizeInBits;
    CK_ULONG                ulIVSizeInBits;
    CK_ULONG                ulSequenceNumber;
    CK_BBOOL                bIsExport;
    CK_WTLS_RANDOM_DATA     RandomInfo;
    CK_WTLS_KEY_MAT_OUT_PTR pReturnedKeyMaterial;
} CK_WTLS_KEY_MAT_PARAMS;
```

The fields of the structure have the following meanings:
Digest Mechanism: the mechanism type of the digest mechanism to be used (possible types can be found in WTLS).

ulMaxSizeInBits: the length (in bits) of the MACing key agreed upon during the protocol handshake phase.

ulKeySizeInBits: the length (in bits) of the secret key agreed upon during the handshake phase.

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.

ulSequenceNumber: the current sequence number used for records sent by the client and server respectively.

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).

RandomInfo: client’s and server’s random data information.

pReturnedKeyMaterial: points to a CK_WTLS_KEY_MAT_OUT structure which receives the handles for the keys generated and the IV.

2.41.3 Pre master secret key generation for RSA key exchange suite

Pre master secret key generation for the RSA key exchange suite in WTLS denoted CKM_WTLS.Pre_MASTER_KEY_GEN, is a mechanism, which generates a variable length secret key. It is used to produce the pre master secret key for RSA key exchange suite used in WTLS. This mechanism returns a handle to the pre master secret key.

It has one parameter, a CK_BYTE, which provides the client’s WTLS version.

The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE and CKA_VALUE attributes to the new key (as well as the CKA_VALUE_LEN attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.

The template sent along with this mechanism during a C.GenerateKey call may indicate that the object class is CKO_SECRET_KEY, the key type is CKK_GENERIC_SECRET, and the CKA_VALUE_LEN attribute indicates the length of the pre master secret key.

For this mechanism, the ulMinKeySize field of the CK_MECHANISM_INFO structure shall indicate 20 bytes.

2.41.4 Master secret key derivation

Master secret derivation in WTLS, denoted CKM_WTLS_MASTER_KEY_DERIVE, is a mechanism used to derive a 20 byte generic secret key from variable length secret key. It is used to produce the master secret key used in WTLS from the pre master secret key. This mechanism returns the value of the client version, which is built into the pre master secret key as well as a handle to the derived master secret key.

It has a parameter, a CK_WTLS_MASTER_KEY_DERIVE_PARAMS structure, which allows for passing the mechanism type of the digest mechanism to be used as well as the passing of random data to the token as well as the returning of the protocol version number which is part of the pre master secret key.
The mechanism contributes the \texttt{CKA\_CLASS}, \texttt{CKA\_KEY\_TYPE}, and \texttt{CKA\_VALUE} attributes to the new key (as well as the \texttt{CKA\_VALUE\_LEN} attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.

The template sent along with this mechanism during a \texttt{C\_DeriveKey} call may indicate that the object class is \texttt{CKO\_SECRET\_KEY}, the key type is \texttt{CKK\_GENERIC\_SECRET}, and the \texttt{CKA\_VALUE\_LEN} attribute has value 20. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

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

The \texttt{CKA\_SENSITIVE} and \texttt{CKA\_EXTRACTABLE} attributes in the template for the new key can both be specified to be either \texttt{CK\_TRUE} or \texttt{CK\_FALSE}. If omitted, these attributes each take on some default value.

If the base key has its \texttt{CKA\_ALWAYS\_SENSITIVE} attribute set to \texttt{CK\_FALSE}, then the derived key will as well. If the base key has its \texttt{CKA\_ALWAYS\_SENSITIVE} attribute set to \texttt{CK\_TRUE}, then the derived key has its \texttt{CKA\_ALWAYS\_SENSITIVE} attribute set to the same value as its \texttt{CKA\_SENSITIVE} attribute.

Similarly, if the base key has its \texttt{CKA\_NEVER\_EXTRACTABLE} attribute set to \texttt{CK\_FALSE}, then the derived key will, too. If the base key has its \texttt{CKA\_NEVER\_EXTRACTABLE} attribute set to \texttt{CK\_TRUE}, then the derived key has its \texttt{CKA\_NEVER\_EXTRACTABLE} attribute set to the opposite value from its \texttt{CKA\_EXTRACTABLE} attribute.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the \texttt{CK\_MECHANISM\_INFO} structure both indicate 20 bytes.

Note that the \texttt{CK\_BYTE} pointed to by the \texttt{CK\_WTLS\_MASTER\_KEY\_DERIVE\_PARAMS} structure's \texttt{pVersion} field will be modified by the \texttt{C\_DeriveKey} call. In particular, when the call returns, this byte will hold the WTLS version associated with the supplied pre master secret key.

Note that this mechanism is only useable for key exchange suites that use a 20-byte pre master secret key with an embedded version number. This includes the RSA key exchange suites, but excludes the Diffie-Hellman and Elliptic Curve Cryptography key exchange suites.

### 2.41.5 Master secret key derivation for Diffie-Hellman and Elliptic Curve Cryptography

Master secret derivation for Diffie-Hellman and Elliptic Curve Cryptography in WTLS, denoted \texttt{CKM\_WTLS\_MASTER\_KEY\_DERIVE\_DH\_ECC}, is a mechanism used to derive a 20 byte generic secret key from variable length secret key. It is used to produce the master secret key used in WTLS from the pre master secret key. This mechanism returns a handle to the derived master secret key.

It has a parameter, a \texttt{CK\_WTLS\_MASTER\_KEY\_DERIVE\_PARAMS} structure, which allows for the passing of the mechanism type of the digest mechanism to be used as well as random data to the token. The \texttt{pVersion} field of the structure must be set to NULL\_PTR since the version number is not embedded in the pre master secret key as it is for RSA-like key exchange suites.

The mechanism contributes the \texttt{CKA\_CLASS}, \texttt{CKA\_KEY\_TYPE}, and \texttt{CKA\_VALUE} attributes to the new key (as well as the \texttt{CKA\_VALUE\_LEN} attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.

The template sent along with this mechanism during a \texttt{C\_DeriveKey} call may indicate that the object class is \texttt{CKO\_SECRET\_KEY}, the key type is \texttt{CKK\_GENERIC\_SECRET}, and the \texttt{CKA\_VALUE\_LEN} attribute has value 20. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

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

The \texttt{CKA\_SENSITIVE} and \texttt{CKA\_EXTRACTABLE} attributes in the template for the new key can both be specified to be either \texttt{CK\_TRUE} or \texttt{CK\_FALSE}. If omitted, these attributes each take on some default value.

If the base key has its \texttt{CKA\_ALWAYS\_SENSITIVE} attribute set to \texttt{CK\_FALSE}, then the derived key will as well. If the base key has its \texttt{CKA\_ALWAYS\_SENSITIVE} attribute set to \texttt{CK\_TRUE}, then the derived key has its \texttt{CKA\_ALWAYS\_SENSITIVE} attribute set to the same value as its \texttt{CKA\_SENSITIVE} attribute.
Similarly, if the base key has its `CKA_NEVER_EXTRACTABLE` attribute set to `CK_FALSE`, then the derived key will, too. If the base key has its `CKA_NEVER_EXTRACTABLE` attribute set to `CK_TRUE`, then the derived key has its `CKA_NEVER_EXTRACTABLE` attribute set to the opposite value from its `CKA_EXTRACTABLE` attribute.

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure both indicate 20 bytes.

Note that this mechanism is only useable for key exchange suites that do not use a fixed length 20-byte pre master secret key with an embedded version number. This includes the Diffie-Hellman and Elliptic Curve Cryptography key exchange suites, but excludes the RSA key exchange suites.

### 2.41.6 WTLS PRF (pseudorandom function)

PRF (pseudo random function) in WTLS, denoted `CKM_WTLS_PRF`, is a mechanism used to produce a securely generated pseudo-random output of arbitrary length. The keys it uses are generic secret keys.

It has a parameter, a `CK_WTLS_PRF_PARAMS` structure, which allows for passing the mechanism type of the digest mechanism to be used, the passing of the input seed and its length, the passing of an identifying label and its length and the passing of the length of the output to the token and for receiving the output.

This mechanism produces securely generated pseudo-random output of the length specified in the parameter.

This mechanism departs from the other key derivation mechanisms in Cryptoki in not using the template sent along with this mechanism during a `C_DeriveKey` function call, which means the template shall be a `NULL_PTR`. For most key-derivation mechanisms, `C_DeriveKey` returns a single key handle as a result of a successful completion. However, since the `CKM_WTLS_PRF` mechanism returns the requested number of output bytes in the `CK_WTLS_PRF_PARAMS` structure specified as the mechanism parameter, the parameter `phKey` passed to `C_DeriveKey` is unnecessary, and should be a `NULL_PTR`.

If a call to `C_DeriveKey` with this mechanism fails, then no output will be generated.

### 2.41.7 Server Key and MAC derivation

Server key, MAC and IV derivation in WTLS, denoted `CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE`, is a mechanism used to derive the appropriate cryptographic keying material used by a cipher suite from the master secret key and random data. This mechanism returns the key handles for the keys generated in the process, as well as the IV created.

It has a parameter, a `CK_WTLS_KEY_MAT_PARAMS` structure, which allows for the passing of the mechanism type of the digest mechanism to be used, random data, the characteristic of the cryptographic material for the given cipher suite, and a pointer to a structure which receives the handles and IV which were generated.

This mechanism contributes to the creation of two distinct keys and returns one IV (if an IV is requested by the caller) back to the caller. The keys are all given an object class of `CKO_SECRET_KEY`.

The MACing key (server write MAC secret) is always given a type of `CKK_GENERIC_SECRET`. It is flagged as valid for signing, verification and derivation operations.

The other key (server write key) is typed according to information found in the template sent along with this mechanism during a `C_DeriveKey` function call. By default, it is flagged as valid for encryption, decryption, and derivation operations.

An IV (server write IV) will be generated and returned if the `ullIVSizeInBits` field of the `CK_WTLS_KEY_MAT_PARAMS` field has a nonzero value. If it is generated, its length in bits will agree with the value in the `ullIVSizeInBits` field.

Both keys inherit the values of the `CKA_SENSITIVE`, `CKA_ALWAYS_SENSITIVE`, `CKA_EXTRACTABLE`, and `CKA_NEVER_EXTRACTABLE` attributes from the base key. The template provided to `C_DeriveKey` may not specify values for any of these attributes that differ from those held by the base key.
Note that the **CK_WTLS_KEY_MAT_OUT** structure pointed to by the **CK_WTLS_KEY_MAT_PARAMS** structure's \( p\)\( \text{ReturnedKeyMaterial} \) field will be modified by the **C\_DeriveKey** call. In particular, the two key handle fields in the **CK_WTLS_KEY_MAT_OUT** structure will be modified to hold handles to the newly-created keys; in addition, the buffer pointed to by the **CK_WTLS_KEY_MAT_OUT** structure's \( p/IV \) field will have the IV returned in them (if an IV is requested by the caller). Therefore, this field must point to a buffer with sufficient space to hold any IV that will be returned.

This mechanism departs from the other key derivation mechanisms in Cryptoki in its returned information.

For most key-derivation mechanisms, **C\_DeriveKey** returns a single key handle as a result of a successful completion. However, since the **CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE** mechanism returns all of its key handles in the **CK_WTLS_KEY_MAT_OUT** structure pointed to by the **CK_WTLS_KEY_MAT_PARAMS** structure specified as the mechanism parameter, the parameter \( phKey \) passed to **C\_DeriveKey** is unnecessary, and should be a NULL\_PTR.

If a call to **C\_DeriveKey** with this mechanism fails, then none of the two keys will be created.

### 2.41.8 Client key and MAC derivation

Client key, MAC and IV derivation in WTLS, denoted **CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE**, is a mechanism used to derive the appropriate cryptographic keying material used by a cipher suite from the master secret key and random data. This mechanism returns the key handles for the keys generated in the process, as well as the IV created.

It has a parameter, a **CK_WTLS_KEY_MAT_PARAMS** structure, which allows for the passing of the mechanism type of the digest mechanism to be used, random data, the characteristic of the cryptographic material for the given cipher suite, and a pointer to a structure which receives the handles and IV which were generated.

This mechanism contributes to the creation of two distinct keys and returns one IV (if an IV is requested by the caller) back to the caller. The keys are all given an object class of **CKO_SECRET_KEY**.

The MACing key (client write MAC secret) is always given a type of **CKK_GENERIC_SECRET**. It is flagged as valid for signing, verification and derivation operations.

The other key (client write key) is typed according to information found in the template sent along with this mechanism during a **C\_DeriveKey** function call. By default, it is flagged as valid for encryption, decryption, and derivation operations.

An IV (client write IV) will be generated and returned if the \( ulIVSizeInBits \) field of the **CK_WTLS_KEY_MAT_PARAMS** field has a nonzero value. If it is generated, its length in bits will agree with the value in the \( ulIVSizeInBits \) field

Both keys inherit the values of the **CKA_SENSITIVE**, **CKA_ALWAYS_SENSITIVE**, **CKA_EXTRACTABLE**, and **CKA_NEVER_EXTRACTABLE** attributes from the base key. The template provided to **C\_DeriveKey** may not specify values for any of these attributes that differ from those held by the base key.

Note that the **CK_WTLS_KEY_MAT_OUT** structure pointed to by the **CK_WTLS_KEY_MAT_PARAMS** structure's \( p\)\( \text{ReturnedKeyMaterial} \) field will be modified by the **C\_DeriveKey** call. In particular, the two key handle fields in the **CK_WTLS_KEY_MAT_OUT** structure will be modified to hold handles to the newly-created keys; in addition, the buffer pointed to by the **CK_WTLS_KEY_MAT_OUT** structure's \( p/IV \) field will have the IV returned in them (if an IV is requested by the caller). Therefore, this field must point to a buffer with sufficient space to hold any IV that will be returned.

This mechanism departs from the other key derivation mechanisms in Cryptoki in its returned information.

For most key-derivation mechanisms, **C\_DeriveKey** returns a single key handle as a result of a successful completion. However, since the **CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE** mechanism returns all of its key handles in the **CK_WTLS_KEY_MAT_OUT** structure pointed to by the **CK_WTLS_KEY_MAT_PARAMS** structure specified as the mechanism parameter, the parameter \( phKey \) passed to **C\_DeriveKey** is unnecessary, and should be a NULL\_PTR.

If a call to **C\_DeriveKey** with this mechanism fails, then none of the two keys will be created.
2.42 SP 800-108 Key Derivation

NIST SP800-108 defines three types of key derivation functions (KDF); a Counter Mode KDF, a Feedback Mode KDF and a Double Pipeline Mode KDF.

This section defines a unique mechanism for each type of KDF. These mechanisms can be used to derive one or more symmetric keys from a single base symmetric key.

The KDFs defined in SP800-108 are all built upon pseudo random functions (PRF). In general terms, the PRFs accepts two pieces of input; a base key and some input data. The base key is taken from the \( hBaseKey \) parameter to \texttt{C\_Derive}. The input data is constructed from an iteration variable (internally defined by the KDF/PRF) and the data provided in the \texttt{CK\_SP800\_108\_PRF\_DATA\_PARAM} array that is part of the mechanism parameter.

\textit{Table 161, SP800-108 Mechanisms vs. Functions}

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_SP800_108_COUNTER_KDF</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SP800_108_FEEDBACK_KDF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_SP800_108_DOUBLE_PIPELINE_KDF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

For these mechanisms, the \texttt{ulMinKeySize} and \texttt{ulMaxKeySize} fields of the \texttt{CK\_MECHANISM\_INFO} structure specify the minimum and maximum supported base key size in bits. Note, these mechanisms support multiple PRF types and key types; as such the values reported by \texttt{ulMinKeySize} and \texttt{ulMaxKeySize} specify the minimum and maximum supported base key size when all PRF and keys types are considered. For example, a Cryptoki implementation may support \texttt{CKK\_GENERIC\_SECRET} keys that can be as small as 8-bits in length and therefore \texttt{ulMinKeySize} could report 8-bits. However for an AES-CMAC PRF the base key must be of type \texttt{CKK\_AES} and must be either 16-bytes, 24-bytes or 32-bytes in lengths and therefore the value reported by \texttt{ulMinKeySize} could be misleading. Depending on the PRF type selected, additional key size restrictions may apply.

2.42.1 Definitions

Mechanisms:

- CKM\_SP800\_108\_COUNTER\_KDF
- CKM\_SP800\_108\_FEEDBACK\_KDF
- CKM\_SP800\_108\_DOUBLE\_PIPELINE\_KDF

Data Field Types:

- CK\_SP800\_108\_ITERATION\_variable
- CK\_SP800\_108\_COUNTER
- CK\_SP800\_108\_DKM\_LENGTH
- CK\_SP800\_108\_BYTE\_ARRAY

DKM Length Methods:

- CK\_SP800\_108\_DKM\_LENGTH\_SUM\_OF\_KEYS
- CK\_SP800\_108\_DKM\_LENGTH\_SUM\_OF\_SEGMENTS
2.42.2 Mechanism Parameters

♦ CK_SP800_108_PRF_TYPE

The CK_SP800_108_PRF_TYPE field of the mechanism parameter is used to specify the type of PRF that is to be used. It is defined as follows:

```
typedef CK_MECHANISM_TYPE CK_SP800_108_PRF_TYPE;
```

The CK_SP800_108_PRF_TYPE field reuses the existing mechanisms definitions. The following table lists the supported PRF types:

Table 162, SP800-108 Pseudo Random Functions

<table>
<thead>
<tr>
<th>Pseudo Random Function Identifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_SHA_1_HMAC</td>
</tr>
<tr>
<td>CKM_SHA224_HMAC</td>
</tr>
<tr>
<td>CKM_SHA256_HMAC</td>
</tr>
<tr>
<td>CKM_SHA384_HMAC</td>
</tr>
<tr>
<td>CKM_SHA512_HMAC</td>
</tr>
<tr>
<td>CKM_SHA3_224_HMAC</td>
</tr>
<tr>
<td>CKM_SHA3_256_HMAC</td>
</tr>
<tr>
<td>CKM_SHA3_384_HMAC</td>
</tr>
<tr>
<td>CKM_SHA3_512_HMAC</td>
</tr>
<tr>
<td>CKM_3DES_CMAC</td>
</tr>
<tr>
<td>CKM_AES_CMAC</td>
</tr>
</tbody>
</table>

♦ CK_PRF_DATA_TYPE

Each mechanism parameter contains an array of CK_PRF_DATA_PARAM structures. The CK_PRF_DATA_PARAM structure contains CK_PRF_DATA_TYPE field. The CK_PRF_DATA_TYPE field is used to identify the type of data identified by each CK_PRF_DATA_PARAM element in the array. Depending on the type of KDF used, some data field types are mandatory, some data field types are optional and some data field types are not allowed. These requirements are defined on a per-mechanism basis in the sections below. The CK_PRF_DATA_TYPE is defined as follows:

```
typedef CK_ULONG CK_PRF_DATA_TYPE;
```

The following table lists all of the supported data field types:
Table 163, SP800-108 PRF Data Field Types

<table>
<thead>
<tr>
<th>Data Field Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CK_SP800_108_ITERATION_VARIABLE</td>
<td>Identifies the iteration variable defined internally by the KDF.</td>
</tr>
<tr>
<td>CK_SP800_108_COUNTER</td>
<td>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.</td>
</tr>
<tr>
<td>CK_SP800_108_DKM_LENGTH</td>
<td>Identifies the length in bits of the derived keying material (DKM) represented as a binary string. Exact formatting of the length value is defined by the CK_SP800_108_DKM_FORMAT structure.</td>
</tr>
<tr>
<td>CK_SP800_108_BYTE_ARRAY</td>
<td>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.</td>
</tr>
</tbody>
</table>

**CK_PRF_DATA_PARAM**

CK_PRF_DATA_PARAM is used to define a segment of input for the PRF. Each mechanism parameter supports an array of CK_PRF_DATA_PARAM structures. The CK_PRF_DATA_PARAM is defined as follows:

```c
typedef struct CK_PRF_DATA_PARAM {
    CK_PRF_DATA_TYPE    type;
    CK_VOID_PTR         pValue;
    CK_ULONG            ulValueLen;
} CK_PRF_DATA_PARAM;
```

The fields of the CK_PRF_DATA_PARAM structure have the following meaning:

- **type** defines the type of data pointed to by pValue
- **pValue** pointer to the data defined by type
- **ulValueLen** size of the data pointed to by pValue

If the `type` field of the CK_PRF_DATA_PARAM structure is set to CK_SP800_108_ITERATION_VARIABLE, then `pValue` must be set the appropriate value for the KDF’s iteration variable type. For the Counter Mode KDF, `pValue` must be assigned a valid CK_SP800_108_COUNTER_FORMAT_PTR and `ulValueLen` must be set to sizeof(CK_SP800_108_COUNTER_FORMAT). For all other KDF types, `pValue` must be set to NULL_PTR and `ulValueLen` must be set to 0.

If the `type` field of the CK_PRF_DATA_PARAM structure is set to CK_SP800_108_COUNTER, then `pValue` must be assigned a valid CK_SP800_108_COUNTER_FORMAT_PTR and `ulValueLen` must be set to sizeof(CK_SP800_108_COUNTER_FORMAT).
If the `type` field of the `CK_PRF_DATA_PARAM` structure is set to `CK_SP800_108_DKM_LENGTH` then
`pValue` must be assigned a valid `CK_SP800_108_DKM_FORMAT_PTR` and `ulValueLen` must be set to
`sizeof(CK_SP800_108_DKM_FORMAT)`.

If the `type` field of the `CK_PRF_DATA_PARAM` structure is set to `CK_SP800_108_BYTE_ARRAY`, then
`pValue` must be assigned a valid `CK_BYTE_PTR` value and `ulValueLen` must be set to a non-zero length.

**CK_SP800_108_COUNTER_FORMAT**

`CK_SP800_108_COUNTER_FORMAT` is used to define the encoding format for a counter value. The
`CK_SP800_108_COUNTER_FORMAT` is defined as follows:

```c
typedef struct CK_SP800_108_COUNTER_FORMAT {
    CK_BBOOL     bLittleEndian;
    CK_ULONG     ulWidthInBits;
} CK_SP800_108_COUNTER_FORMAT;
```

The fields of the `CK_SP800_108_COUNTER_FORMAT` structure have the following meaning:

- `bLittleEndian` defines if the counter should be represented in Big Endian or Little
  Endian format
- `ulWidthInBits` defines the number of bits used to represent the counter value

**CK_SP800_108_DKM_LENGTH_METHOD**

`CK_SP800_108_DKM_LENGTH_METHOD` is used to define how the DKM length value is calculated.

The `CK_SP800_108_DKM_LENGTH_METHOD` type is defined as follows:

```c
typedef CK_ULONG CK_SP800_108_DKM_LENGTH_METHOD;
```

The following table lists all of the supported DKM Length Methods:

<table>
<thead>
<tr>
<th>DKM Length Method Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS</code></td>
<td>Specifies that the DKM length should be set to the sum of the length of all keys derived by this invocation of the KDF.</td>
</tr>
<tr>
<td><code>CK_SP800_108_DKM_LENGTH_SUM_OF_SEGMENTS</code></td>
<td>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.</td>
</tr>
</tbody>
</table>

**CK_SP800_108_DKM_LENGTH_FORMAT**

`CK_SP800_108_DKM_LENGTH_FORMAT` is used to define the encoding format for the DKM length
value. The `CK_SP800_108_DKM_LENGTH_FORMAT` is defined as follows:

```c
typedef struct CK_SP800_108_DKM_LENGTH_FORMAT
```
The fields of the CK_SP800_108_DKM_LENGTH_FORMAT structure have the following meaning:

- `dkmLengthMethod` defines the method used to calculate the DKM length value
- `bLittleEndian` defines if the DKM length value should be represented in Big or Little Endian format
- `ulWidthInBits` defines the number of bits used to represent the DKM length value

### CK_DERIVED_KEY

`CK_DERIVED_KEY` is used to define an additional key to be derived as well as provide a `CK_OBJECT_HANDLE_PTR` to receive the handle for the derived keys. The `CK_DERIVED_KEY` is defined as follows:

```c
typedef struct CK_DERIVED_KEY
{
    CK_ATTRIBUTE_PTR     pTemplate;
    CK_ULONG             ulAttributeCount;
    CK_OBJECT_HANDLE_PTR phKey;
} CK_DERIVED_KEY;
```

The fields of the `CK_DERIVED_KEY` structure have the following meaning:

- `pTemplate` pointer to a template that defines a key to derive
- `ulAttributeCount` number of attributes in the template pointed to by `pTemplate`
- `phKey` pointer to receive the handle for a derived key

### CK_SP800_108_KDF_PARAMS, CK_SP800_108_KDF_PARAMS_PTR

`CK_SP800_108_KDF_PARAMS` is a structure that provides the parameters for the `CKM_SP800_108_COUNTER_KDF` and `CKM_SP800_108_DOUBLE_PIPELINE_KDF` mechanisms.

```c
typedef struct CK_SP800_108_KDF_PARAMS
{
    CK_PRF_TYPE            prfType;
    CK_ULONG               ulNumberOfDataParams;
    CK_PRF_DATA_PARAM_PTR  pDataParams;
    CK_ULONG               ulAdditionalDerivedKeys;
} CK_SP800_108_KDF_PARAMS
```
CK_DERIVED_KEY pAdditionalDerivedKeys;
} CK_SP800_108_KDF_PARAMS;

typedef CK_SP800_108_KDF_PARAMS CK_PTR
CK_SP800_108_KDF_PARAMS_PTR;

The fields of the CK_SP800_108_KDF_PARAMS structure have the following meaning:

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>prfType</td>
<td>type of PRF</td>
</tr>
</tbody>
</table>
| ulNumberOfDataParams | number of elements in the array pointed to by pDataParams |}
| pDataParams      | an array of CK_PRF_DATA_PARAM structures. The array defines input parameters that are used to construct the “data” input to the PRF. |
| 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. |
| pAdditionalDerivedKeys | an array of CK_DERIVED_KEY structures. If ulAdditionalDerivedKeys is set to 0, this parameter must be set to NULL_PTR |

**CK_SP800_108_FEEDBACK_KDF_PARAMS, CK_SP800_108_FEEDBACK_KDF_PARAMS_PTR**

The CK_SP800_108_FEEDBACK_KDF_PARAMS structure provides the parameters for the CKM_SP800_108_FEEDBACK_KDF mechanism. It is defined as follows:

typedef struct CK_SP800_108_FEEDBACK_KDF_PARAMS
{
    CK_PRF_TYPE prfType;
    CK_ULONG ulNumberOfDataParams;
    CK_PRF_DATA_PARAM_PTR pDataParams;
    CK_ULONG ulIVLen;
    CK_BYTE_PTR pIV;
    CK_ULONG ulAdditionalDerivedKeys;
    CK_DERIVED_KEY pAdditionalDerivedKeys;
} CK_SP800_108_FEEDBACK_KDF_PARAMS;

typedef CK_SP800_108_FEEDBACK_KDF_PARAMS CK_PTR
CK_SP800_108_FEEDBACK_KDF_PARAMS_PTR;

The fields of the CK_SP800_108_FEEDBACK_KDF_PARAMS structure have the following meaning:

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>prfType</td>
<td>type of PRF</td>
</tr>
</tbody>
</table>
| ulNumberOfDataParams | number of elements in the array pointed to by pDataParams |}
| pDataParams      | an array of CK_PRF_DATA_PARAM structures. The array defines input parameters that are used to construct the “data” input to the PRF. |
ulIVLen

the length in bytes of the IV. If pIV is set to NULL_PTR, this parameter must be set to 0.

pIV

an array of bytes to be used as the IV for the feedback mode KDF. This parameter is optional and can be set to NULL_PTR. If ulIVLen is set to 0, this parameter must be set to NULL_PTR.

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.

pAdditionalDerivedKeys

an array of CK_DERIVED_KEYS structures. If ulAdditionalDerivedKeys is set to 0, this parameter must be set to NULL_PTR.

2.42.3 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>
<thead>
<tr>
<th>Data Field Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CK_SP800_108_ITERATION_VARIABLE</td>
<td>This data field type is mandatory. This data field type identifies the location of the iteration variable in the constructed PRF input data. The iteration variable for this KDF type is a counter. Exact formatting of the counter value is defined by the CK_SP800_108_COUNTER_FORMAT structure.</td>
</tr>
<tr>
<td>CK_SP800_108_COUNTER</td>
<td>This data field type is invalid for this KDF type.</td>
</tr>
<tr>
<td>CK_SP800_108_DKM_LENGTH</td>
<td>This data field type is optional. This data field type identifies the location of the DKM length in the constructed PRF input data. Exact formatting of the DKM length is defined by the CK_SP800_108_DKM_LENGTH_FORMAT structure. If specified, only one instance of this type may be specified.</td>
</tr>
<tr>
<td>CK_SP800_108_BYTE_ARRAY</td>
<td>This data field type is optional. This data field type identifies the location and value of a byte array of data in the constructed PRF input data. This standard does not restrict the number of instances of this data type.</td>
</tr>
</tbody>
</table>

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.
2.42.4 Feedback Mode KDF

The SP800-108 Feedback Mode KDF mechanism, denoted CKM_SP800_108_FEEDBACK_KDF, represents the KDF defined SP800-108 section 5.2. CKM_SP800_108_FEEDBACK_KDF is a mechanism for deriving one or more symmetric keys from a symmetric base key. It has a parameter, a CK_SP800_108_FEEDBACK_KDF_PARAMS structure.

The following table lists the data field types that are supported for this KDF type and their meaning:

Table 166, Feedback Mode data field requirements

<table>
<thead>
<tr>
<th>Data Field Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CK_SP800_108_ITERATION_VARIABLE</td>
<td>This data field type is mandatory. This data field type identifies the location of the iteration variable in the constructed PRF input data. The iteration variable is defined as K(i-1) in section 5.2 of SP800-108. The size, format and value of this data input is defined by the internal KDF structure and PRF output. Exact formatting of the counter value is defined by the CK_SP800_108_COUNTER_FORMAT structure.</td>
</tr>
<tr>
<td>CK_SP800_108_COUNTER</td>
<td>This data field type is optional. This data field type identifies the location of the counter in the constructed PRF input data. Exact formatting of the counter value is defined by the CK_SP800_108_COUNTER_FORMAT structure. If specified, only one instance of this type may be specified.</td>
</tr>
<tr>
<td>CK_SP800_108_DKM_LENGTH</td>
<td>This data field type is optional. This data field type identifies the location of the DKM length in the constructed PRF input data. Exact formatting of the DKM length is defined by the CK_SP800_108_DKM_LENGTH_FORMAT structure. If specified, only one instance of this type may be specified.</td>
</tr>
<tr>
<td>CK_SP800_108_BYTE_ARRAY</td>
<td>This data field type is optional. This data field type identifies the location and value of a byte array of data in the constructed PRF input data. This standard does not restrict the number of instances of this data type.</td>
</tr>
</tbody>
</table>

SP800-108 limits the amount of derived keying material that can be produced by a Feedback Mode KDF by limiting the internal loop counter to \(2^{32} - 1\). Therefore the maximum number of bits that can be produced is \(2^{32} - 1\)h, where “h” is the length in bits of the output of the selected PRF.

2.42.5 Double Pipeline Mode KDF

The SP800-108 Double Pipeline Mode KDF mechanism, denoted CKM_SP800_108_DOUBLE_PIPELINE_KDF, represents the KDF defined SP800-108 section 5.3. CKM_SP800_108DOUBLE_PIPELINE_KDF is a mechanism for deriving one or more symmetric keys from a symmetric base key.

It has a parameter, a CK_SP800_108_KDF_PARAMS structure.

The following table lists the data field types that are supported for this KDF type and their meaning:
### Table 167, Double Pipeline Mode data field requirements

<table>
<thead>
<tr>
<th>Data Field Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CK_SP800_108_ITERATION_VARIABLE</td>
<td>This data field type is mandatory. This data field type identifies the location of the iteration variable in the constructed PRF input data. The iteration variable is defined as A(i) in section 5.3 of SP800-108. The size, format and value of this data input is defined by the internal KDF structure and PRF output. Exact formatting of the counter value is defined by the CK_SP800_108_COUNTER_FORMAT structure.</td>
</tr>
<tr>
<td>CK_SP800_108_COUNTER</td>
<td>This data field type is optional. This data field type identifies the location of the counter in the constructed PRF input data. Exact formatting of the counter value is defined by the CK_SP800_108_COUNTER_FORMAT structure. If specified, only one instance of this type may be specified.</td>
</tr>
<tr>
<td>CK_SP800_108_DKM_LENGTH</td>
<td>This data field type is optional. This data field type identifies the location of the DKM length in the constructed PRF input data. Exact formatting of the DKM length is defined by the CK_SP800_108_DKM_LENGTH_FORMAT structure. If specified, only one instance of this type may be specified.</td>
</tr>
<tr>
<td>CK_SP800_108_BYTE_ARRAY</td>
<td>This data field type is optional. This data field type identifies the location and value of a byte array of data in the constructed PRF input data. This standard does not restrict the number of instances of this data type.</td>
</tr>
</tbody>
</table>

SP800-108 limits the amount of derived keying material that can be produced by a Double-Pipeline Mode KDF by limiting the internal loop counter to \((2^{32}−1)\). Therefore the maximum number of bits that can be produced is \((2^{32}−1)h\), where “h” is the length in bits of the output of the selected PRF.

The Double Pipeline KDF requires an internal IV value. The IV is constructed using the same method used to construct the PRF input data; the data/values identified by the array of CK_PRF_DATA_PARAM structures are concatenated in to a byte array that is used as the IV. As shown in SP800-108 section 5.3, the CK_SP800_108_ITERATION_VARIABLE and CK_SP800_108_COUNTER data field types are not included in IV construction process. All other data field types are included in the construction process.

#### 2.42.6 Deriving Additional Keys

The KDFs defined in this section can be used to derive more than one symmetric key from the base key. The C_Derive function accepts one CK_ATTRIBUTE_PTR to define a single derived key and one CK_OBJECT_HANDLE_PTR to receive the handle for the derived key.

To derive additional keys, the mechanism parameter structure can be filled in with one or more CK_DERIVED_KEY structures. Each structure contains a CK_ATTRIBUTE_PTR to define a derived key and a CK_OBJECT_HANDLE_PTR to receive the handle for the additional derived keys. The key defined by the C_Derive function parameters is always derived before the keys defined by the CKDERIVED_KEY array that is part of the mechanism parameter. The additional keys that are defined by the CK_DERIVED_KEY array are derived in the order they are defined in the array. That is to say that the derived keying material produced by the KDF is processed from left to right, and bytes are assigned...
first to the key defined by the \texttt{C\_Derive} function parameters, and then bytes are assigned to the keys that
are defined by the \texttt{CK\_DERIVED\_KEY} array in the order they are defined in the array.

Each internal iteration of a KDF produces a unique segment of PRF output. Sometimes, a single iteration
will produce enough keying material for the key being derived. Other times, additional internal iterations
are performed to produce multiple segments which are concatenated together to produce enough keying
material for the derived key(s).

When deriving multiple keys, no key can be created using part of a segment that was used for another
key. All keys must be created from disjoint segments. For example, if the parameters are defined such
that a 48-byte key (defined by the \texttt{C\_Derive} function parameters) and a 16-byte key (defined by the
content of \texttt{CK\_DERIVED\_KEY}) are to be derived using \texttt{CKM\_SHA256\_HMAC} as a PRF, three internal
iterations of the KDF will be performed and three segments of PRF output will be produced. The first
segment and half of the second segment will be used to create the 48-byte key and the third segment will
be used to create the 16-byte key:

\begin{center}
\begin{tabular}{c|c|c|c}
\hline
\textbf{1 KDF Segments of Output:} & \textbf{2 Derived Keys:} \\
\hline
\texttt{32-byte segment} & \texttt{48-byte key} & \texttt{unused} & \texttt{16-byte key} & \texttt{unused} \\
\hline
\end{tabular}
\end{center}

In the above example, if the \texttt{CK\_SP800\_108\_DKM\_LENGTH} data field type is specified with method
\texttt{CK\_SP800\_108\_DKM\_LENGTH\_SUM\_OF\_KEYS}, then the DKM length value will be 512 bits. If the
\texttt{CK\_SP800\_108\_DKM\_LENGTH} data field type is specified with method
\texttt{CK\_SP800\_108\_DKM\_LENGTH\_SUM\_OF\_SEGMENTS}, then the DKM length value will be 768 bits.

When deriving multiple keys, if any of the keys cannot be derived for any reason, none of the keys shall
be derived. If the failure was caused by the content of a specific key's template (ie the template defined
by the content of \texttt{pTemplate}), the corresponding \texttt{pHKey} value will be set to \texttt{CK\_HANDLE\_INVALID} to
identify the offending template.

\subsection{2.42.7 Key Derivation Attribute Rules}

The \texttt{CKM\_SP800\_108\_COUNTER\_KDF}, \texttt{CKM\_SP800\_108\_FEEDBACK\_KDF} and
\texttt{CKM\_SP800\_108\_DOUBLE\_PIPELINE\_KDF} mechanisms have the following rules about key sensitivity
and extractability:

\begin{itemize}
\item The \texttt{CKA\_SENSITIVE} and \texttt{CKA\_EXTRACTABLE} attributes in the template for the new key(s) can
both be specified to be either \texttt{CK\_TRUE} or \texttt{CK\_FALSE}. If omitted, these attributes each take on
some default value.
\item If the base key has its \texttt{CKA\_ALWAYS\_SENSITIVE} attribute set to \texttt{CK\_FALSE}, then the derived key
will as well. If the base key has its \texttt{CKA\_ALWAYS\_SENSITIVE} attribute set to \texttt{CK\_TRUE}, then the
derived key has its \texttt{CKA\_ALWAYS\_SENSITIVE} attribute set to the same value as its
\texttt{CKA\_SENSITIVE} attribute.
\item Similarly, if the base key has its \texttt{CKA\_NEVER\_EXTRACTABLE} attribute set to \texttt{CK\_FALSE}, then the derived key will, too. If the base key has its \texttt{CKA\_NEVER\_EXTRACTABLE} attribute set to
\texttt{CK\_TRUE}, then the derived key has its \texttt{CKA\_NEVER\_EXTRACTABLE} attribute set to the opposite
value from its \texttt{CKA\_EXTRACTABLE} attribute.
\end{itemize}

\subsection{2.42.8 Constructing PRF Input Data}

\texttt{SP800-108} defines the PRF input data for each KDF at a high level using terms like “label”, “context”,
“separator”, “counter”…etc. The value, formatting and order of the input data is not strictly defined by
\texttt{SP800-108}, instead it is described as being defined by the “encoding scheme”.

To support any encoding scheme, these mechanisms construct the PRF input data from from the array of
\texttt{CK\_PRF\_DATA\_PARAM} structures in the mechanism parameter. All of the values defined by the
\texttt{CK\_PRF\_DATA\_PARAM} array are concatenated in the order they are defined and passed in to the PRF
as the data parameter.
### 2.42.8.1 Sample Counter Mode KDF

SP800-108 section 5.1 outlines a sample Counter Mode KDF which defines the following PRF input:

\[
\text{PRF} \left( K_i, [i]_2 \| \text{Label} \| 0x00 \| \text{Context} \| [L]_2 \right)
\]

Section 5.1 does not define the number of bits used to represent the counter (the “r” value) or the DKM length (the “L” value), so 16-bits is assumed for both cases. The following sample code shows how to define this PRF input data using an array of CK_PRF_DATA_PARAM structures.

```c
#define DIM((a)) (sizeof((a))/sizeof((a)[0]))

CK_OBJECT_HANDLE hBaseKey;
CK_OBJECT_HANDLE hDerivedKey;
CK_ATTRIBUTE derivedKeyTemplate = { ... };

CK_BYTE baLabel[] = {0xde, 0xad, 0xbe, 0xef};
CK_ULONG ulLabelLen = sizeof(baLabel);
CK_BYTE baContext[] = {0xe0, 0xed, 0xbe, 0xef};
CK_ULONG ulContextLen = sizeof(baContext);

CK_SP800_108_COUNTER_FORMAT counterFormat = { 0, 16};
CK_SP800_108_DKM_FORMAT dkmFormat = {CK_SP800_108_SUM_OF_KEYS, 0, 16};
CK_PRF_DATA_PARAM dataParams[] =
    { CK_SP800_108_ITERATION_VARIABLE, &counterFormat, sizeof(counterFormat) },
    { CK_SP800_108_BYTE_ARRAY, baLabel, ulLabelLen },
    { CK_SP800_108_BYTE_ARRAY, {0x00}, 1 },
    { CK_SP800_108_BYTE_ARRAY, baContext, ulContextLen },
    { CK_SP800_108_DKM_LENGTH, dkmFormat, sizeof(dkmFormat) };

CK_SP800_108_KDF_PARAMS kdfParams =
    { CK_PRF_AES_CMAC, DIM(dataParams), &dataParams, 0, /* no addition derived keys */ NULL /* no addition derived keys */ };;

CK_MECHANISM = mechanism
    { CKM_FLEXIBLE_KDF, &kdfParams, sizeof(kdfParams) };

hBaseKey = GetBaseKeyHandle(.....);
rv = C_DeriveKey(
    hSession,
    &mechanism,
    hBaseKey,
    &derivedKeyTemplate,
    DIM(derivedKeyTemplate),
...)
```
2.42.8.2 Sample SCP03 Counter Mode KDF

The SCP03 standard defines a variation of a counter mode KDF which defines the following PRF input:

\[
\text{PRF} (K_t, \text{Label} || 0x00 || [L]_2 || [r]_2 || \text{Context})
\]

SCP03 defines the number of bits used to represent the counter (the “r” value) and number of bits used to represent the DKM length (the “L” value) as 16-bits. The following sample code shows how to define this PRF input data using an array of CK_PRF_DATA_PARAM structures.

```c
#define DIM(a) (sizeof((a))/sizeof((a)[0]))

CK_OBJECT_HANDLE hBaseKey;
CK_OBJECT_HANDLE hDerivedKey;
CK_ATTRIBUTE derivedKeyTemplate = { ... };

CK_BYTE baLabel[] = {0xde, 0xad, 0xbe, 0xef};
CK_ULONG ulLabelLen = sizeof(baLabel);
CK_BYTE baContext[] = {0xfe, 0xed, 0xbe, 0xef};
CK_ULONG ulContextLen = sizeof(baContext);
CK_SP800_108_COUNTER_FORMAT counterFormat = {0, 16};
CK_SP800_108_DKM_FORMAT dkmFormat = {CK_SP800_108_SUM_OF_KEYS, 0, 16};

CK_PRF_DATA_PARAM dataParams[] =
{
    { CK_SP800_108_BYTE_ARRAY, baLabel, ulLabelLen },
    { CK_SP800_108_BYTE_ARRAY, {0x00}, 1 },
    { CK_SP800_108_DKM_LENGTH, dkmFormat, sizeof(dkmFormat) },
    { CK_SP800_108_ITERATION_VARIABLE,
        &counterFormat, sizeof(counterFormat) },
    { CK_SP800_108_BYTE_ARRAY, baContext, ulContextLen }
};

CK_SP800_108_KDF_PARAMS kdfParams =
{
    CK_PRF_AES_CMAC,
    DIM(dataParams),
    &dataParams,
    0, /* no addition derived keys */
    NULL /* no addition derived keys */
};

CK_MECHANISM = mechanism
{
    CKM_FLEXIBLE_KDF,
    &kdfParams,
    sizeof(kdfParams)
};

hBaseKey = GetBaseKeyHandle(.....);

rv = C_DeriveKey(
    hSession,
    &mechanism,
    &hDerivedKey);
```
2.42.8.3 Sample Feedback Mode KDF

SP800-108 section 5.2 outlines a sample Feedback Mode KDF which defines the following PRF input:

\[
\text{PRF} \left( K_t, K(i-1) \ \big| \ |\ i \ | \ | Label \ | \ | 0x00 \ | \ | Context \ | \ | [L] \right)
\]

Section 5.2 does not define the number of bits used to represent the counter (the “r” value) or the DKM length (the “L” value), so 16-bits is assumed for both cases. The counter is defined as being optional and is included in this example. The following sample code shows how to define this PRF input data using an array of CK_PRF_DATA_PARAM structures.

```c
#define DIM(a) (sizeof((a))/sizeof((a)[0]))

CK_OBJECT_HANDLE hBaseKey;
CK_OBJECT_HANDLE hDerivedKey;
CK_ATTRIBUTE derivedKeyTemplate = { ... };  
CK_BYTE baFeedbackIV[] = {0x01, 0x02, 0x03, 0x04};
CK_ULONG ulFeedbackIVLen = sizeof(baFeedbackIV);
CK_BYTE baLabel[] = {0xde, 0xad, 0xbe, 0xef};
CK_ULONG ulLabelLen = sizeof(baLabel);
CK_BYTE baContext[] = {0x00, 0x00, 0x00, 0x00};
CK_ULONG ulContextLen = sizeof(baContext);

CK_SP800_108_COUNTER_FORMAT counterFormat = { 0, 16};
CK_SP800_108_DKM_FORMAT dkmFormat = {CK_SP800_108_SUM_OF_KEYS, 0, 16};

CK_PRF_DATA_PARAM dataParams[] = {
    { CK_SP800_108_ITERATION_VARIABLE,
        &counterFormat, sizeof(counterFormat) },
    { CK_SP800_108_BYTE_ARRAY, baLabel, ulLabelLen },
    { CK_SP800_108_BYTE_ARRAY, {0x00}, 1 },
    { CK_SP800_108_BYTE_ARRAY, baContext, ulContextLen },
    { CK_SP800_108_DKM_LENGTH, dkmFormat, sizeof(dkmFormat) }
};

CK_SP800_108_FEEDBACK_KDF_PARAMS kdfParams = {
    CK_PRF_AES_CMAC,
    DIM(dataParams),
    &dataParams,
    ulFeedbackIVLen,
    pFeedbackIV,
    0, /* no addition derived keys */
    NULL /* no addition derived keys */
};

CK_MECHANISM = mechanism
{  
    CKM_FLEXIBLE_KDF,
    &kdfParams,
};
```
2.42.8.4 Sample Double-Pipeline Mode KDF

SP800-108 section 5.3 outlines a sample Double-Pipeline Mode KDF which defines the two following PRF inputs:

\[ \text{PRF} \left(K_I, A(i-1)\right) \]
\[ \text{PRF} \left(K_I, K(i-1) \mid || \mid Label \mid 0x00 \mid Context \mid \mid L\mid\right) \]

Section 5.3 does not define the number of bits used to represent the counter (the “r” value) or the DKM length (the “L” value), so 16-bits is assumed for both cases. The counter is defined as being optional so it is left out in this example. The following sample code shows how to define this PRF input data using an array of \text{CK\_PRF\_DATA\_PARAM} structures.

```c
#define DIM(a) (sizeof((a))/sizeof((a)[0]))

CK_OBJECT_HANDLE hBaseKey;
CK_OBJECT_HANDLE hDerivedKey;
CK_ATTRIBUTE derivedKeyTemplate = { ... };
CK_BYTE baLabel[] = {0xde, 0xad, 0xbe, 0xef};
CK_ULONG ulLabelLen = sizeof(baLabel);
CK_BYTE baContext[] = {0xfe, 0xed, 0xbe, 0xef};
CK_ULONG ulContextLen = sizeof(baContext);
CK_SP800_108_DKM_FORMAT dkmFormat = {CK_SP800_108_SUM_OF_KEYS, 0, 16};
CK_PRF_DATA_PARAM dataParams[] =
{
    { CK_SP800_108_BYTE_ARRAY, baLabel, ulLabelLen },
    { CK_SP800_108_BYTE_ARRAY, {0x00}, 1 },
    { CK_SP800_108_BYTE_ARRAY, baContext, ulContextLen },
    { CK_SP800_108_DKM_LENGTH, dkmFormat, sizeof(dkmFormat) }
};
CK_SP800_108_KDF_PARAMS kdfParams =
{
    CK_PRF_AES_CMAC, 
    DIM(dataParams),
    &dataParams,
    0, /* no addition derived keys */
    NULL /* no addition derived keys */
};
```

```
29 May 2019
Standards Track Work Product
Copyright © OASIS Open 2019. All Rights Reserved.
```

Page 188 of 281
CKM_FLEXIBLE_KDF,
  &kdfParams,
  sizeof(kdfParams)
);  

hBaseKey = GetBaseKeyHandle(....);

rv = C_DeriveKey(
  hSession,
  &mechanism,
  hBaseKey,
  &derivedKeyTemplate,
  DIM(derivedKeyTemplate),
  &hDerivedKey);

2.43 Miscellaneous simple key derivation mechanisms

Table 168, Miscellaneous simple key derivation Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_CONCATENATE_BASE_AND_KEY</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_CONCATENATE_BASE_AND_DATA</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_CONCATENATE_DATA_AND_BASE</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_XOR_BASE_AND_DATA</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_EXTRACT_KEY_FROM_KEY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

2.43.1 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

2.43.2 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;
}
The fields of the structure have the following meanings:

\[
\text{pData} \quad \text{pointer to the byte string}
\]

\[
\text{ulLen} \quad \text{length of the byte string}
\]

\[
\text{CK\_KEY\_DERIVATION\_STRING\_DATA\_PTR} \text{ is a pointer to a CK\_KEY\_DERIVATION\_STRING\_DATA.}
\]

\[
\text{CK\_EXTRACT\_PARAMS; CK\_EXTRACT\_PARAMS\_PTR}
\]

\[
\text{CK\_EXTRACT\_PARAMS} \text{ provides the parameter to the CKM\_EXTRACT\_KEY\_FROM\_KEY mechanism. It specifies which bit of the base key should be used as the first bit of the derived key. It is defined as follows:}
\]

\[
\text{typedef CK\_ULONG CK\_EXTRACT\_PARAMS;}
\]

\[
\text{CK\_EXTRACT\_PARAMS\_PTR} \text{ is a pointer to a CK\_EXTRACT\_PARAMS.}
\]

\[
\text{2.43.3 Concatenation of a base key and another key}
\]

This mechanism, denoted CKM\_CONCATENATE\_BASE\_AND\_KEY, derives a secret key from the concatenation of two existing secret keys. The two keys are specified by handles; the values of the keys specified are concatenated together in a buffer.

This mechanism takes a parameter, a CK\_OBJECT\_HANDLE. This handle produces the key value information which is appended to the end of the base key's value information (the base key is the key whose handle is supplied as an argument to C\_DeriveKey).

For example, if the value of the base key is 0x01234567, and the value of the other key is 0x89ABCDEF, then the value of the derived key will be taken from a buffer containing the string 0x0123456789ABCDEF.

- 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 sum of the lengths of the values of the two original keys.
- If no key type is provided in the template, but a length is, then the key produced by this mechanism will be a generic secret key of the specified length.
- If no length is provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn't, an error will be returned.
- If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism will be of the specified type and length.

If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more bytes than are available by concatenating the two original keys' values, an error is generated.

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

- If either of the two original keys 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 either of the two original keys 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 both of the original keys have their **CKA_ALWAYS_SENSITIVE** attributes set to CK_TRUE.

• Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to CK_TRUE if and only if both of the original keys have their **CKA_NEVER_EXTRACTABLE** attributes set to CK_TRUE.

### 2.43.4 Concatenation of a base key and data

This mechanism, denoted **CKM_CONCATENATE_BASE_AND_DATA**, derives a secret key by concatenating data onto the end of a specified secret key.

This mechanism takes a parameter, a **CK_KEY_DERIVATION_STRING_DATA** structure, which specifies the length and value of the data which will be appended to the base key to derive another key.

For example, if the value of the base key is 0x01234567, and the value of the data is 0x89ABCDEF, then the value of the derived key will be taken from a buffer containing the string 0x0123456789ABCDEF.

• 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 sum of the lengths of the value of the original key and the data.

• If no key type is provided in the template, but a length is, then the key produced by this mechanism will be a generic secret key of the specified length.

• If no length is provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn't, an error will be returned.

• If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism will be of the specified type and length.

If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more bytes than are available by concatenating the original key's value and the data, an error is generated.

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

• If the base key has its **CKA_SENSITIVE** attribute set to CK_TRUE, so does the derived key. If not, then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied template or from a default value.

• Similarly, if the base key has its **CKA_EXTRACTABLE** attribute set to CK_FALSE, so does the derived key. If not, then the derived key's **CKA_EXTRACTABLE** attribute is set either from the supplied template or from a default value.

• The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to CK_TRUE if and only if the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE.

• Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to CK_TRUE if and only if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_TRUE.

### 2.43.5 Concatenation of data and a base key

This mechanism, denoted **CKM_CONCATENATE_DATA_AND_BASE**, derives a secret key by prepending data to the start of a specified secret key.

This mechanism takes a parameter, a **CK_KEY_DERIVATION_STRING_DATA** structure, which specifies the length and value of the data which will be prepended to the base key to derive another key.

For example, if the value of the base key is 0x01234567, and the value of the data is 0x89ABCDEF, then the value of the derived key will be taken from a buffer containing the string 0x89ABCDEF01234567.

• 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 sum of the lengths of the data and the value of the original key.
If no key type is provided in the template, but a length is, then the key produced by this mechanism will be a generic secret key of the specified length.

If no length is provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn't, an error will be returned.

If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism will be of the specified type and length.

If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more bytes than are available by concatenating the data and the original key's value, an error is generated.

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

- If the base key has its `CKA_SENSITIVE` attribute set to CK_TRUE, so does the derived key. If not, then the derived key's `CKA_SENSITIVE` attribute is set either from the supplied template or from a default value.
- Similarly, if the base key has its `CKA_EXTRACTABLE` attribute set to CK_FALSE, so does the derived key. If not, then the derived key's `CKA_EXTRACTABLE` attribute is set either from the supplied template or from a default value.
- The derived key's `CKA_ALWAYS_SENSITIVE` attribute is set to CK_TRUE if and only if the base key has its `CKA_ALWAYS_SENSITIVE` attribute set to CK_TRUE.
- Similarly, the derived key's `CKA_NEVER_EXTRACTABLE` attribute is set to CK_TRUE if and only if the base key has its `CKA_NEVER_EXTRACTABLE` attribute set to CK_TRUE.

### 2.43.6 XORing of a key and data

XORing key derivation, denoted `CKM_XOR_BASE_AND_DATA`, is a mechanism which provides the capability of deriving a secret key by performing a bit XORing of a key pointed to by a base key handle and some data.

This mechanism takes a parameter, a `CK_KEY_DERIVATION_STRING_DATA` structure, which specifies the data with which to XOR the original key's value.

For example, if the value of the base key is 0x01234567, and the value of the data is 0x89ABCDEF, then the value of the derived key will be taken from a buffer containing the string 0x88888888.

- If no length or key type is provided in the template, then the key produced by this mechanism will be a generic secret key. Its length will be equal to the minimum of the lengths of the data and the value of the original key.
- If no key type is provided in the template, but a length is, then the key produced by this mechanism will be a generic secret key of the specified length.
- If no length is provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn't, an error will be returned.
- If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism will be of the specified type and length.

If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more bytes than are available by taking the shorter of the data and the original key's value, an error is generated.

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

- If the base key has its `CKA_SENSITIVE` attribute set to CK_TRUE, so does the derived key. If not, then the derived key's `CKA_SENSITIVE` attribute is set either from the supplied template or from a default value.
Similarly, if the base key has its `CKA_EXTRACTABLE` attribute set to `CK_FALSE`, so does the derived key. If not, then the derived key's `CKA_EXTRACTABLE` attribute is set either from the supplied template or from a default value.

The derived key's `CKA_ALWAYS_SENSITIVE` attribute is set to `CK_TRUE` if and only if the base key has its `CKA_ALWAYS_SENSITIVE` attribute set to `CK_TRUE`.

Similarly, the derived key's `CKA_NEVER_EXTRACTABLE` attribute is set to `CK_TRUE` if and only if the base key has its `CKA_NEVER_EXTRACTABLE` attribute set to `CK_TRUE`.

### 2.43.7 Extraction of one key from another key

Extraction of one key from another key, denoted `CKM_EXTRACT_KEY_FROM_KEY`, is a mechanism which provides the capability of creating one secret key from the bits of another secret key.

This mechanism has a parameter, a `CK_EXTRACT_PARAMS`, which specifies which bit of the original key should be used as the first bit of the newly-derived key.

We give an example of how this mechanism works. Suppose a token has a secret key with the 4-byte value 0x329F84A9. We will derive a 2-byte secret key from this key, starting at bit position 21 (i.e., the value of the parameter to the `CKM_EXTRACT_KEY_FROM_KEY` mechanism is 21).

1. We write the key's value in binary: 0011 0010 1001 1111 1000 0100 1010 1001. We regard this binary string as holding the 32 bits of the key, labeled as b0, b1, ..., b31.
2. We then extract 16 consecutive bits (i.e., 2 bytes) from this binary string, starting at bit b21. We obtain the binary string 1001 0101 0010 0110.
3. The value of the new key is thus 0x9526.

Note that when constructing the value of the derived key, it is permissible to wrap around the end of the binary string representing the original key's value.

If the original key used in this process is sensitive, then the derived key must also be sensitive for the derivation to succeed.

- If no length or key type is provided in the template, then an error will be returned.
- If no key type is provided in the template, but a length is, then the key produced by this mechanism will be a generic secret key of the specified length.
- If no length is provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn't, an error will be returned.
- If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism will be of the specified type and length.

If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more bytes than the original key has, an error is generated.

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

- If the base key has its `CKA_SENSITIVE` attribute set to `CK_TRUE`, so does the derived key. If not, then the derived key's `CKA_SENSITIVE` attribute is set either from the supplied template or from a default value.
- Similarly, if the base key has its `CKA_EXTRACTABLE` attribute set to `CK_FALSE`, so does the derived key. If not, then the derived key's `CKA_EXTRACTABLE` attribute is set either from the supplied template or from a default value.
- The derived key's `CKA_ALWAYS_SENSITIVE` attribute is set to `CK_TRUE` if and only if the base key has its `CKA_ALWAYS_SENSITIVE` attribute set to `CK_TRUE`.
- Similarly, the derived key's `CKA_NEVER_EXTRACTABLE` attribute is set to `CK_TRUE` if and only if the base key has its `CKA_NEVER_EXTRACTABLE` attribute set to `CK_TRUE`.
Table 169, CMS Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR¹</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_CMS_SIG</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.44.1 Definitions

Mechanisms:

CKM_CMS_SIG

2.44.2 CMS Signature Mechanism Objects

These objects provide information relating to the CKM_CMS_SIG mechanism. CKM_CMS_SIG mechanism object attributes represent information about supported CMS signature attributes in the token. They are only present on tokens supporting the CKM_CMS_SIG mechanism, but must be present on those tokens.

Table 170, CMS Signature Mechanism Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_REQUIRED_CMS_ATTRIBUTE S</td>
<td>Byte array</td>
<td>Attributes the token always will include in the set of CMS signed attributes</td>
</tr>
<tr>
<td>CKA_DEFAULT_CMS_ATTRIBUTES</td>
<td>Byte array</td>
<td>Attributes the token will include in the set of CMS signed attributes in the absence of any attributes specified by the application</td>
</tr>
<tr>
<td>CKA_SUPPORTED_CMS_ATTRIBUTES</td>
<td>Byte array</td>
<td>Attributes the token may include in the set of CMS signed attributes upon request by the application</td>
</tr>
</tbody>
</table>

The contents of each byte array will be a DER-encoded list of CMS Attributes with optional accompanying values. Any attributes in the list shall be identified with its object identifier, and any values shall be DER-encoded. The list of attributes is defined in ASN.1 as:

```
Attributes ::= SET SIZE (1..MAX) OF Attribute
Attribute ::= SEQUENCE {
  attrType OBJECT IDENTIFIER,
  attrValues SET OF ANY DEFINED BY OBJECT IDENTIFIER OPTIONAL
}
```

The client may not set any of the attributes.

2.44.3 CMS mechanism parameters

- CK_CMS_SIG_PARAMS, CK_CMS_SIG_PARAMS_PTR

CK_CMS_SIG_PARAMS is a structure that provides the parameters to the CKM_CMS_SIG mechanism. It is defined as follows:

```
typedef struct CK_CMS_SIG_PARAMS {
```
The fields of the structure have the following meanings:

- `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.

- `pSigningMechanism`: Mechanism to use when signing a constructed `CMS_SignedAttributes` value. E.g. `CKM_SHA1_RSA_PKCS`.

- `pDigestMechanism`: Mechanism to use when digesting the data. Value shall be `NULL_PTR` when the digest mechanism to use follows from the `pSigningMechanism` parameter.

- `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.

- `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.

- `ulRequestedAttributesLen`: Length in bytes of the value pointed to by `pRequestedAttributes`.

- `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`.

- `ulRequiredAttributesLen`: Length in bytes, of the value pointed to by `pRequiredAttributes`. 
2.44.4 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 present through the parameters `pRequiredAttributes`. The signature is computed in accordance with the parameter `pSigningMechanism`.

When this mechanism is used in successful calls to `C_Sign` or `C_SignFinal`, the `pSignature` return value will point to a DER-encoded value of type `SignerInfo`. `SignerInfo` is defined in ASN.1 as follows (for a complete definition of all fields and types, see RFC 2630):

```
SignerInfo ::= SEQUENCE {
  version CMSVersion,
  sid SignerIdentifier,
  digestAlgorithm DigestAlgorithmIdentifier,
  signedAttrs [0] IMPLICIT SignedAttributes OPTIONAL,
  signatureAlgorithm SignatureAlgorithmIdentifier,
  signature SignatureValue,
  unsignedAttr [1] IMPLICIT UnsignedAttributes OPTIONAL
}
```

The `certificateHandle` parameter, when set, helps the token populate the `sid` field of the `SignerInfo` value. If `certificateHandle` is `NULL_PTR` the choice of a suitable certificate reference in the `SignerInfo` result value is left to the token (the token could, e.g., interact with the user).

This mechanism shall not be used in calls to `C_Verify` or `C_VerifyFinal` (use the `pSigningMechanism` mechanism instead).

For the `pRequiredAttributes` field, the token may have to interact with the user to find out whether to accept a proposed value or not. The token should never accept any proposed attribute values without some kind of confirmation from its owner (but this could be through, e.g., configuration or policy settings and not direct interaction). If a user rejects proposed values, or the signature request as such, the value `CKR_FUNCTION_REJECTED` shall be returned.

When possible, applications should use the `CKM_CMS_SIG` mechanism when generating CMS-compatible signatures rather than lower-level mechanisms such as `CKM_SHA1_RSA_PKCS`. This is especially true when the signatures are to be made on content that the token is able to present to a user. Exceptions may include those cases where the token does not support a particular signing attribute. Note however that the token may refuse usage of a particular signature key unless the content to be signed is known (i.e. the `CKM_CMS_SIG` mechanism is used).

When a token does not have presentation capabilities, the PKCS #11-aware application may avoid sending the whole message to the token by electing to use a suitable signature mechanism (e.g. `CKM_RSA_PKCS`) as the `pSigningMechanism` value in the `CK_CMS_SIG_PARAMS` structure, and digesting the message itself before passing it to the token.

PKCS #11-aware applications making use of tokens with presentation capabilities, should attempt to provide messages to be signed by the token in a format possible for the token to present to the user. Tokens that receive multipart MIME-messages for which only certain parts are possible to present may fail the signature operation with a return value of `CKR_DATA_INVALID`, but may also choose to add a signing attribute indicating which parts of the message were possible to present.

2.45 Blowfish

Blowfish, a secret-key block cipher. It is a Feistel network, iterating a simple encryption function 16 times. The block size is 64 bits, and the key can be any length up to 448 bits. Although there is a complex
initialization phase required before any encryption can take place, the actual encryption of data is very
efficient on large microprocessors.

Table 171, Blowfish Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Encrypt &amp; Decrypt</td>
</tr>
<tr>
<td>CKM_BLOWFISH_CBC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_BLOWFISH_CBC_PAD</td>
<td>✓</td>
</tr>
</tbody>
</table>

2.45.1 Definitions

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

Mechanisms:
- CKM_BLOWFISH_KEY_GEN
- CKM_BLOWFISH_CBC
- CKM_BLOWFISH_CBC_PAD

2.45.2 BLOWFISH secret key objects

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

Table 172, BLOWFISH Secret Key Object

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE</td>
<td>Byte array</td>
<td>Key value the key can be any length up to 448 bits. Bit length restricted to a byte array.</td>
</tr>
<tr>
<td>CKA_VALUE_LEN</td>
<td>CK_ULONG</td>
<td>Length in bytes of key value</td>
</tr>
</tbody>
</table>

Refer to [PKCS11-Base] table 11 for footnotes

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

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_BLOWFISH;
CK_UTF8CHAR label[] = "A blowfish secret key object";
CK_BYTE value[16] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_ENCRYPT, &true, sizeof(true)},
    {CKA_VALUE, value, sizeof(value)}
};
```
2.45.3 Blowfish key generation

The Blowfish key generation mechanism, denoted `CKM_BLOWFISH_KEY_GEN`, is a key generation mechanism Blowfish.

It does not have a parameter.

The mechanism generates Blowfish keys with a particular length, as specified in the `CKA_VALUE_LEN` attribute of the template for the key.

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

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

2.45.4 Blowfish-CBC

Blowfish-CBC, denoted `CKM_BLOWFISH_CBC`, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping.

It has a parameter, a 8-byte initialization vector.

This mechanism can wrap and unwrap any secret key. For wrapping, the mechanism encrypts the value of the `CKA_VALUE` attribute of the key that is wrapped, padded on the trailing end with up to block size minus one null bytes so that the resulting length is a multiple of the block size. The output data is the same length as the padded input data. It does not wrap the key type, key length, or any other information about the key; the application must convey these separately.

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

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input Length</th>
<th>Output Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>BLOWFISH</td>
<td>Multiple of block size</td>
<td>Same as input length</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>BLOWFISH</td>
<td>Multiple of block size</td>
<td>Same as input length</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>BLOWFISH</td>
<td>Any</td>
<td>Input length rounded up to multiple of the block size</td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>BLOWFISH</td>
<td>Multiple of block size</td>
<td>Determined by type of key being unwrapped or <code>CKA_VALUE_LEN</code></td>
</tr>
</tbody>
</table>

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure specify the supported range of BLOWFISH key sizes, in bytes.

2.45.5 Blowfish-CBC with PKCS padding

Blowfish-CBC-PAD, denoted `CKM_BLOWFISH_CBC_PAD`, is a mechanism for single- and multiple-part encryption and decryption, key wrapping and key unwrapping, cipher-block chaining mode and the block cipher padding method detailed in PKCS #7.

It has a parameter, a 8-byte initialization vector.

The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified for the `CKA_VALUE_LEN` attribute.
The entries in the table below for data length constraints when wrapping and unwrapping keys do not apply to wrapping and unwrapping private keys.

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

Table 174, BLOWFISH-CBC with PKCS Padding: Key and Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input Length</th>
<th>Output Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>BLOWFISH</td>
<td>Any</td>
<td>Input length rounded up to multiple of the block size</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>BLOWFISH</td>
<td>Multiple of block size</td>
<td>Between 1 and block length block size bytes shorter than input length</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>BLOWFISH</td>
<td>Any</td>
<td>Input length rounded up to multiple of the block size</td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>BLOWFISH</td>
<td>Multiple of block size</td>
<td>Between 1 and block length block size bytes shorter than input length</td>
</tr>
</tbody>
</table>

2.46 Twofish


2.46.1 Definitions

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

Mechanisms:

- CKM_TWOFISH_KEY_GEN
- CKM_TWOFISH_CBC
- CKM_TWOFISH_CBC_PAD

2.46.2 Twofish secret key objects

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

Table 175, Twofish Secret Key Object

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE¹⁴,⁶,⁷</td>
<td>Byte array</td>
<td>Key value 128-, 192-, or 256-bit key</td>
</tr>
<tr>
<td>CKA_VALUE_LEN²,³</td>
<td>CK_ULONG</td>
<td>Length in bytes of key value</td>
</tr>
</tbody>
</table>

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

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

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_TWOFISH;
CK_UTF8CHAR label[] = "A twofish secret key object";
CK_BYTE value[16] = {...};
CK_BBOOL true = CK_TRUE;
```
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_ENCRYPT, &true, sizeof(true)},
    {CKA_VALUE, value, sizeof(value)}
};

2.46.3 Twofish key generation
The Twofish key generation mechanism, denoted CKM_TWOFISH_KEY_GEN, is a key generation mechanism Twofish.
It does not have a parameter.
The mechanism generates Blowfish keys with a particular length, as specified in the CKA_VALUE_LEN attribute of the template for the key.
The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, and CKA_VALUE attributes to the new key. Other attributes supported by the key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.
For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of key sizes, in bytes.

2.46.4 Twofish -CBC
Twofish-CBC, denoted CKM_TWOFISH_CBC, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping.
It has a parameter, a 16-byte initialization vector.

2.46.5 Twofish-CBC with PKCS padding
Twofish-CBC-PAD, denoted CKM_TWOFISH_CBC_PAD, is a mechanism for single- and multiple-part encryption and decryption, key wrapping and key unwrapping, cipher-block chaining mode and the block cipher padding method detailed in PKCS #7.
It has a parameter, a 16-byte initialization vector.
The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified for the CKA_VALUE_LEN attribute.

2.47 CAMELLIA
Camellia is a block cipher with 128-bit block size and 128-, 192-, and 256-bit keys, similar to AES.
Camellia is described e.g. in IETF RFC 3713.

Table 176, Camellia Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digests</th>
<th>Gen - Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_CAMELLIA_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
2.47.1 Definitions

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

Mechanisms:

- CKM_CAMELLIA_KEY_GEN
- CKM_CAMELLIA_ECB
- CKM_CAMELLIA_CBC
- CKM_CAMELLIA_MAC
- CKM_CAMELLIA_MAC_GENERAL
- CKM_CAMELLIA_CBC_PAD
- CKM_CAMELLIACBC_ENCRYPT_DATE
- CKM_CAMELLIA_CBC_ENCRYPT_DATE

2.47.2 Camellia secret key objects

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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE</td>
<td>Byte array</td>
<td>Key value (16, 24, 32 bytes)</td>
</tr>
<tr>
<td>CKA_VALUE_LEN</td>
<td>CK_ULONG</td>
<td>Length in bytes of key value</td>
</tr>
</tbody>
</table>

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

```c
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[] = {
```
{CKA_CLASS, &class, sizeof(class)},
{CKA_KEY_TYPE, &keyType, sizeof(keyType)},
{CKA_TOKEN, &true, sizeof(true)},
{CKA_LABEL, label, sizeof(label)-1},
{CKA_ENCRYPT, &true, sizeof(true)},
{CKA_VALUE, value, sizeof(value)}
};

2.47.3 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.

2.47.4 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 179, Camellia-CBC: Key and Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>CKK_CAMELLIA</td>
<td>multiple of block size</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>CKK_CAMELLIA</td>
<td>multiple of block size</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>CKK_CAMELLIA</td>
<td>any</td>
<td>input length rounded up to multiple of block size</td>
<td></td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>CKK_CAMELLIA</td>
<td>multiple of block size</td>
<td>determined by type of key being unwrapped or CKA_VALUE_LEN</td>
<td></td>
</tr>
</tbody>
</table>

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

2.47.5 Camellia-CBC

Camellia-CBC, denoted CKM_CAMELLIA_CBC, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on Camellia and cipher-block chaining mode. It has a parameter, a 16-byte initialization vector.

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

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

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

Table 179, Camellia-CBC: Key and Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>CKK_CAMELLIA</td>
<td>multiple of block size</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>CKK_CAMELLIA</td>
<td>multiple of block size</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>CKK_CAMELLIA</td>
<td>any</td>
<td>input length rounded up to multiple of the block size</td>
<td></td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>CKK_CAMELLIA</td>
<td>multiple of block size</td>
<td>determined by type of key being unwrapped or CKA_VALUE_LEN</td>
<td></td>
</tr>
</tbody>
</table>

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of Camellia key sizes, in bytes.
2.47.6 Camellia-CBC with PKCS padding

Camellia-CBC with PKCS padding, denoted CKM_CAMELLIA_CBC_PAD, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on Camellia; cipher-block chaining mode; and the block cipher padding method detailed in PKCS #7.

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

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

In addition to being able to wrap and unwrap secret keys, this mechanism can wrap and unwrap RSA, Diffie-Hellman, X9.42 Diffie-Hellman, EC (also related to ECDSA) and DSA private keys (see Section TBA 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.

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

Table 180, Camellia-CBC with PKCS Padding: Key and Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>CKK_CAMELLIA</td>
<td>any</td>
<td>input length rounded up to multiple of the block size</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>CKK_CAMELLIA</td>
<td>multiple of block size</td>
<td>between 1 and block size bytes shorter than input length</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>CKK_CAMELLIA</td>
<td>any</td>
<td>input length rounded up to multiple of the block size</td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>CKK_CAMELLIA</td>
<td>multiple of block size</td>
<td>between 1 and block length bytes shorter than input length</td>
</tr>
</tbody>
</table>

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

2.47.7 CAMELLIA with Counter mechanism parameters

♦ CK_CAMELLIA_CTR_PARAMS; CK_CAMELLIA_CTR_PARAMS_PTR

CK_CAMELLIA_CTR_PARAMS is a structure that provides the parameters to the CKM_CAMELLIA_CTR mechanism. It is defined as follows:

```
typedef struct CK_CAMELLIA_CTR_PARAMS {
    CK_ULONG ulCounterBits;
    CK_BYTE cb[16];
} CK_CAMELLIA_CTR_PARAMS;
```

ulCounterBits specifies the number of bits in the counter block (cb) that shall be incremented. This number shall be such that 0 < ulCounterBits <= 128. For any values outside this range the mechanism shall return CKR_MECHANISM_PARAM_INVALID.

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

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

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

CK_CAMELLIA_CTR_PARAMS_PTR is a pointer to a CK_CAMELLIA_CTR_PARAMS.

### 2.47.8 General-length Camellia-MAC

General-length Camellia-MAC, denoted CKM_CAMELLIA_MAC General, is a mechanism for single- and multiple-part signatures and verification, based on Camellia and data authentication as defined in [CAMELLIA].

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 Camellia cipher block produced in the MACing process.

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>CKK_CAMELLIA</td>
<td>any</td>
<td>1-block size, as specified in parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>CKK_CAMELLIA</td>
<td>any</td>
<td>1-block size, as specified in parameters</td>
</tr>
</tbody>
</table>

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

### 2.47.9 Camellia-MAC

Camellia-MAC, denoted by CKM_CAMELLIA_MAC, is a special case of the general-length Camellia-MAC mechanism. Camellia-MAC always produces and verifies MACs that are half the block size in length.

It does not have a parameter.

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>CKK_CAMELLIA</td>
<td>any</td>
<td>½ block size (8 bytes)</td>
</tr>
<tr>
<td>C_Verify</td>
<td>CKK_CAMELLIA</td>
<td>any</td>
<td>½ block size (8 bytes)</td>
</tr>
</tbody>
</table>

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of Camellia key sizes, in bytes.
2.48 Key derivation by data encryption - Camellia

These mechanisms allow derivation of keys using the result of an encryption operation as the key value. They are for use with the C_DeriveKey function.

2.48.1 Definitions

Mechanisms:

CKM_CAMELLIA_ECB_ENCRYPT_DATA
CKM_CAMELLIA_CBC_ENCRYPT_DATA

```
typedef struct CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS {
    CK_BYTE      iv[16];
    CK_BYTE_PTR  pData;
    CK_ULONG     length;
} CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS;
```

2.48.2 Mechanism Parameters

Uses CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS, and CK_KEYDERIVATION_STRING_DATA.

Table 183, Mechanism Parameters for Camellia-based key derivation

<table>
<thead>
<tr>
<th>CKM_CAMELLIA_ECB_ENCRYPT_DATA</th>
<th>Uses CK_KEYDERIVATION_STRING_DATA structure. Parameter is the data to be encrypted and must be a multiple of 16 long.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_CAMELLIA_CBC_ENCRYPT_DATA</td>
<td>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.</td>
</tr>
</tbody>
</table>

2.49 ARIA

ARIA is a block cipher with 128-bit block size and 128-, 192-, and 256-bit keys, similar to AES. ARIA is described in NSRI “Specification of ARIA”.

Table 184, ARIA Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_ARIA_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mechanism</td>
<td>Encrypt &amp; Decrypt</td>
<td>Sign &amp; Verify</td>
<td>SR &amp; VR</td>
<td>Digest</td>
<td>Gen. Key/Key Pair</td>
<td>Wrap &amp; Unwrap</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-------------------</td>
<td>--------------</td>
<td>---------</td>
<td>--------</td>
<td>------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>CKM_ARIA_MAC_GENERAL</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ARIA_MAC</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ARIA_ECB_ENCRYPT_DATA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_ARIA_CBC_ENCRYPT_DATA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

### 2.49.1 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

### 2.49.2 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 185, ARIA Secret Key Object Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE</td>
<td>Byte array</td>
<td>Key value (16, 24, or 32 bytes)</td>
</tr>
<tr>
<td>CKA_VALUE_LEN</td>
<td>CK_ULONG</td>
<td>Length in bytes of key value</td>
</tr>
</tbody>
</table>

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

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

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_ARIA;
CK_UTF8CHAR label[] = "An ARIA secret key object";
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_ENCRYPT, &true, sizeof(true)},
};
```
{CKA_VALUE, value, sizeof(value)}

2.49.3 ARIA key generation

The ARIA key generation mechanism, denoted CKM_ARIA_KEY_GEN, is a key generation mechanism for Aria.

It does not have a parameter.

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

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

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

2.49.4 ARIA-ECB

ARIA-ECB, denoted CKM_ARIA_ECB, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on Aria and electronic codebook mode.

It does not have a parameter.

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

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

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>CKK_ARIA</td>
<td>multiple of block size</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>CKK_ARIA</td>
<td>multiple of block size</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>CKK_ARIA</td>
<td>any</td>
<td>input length rounded up to multiple of block size</td>
<td></td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>CKK_ARIA</td>
<td>multiple of block size</td>
<td>determined by type of key being unwrapped or CKA_VALUE_LEN</td>
<td></td>
</tr>
</tbody>
</table>

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

2.49.5 ARIA-CBC

ARIA-CBC, denoted CKM_ARIA_CBC, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on ARIA and cipher-block chaining mode.
It has a parameter, a 16-byte initialization vector.

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>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>CKK_ARIA</td>
<td>multiple of block size</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>CKK_ARIA</td>
<td>multiple of block size</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>CKK_ARIA</td>
<td>any</td>
<td>input length rounded up to multiple of the block size</td>
<td></td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>CKK_ARIA</td>
<td>multiple of block size</td>
<td>determined by type of key being unwrapped or CKA_VALUE_LEN</td>
<td></td>
</tr>
</tbody>
</table>

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

### 2.49.6 ARIA-CBC with PKCS padding

ARIA-CBC with PKCS padding, denoted **CKM_ARIA_CBC_PAD**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on ARIA; cipher-block chaining mode; and the block cipher padding method detailed in PKCS #7.

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

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

In addition to being able to wrap and unwrap secret keys, this mechanism can wrap and unwrap RSA, Diffie-Hellman, X9.42 Diffie-Hellman, EC (also related to ECDSA) and DSA private keys (see Section TBA 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.

Constraints on key types and the length of data are summarized in the following table:
Table 188, ARIA-CBC with PKCS Padding: Key and Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>CKK_ARIA</td>
<td>any</td>
<td>input length rounded up to multiple of the block size</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>CKK_ARIA</td>
<td>multiple of block size</td>
<td>between 1 and block size bytes shorter than input length</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>CKK_ARIA</td>
<td>any</td>
<td>input length rounded up to multiple of the block size</td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>CKK_ARIA</td>
<td>multiple of block size</td>
<td>between 1 and block length bytes shorter than input length</td>
</tr>
</tbody>
</table>

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

2.49.7 General-length ARIA-MAC

General-length ARIA-MAC, denoted CKM_ARIA_MAC_GENERAL, is a mechanism for single- and multiple-part signatures and verification, based on ARIA and data authentication as defined in [FIPS 113]. It has a parameter, a CK_MAC_GENERAL_PARAMS structure, which specifies the output length desired from the mechanism.

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

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

Table 189, General-length ARIA-MAC: Key and Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>CKK_ARIA</td>
<td>any</td>
<td>1-block size, as specified in parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>CKK_ARIA</td>
<td>any</td>
<td>1-block size, as specified in parameters</td>
</tr>
</tbody>
</table>

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

2.49.8 ARIA-MAC

ARIA-MAC, denoted by CKM_ARIA_MAC, is a special case of the general-length ARIA-MAC mechanism. ARIA-MAC always produces and verifies MACs that are half the block size in length. It does not have a parameter.

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

Table 190, ARIA-MAC: Key and Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>CKK_ARIA</td>
<td>any</td>
<td>½ block size (8 bytes)</td>
</tr>
<tr>
<td>C_Verify</td>
<td>CKK_ARIA</td>
<td>any</td>
<td>½ block size (8 bytes)</td>
</tr>
</tbody>
</table>

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

2.50 Key derivation by data encryption - ARIA

These mechanisms allow derivation of keys using the result of an encryption operation as the key value. They are for use with the C_DeriveKey function.
2.50.1 Definitions

Mechanisms:

CKM_ARIA_ECB_ENCRYPT_DATA
CKM_ARIA_CBC_ENCRYPT_DATA

```c
typedef struct CK_ARIA_CBC_ENCRYPT_DATA_PARAMS {
    CK_BYTE      iv[16];
    CK_BYTE_PTR  pData;
    CK_ULONG     length;
} CK_ARIA_CBC_ENCRYPT_DATA_PARAMS;
```

typedef CK_ARIA_CBC_ENCRYPT_DATA_PARAMS CK_PTR CK_ARIA_CBC_ENCRYPT_DATA_PARAMS_PTR;

2.50.2 Mechanism Parameters

Uses CK_ARIA_CBC_ENCRYPT_DATA_PARAMS, and CK_KEY_DERIVATION_STRING_DATA.

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_ARIA_ECB_ENCRYPT_DATA</td>
<td>Uses CK_KEY_DERIVATION_STRING_DATA structure. Parameter is the data to be encrypted and must be a multiple of 16 long.</td>
</tr>
<tr>
<td>CKM_ARIA_CBC_ENCRYPT_DATA</td>
<td>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.</td>
</tr>
</tbody>
</table>

2.51 SEED

SEED is a symmetric block cipher developed by the South Korean Information Security Agency (KISA). It has a 128-bit key size and a 128-bit block size.

Its specification has been published as Internet [RFC 4269].

RFCs have been published defining the use of SEED in


TLS cipher suites that use SEED include:

```
CipherSuite TLS_RSA_WITH_SEED_CBC_SHA = { 0x00, 0x96};
CipherSuite TLS_DH_DSS_WITH_SEED_CBC_SHA = { 0x00, 0x97};
CipherSuite TLS_DH_RSA_WITH_SEED_CBC_SHA = { 0x00, 0x98};
CipherSuite TLS_DHE_DSS_WITH_SEED_CBC_SHA = { 0x00, 0x99};
```
CipherSuite TLS_DHE_RSA_WITH_SEED_CBC_SHA = { 0x00, 0x9A};
CipherSuite TLS_DH_anon_WITH_SEED_CBC_SHA = { 0x00, 0x9B};

As with any block cipher, it can be used in the ECB, CBC, OFB and CFB modes of operation, as well as in a MAC algorithm such as HMAC.

OIDs have been published for all these uses. A list may be seen at http://www.alvestrand.no/objectid/1.2.410.200004.1.html

Table 192, SEED Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR¹</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_SEED_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SEED_ECB</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SEED_CBC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SEED_CBC_PAD</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_SEED_MAC_GENERAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SEED_MAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_SEED_ECB_ENCRYPT_DATA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SEED_CBC_ENCRYPT_DATA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

2.51.1 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:

CKM_SEED_KEY_GEN
CKM_SEED_ECB
CKM_SEED_CBC
CKM_SEED_MAC
CKM_SEED_MAC_GENERAL
CKM_SEED_CBC_PAD

For all of these mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO are always 16.

2.51.2 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:
Table 193, SEED Secret Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE^{1,4,6,7}</td>
<td>Byte array</td>
<td>Key value (always 16 bytes long)</td>
</tr>
</tbody>
</table>

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

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

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_SEED;
CK_UTF8CHAR label[] = “A SEED 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_VALUE, value, sizeof(value)}
};
```

2.51.3 SEED key generation

The SEED key generation mechanism, denoted CKM_SEED_KEY_GEN, is a key generation mechanism for SEED.

It does not have a parameter.

The mechanism generates SEED keys.

The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, and CKA_VALUE attributes to the new key. Other attributes supported by the SEED 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.

2.51.4 SEED-ECB

SEED-ECB, denoted CKM_SEED_ECB, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on SEED and electronic codebook mode.

It does not have a parameter.

2.51.5 SEED-CBC

SEED-CBC, denoted CKM_SEED_CBC, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on SEED and cipher-block chaining mode.

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

2.51.6 SEED-CBC with PKCS padding

SEED-CBC with PKCS padding, denoted CKM_SEED_CBC_PAD, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on SEED; cipher-block chaining mode; and the block cipher padding method detailed in PKCS #7.

It has a parameter, a 16-byte initialization vector.
2.51.7 General-length SEED-MAC

General-length SEED-MAC, denoted **CKM_SEED_MAC_GENERAL**, is a mechanism for single- and multiple-part signatures and verification, based on SEED and data authentication as defined in 0.

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

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

2.51.8 SEED-MAC

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

It does not have a parameter.

2.52 Key derivation by data encryption - SEED

These mechanisms allow derivation of keys using the result of an encryption operation as the key value. They are for use with the C_DeriveKey function.

2.52.1 Definitions

Mechanisms:

- **CKM_SEED_ECB_ENCRYPT_DATA**
- **CKM_SEED_CBC_ENCRYPT_DATA**

```c
typedef struct CK_SEED_CBC_ENCRYPT_DATA_PARAMS
    CK_CBC_ENCRYPT_DATA_PARAMS;
```

```c
typedef CK_CBC_ENCRYPT_DATA_PARAMS CK_PTR
    CK_CBC_ENCRYPT_DATA_PARAMS_PTR;
```

2.52.2 Mechanism Parameters

Table 194, Mechanism Parameters for SEED-based key derivation

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CKM_SEED_ECB_ENCRYPT_DATA</strong></td>
<td>Uses <strong>CK_KEY_DERIVATION_STRING_DATA</strong> structure. Parameter is the data to be encrypted and must be a multiple of 16 long.</td>
</tr>
<tr>
<td><strong>CKM_SEED_CBC_ENCRYPT_DATA</strong></td>
<td>Uses <strong>CK_CBC_ENCRYPT_DATA_PARAMS</strong>. Parameter is an 16 byte IV value followed by the data. The data value part must be a multiple of 16 bytes long.</td>
</tr>
</tbody>
</table>

2.53 OTP

2.53.1 Usage overview

OTP tokens represented as PKCS #11 mechanisms may be used in a variety of ways. The usage cases can be categorized according to the type of sought functionality.
### 2.53.2 Case 1: Generation of OTP values

Figure 1: Retrieving OTP values through `C_Sign`

Figure 1 shows an integration of PKCS #11 into an application that needs to authenticate users holding OTP tokens. In this particular example, a connected hardware token is used, but a software token is equally possible. The application invokes `C_Sign` to retrieve the OTP value from the token. In the example, the application then passes the retrieved OTP value to a client API that sends it via the network to an authentication server. The client API may implement a standard authentication protocol such as RADIUS [RFC 2865] or EAP [RFC 3748], or a proprietary protocol such as that used by RSA Security's ACE/Agent® software.
2.53.3 Case 2: Verification of provided OTP values

Figure 2: Server-side verification of OTP values

Figure 2 illustrates the server-side equivalent of the scenario depicted in Figure 1. In this case, a server application invokes `C_Verify` with the received OTP value as the signature value to be verified.

2.53.4 Case 3: Generation of OTP keys

Client Application

C_GenerateKey()

PKCS #11 Library

Internal Token API

Token (or software version thereof)
Figure 3: Generation of an OTP key

Figure 3 shows an integration of PKCS #11 into an application that generates OTP keys. The application invokes \texttt{C\_GenerateKey} to generate an OTP key of a particular type on the token. The key may subsequently be used as a basis to generate OTP values.

\section*{2.53.5 OTP objects}

\subsection*{2.53.5.1 Key objects}

OTP key objects (object class \texttt{CKO\_OTP\_KEY}) hold secret keys used by OTP tokens. The following table defines the attributes common to all OTP keys, in addition to the attributes defined for secret keys, all of which are inherited by this class:
Table 195: Common OTP key attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| CKA_OTP_FORMAT | CK ULONG | Format of OTP values produced with this key:  
CK_OTP_FORMAT_DECIMAL = Decimal (default) (UTF8-encoded)  
CK_OTP_FORMAT_HEXDECIMAL = Hexadecimal (UTF8-encoded)  
CK_OTP_FORMAT_ALPHANUMERIC = Alphanumeric (UTF8-encoded)  
CK_OTP_FORMAT_BINARY = Only binary values. |
| CKA_OTP_LENGTH⁹ | CK ULONG | Default length of OTP values (in the CKA_OTP_FORMAT) produced with this key. |
| CKA_OTP_USER_FRIENDLY_MODE⁹ | 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_REQUIREMENT⁹ | 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_REQUIREMENT⁹ | 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. |
<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| CKA_OTP_COUNTER_REQUIREMENTS | 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_REQUIREMENTS | 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. ulValueLen = 0). |
| CKA_OTP_TIME | RFC 2279 string | Value of the associated internal UTC time in the form YYYYMMDDhhmmss. Default value is empty (i.e. ulValueLen = 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. ulValueLen = 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. ulValueLen = 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. ulValueLen = 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. ulValueLen = 0). |
| CKA_VALUE | Byte array | Value of the key. |
| CKA_VALUE_LEN | CK_ULONG | Length in bytes of key value. |

Refer to [PKCS11-Base] table 11 for footnotes.
Note: A Cryptoki library may support PIN-code caching in order to reduce user interactions. An OTP-
PKCS #11 application should therefore always consult the state of the CKA_OTP_PIN_REQUIREMENT
attribute before each call to C_SignInit, as the value of this attribute may change dynamically.

For OTP tokens with multiple keys, the keys may be enumerated using C_FindObjects. The
CKA_OTP_SERVICE_IDENTIFIER and/or the CKA_OTP_SERVICE_LOGO attribute may be used to
distinguish between keys. The actual choice of key for a particular operation is however application-
specific and beyond the scope of this document.

For all OTP keys, the CKA_ALLOWED_MECHANISMS attribute should be set as required.

### 2.53.6 OTP-related notifications

This document extends the set of defined notifications as follows:

- **CKN_OTP_CHANGED**: Cryptoki is informing the application that the OTP for a key on a
  connected token just changed. This notification is particularly useful when applications wish to display
  the current OTP value for time-based mechanisms.

### 2.53.7 OTP mechanisms

The following table shows, for the OTP mechanisms defined in this document, their support by different
cryptographic operations. For any particular token, of course, a particular operation may well support
only a subset of the mechanisms listed. There is also no guarantee that a token that supports one
mechanism for some operation supports any other mechanism for any other operation (or even supports
that same mechanism for any other operation).

**Table 196: OTP mechanisms vs. applicable functions**

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR(^1)</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_SECURID_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SECURID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_HOTP_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_HOTP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_ACTI_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ACTI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

The remainder of this section will present in detail the OTP mechanisms and the parameters that are
supplied to them.

#### 2.53.7.1 OTP mechanism parameters

- **CK_PARAM_TYPE**

**CK_PARAM_TYPE** is a value that identifies an OTP parameter type. It is defined as follows:

```c
typedef CK_ULONG CK_PARAM_TYPE;
```

The following **CK_PARAM_TYPE** types are defined:
Table 197, OTP parameter types

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CK_OTP_PIN</td>
<td>RFC 2279 string</td>
<td>A UTF8 string containing a PIN for use when computing or verifying PIN-based OTP values.</td>
</tr>
<tr>
<td>CK_OTP_CHALLENGE</td>
<td>Byte array</td>
<td>Challenge to use when computing or verifying challenge-based OTP values.</td>
</tr>
<tr>
<td>CK_OTP_TIME</td>
<td>RFC 2279 string</td>
<td>UTC time value in the form YYYYMMDDhhmms to use when computing or verifying time-based OTP values.</td>
</tr>
<tr>
<td>CK_OTP_COUNTER</td>
<td>Byte array</td>
<td>Counter value to use when computing or verifying counter-based OTP values.</td>
</tr>
<tr>
<td>CK_OTP_FLAGS</td>
<td>CK_FLAGS</td>
<td>Bit flags indicating the characteristics of the sought OTP as defined below.</td>
</tr>
<tr>
<td>CK_OTP_OUTPUT_LENGTH</td>
<td>CK_ULONG</td>
<td>Desired output length (overrides any default value). A Cryptoki library will return CKR_MECHANISM_PARAM_INVALID if a provided length value is not supported.</td>
</tr>
<tr>
<td>CK_OTP_FORMAT</td>
<td>CK_ULONG</td>
<td>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.</td>
</tr>
<tr>
<td>CK_OTP_VALUE</td>
<td>Byte array</td>
<td>An actual OTP value. This parameter type is intended for C_Sign output, see paragraphs below.</td>
</tr>
</tbody>
</table>

The following table defines the possible values for the CK_OTP_FLAGS type:

Table 198: OTP Mechanism Flags

<table>
<thead>
<tr>
<th>Bit flag</th>
<th>Mask</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKF_NEXT_OTP</td>
<td>0x00000001</td>
<td>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 &quot;next&quot; OTP or the library is not capable of generating the next OTP.</td>
</tr>
</tbody>
</table>

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.
<table>
<thead>
<tr>
<th>Bit flag</th>
<th>Mask</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKF_EXCLUDE_TIME</td>
<td>0x00000002</td>
<td>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.</td>
</tr>
<tr>
<td>CKF_EXCLUDE_COUNTER</td>
<td>0x00000004</td>
<td>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.</td>
</tr>
<tr>
<td>CKF_EXCLUDE_CHALLENGE</td>
<td>0x00000008</td>
<td>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.</td>
</tr>
<tr>
<td>CKF_EXCLUDE_PIN</td>
<td>0x00000010</td>
<td>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.</td>
</tr>
<tr>
<td>CKF_USER_FRIENDLY_OTP</td>
<td>0x00000020</td>
<td>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.</td>
</tr>
</tbody>
</table>

Note: Even if CKA_OTP_FORMAT is not set to CK_OTP_FORMAT_BINARY, then there may still be value in setting the CKF_USER_FRIENDLY_OTP flag (assuming CKA_OTP_USER_FRIENDLY_MODE is CK_TRUE, of course) if the intent is for a human to read the generated OTP value, since it may become shorter or otherwise better suited for a user. Applications that do not intend to provide a returned OTP value to a user should not set the CKF_USER_FRIENDLY_OTP flag.

♦ CK_OTP_PARAM; CK_OTP_PARAM_PTR

CK_OTP_PARAM is a structure that includes the type, value, and length of an OTP parameter. It is defined as follows:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>Type of OTP parameter (e.g., numeric, alphanumeric, date)</td>
</tr>
<tr>
<td>value</td>
<td>Value of the OTP parameter</td>
</tr>
<tr>
<td>length</td>
<td>Length of the OTP parameter</td>
</tr>
</tbody>
</table>

pkcs11-curr-v3.0-csprd01
Standards Track Work Product Copyright © OASIS Open 2019. All Rights Reserved.
29 May 2019
Page 222 of 281
typedef struct CK_OTP_PARAM {
    CK_PARAM_TYPE type;
    CK_VOID_PTR pValue;
    CK_ULONG ulValueLen;
} CK_OTP_PARAM;

The fields of the structure have the following meanings:

- **type**  the parameter type
- **pValue**  pointer to the value of the parameter
- **ulValueLen**  length in bytes of the value

If a parameter has no value, then ulValueLen = 0, and the value of pValue is irrelevant. Note that pValue is a "void" pointer, facilitating the passing of arbitrary values. Both the application and the Cryptoki library must ensure that the pointer can be safely cast to the expected type (i.e., without word-alignment errors).

CK_OTP_PARAM_PTR is a pointer to a CK_OTP_PARAM.

CK_OTP_PARAMS; CK_OTP_PARAMS_PTR

CK_OTP_PARAMS is a structure that is used to provide parameters for OTP mechanisms in a generic fashion. It is defined as follows:

typedef struct CK_OTP_PARAMS {
    CK_OTP_PARAM_PTR pParams;
    CK_ULONG ulCount;
} CK_OTP_PARAMS;

The fields of the structure have the following meanings:

- **pParams**  pointer to an array of OTP parameters
- **ulCount**  the number of parameters in the array

CK_OTP_PARAMS_PTR is a pointer to a CK_OTP_PARAMS.

When calling C_SignInit or C_VerifyInit with a mechanism that takes a CK_OTP_PARAMS structure as a parameter, the CK_OTP_PARAMS structure shall be populated in accordance with the CKA_OTP_ X_REQUIREMENT key attributes for the identified key, where X is PIN, CHALLENGE, TIME, or COUNTER.

For example, if CKA_OTP_TIME_REQUIREMENT = CK_OTP_PARAM_MANDATORY, then the CK_OTP_TIME parameter shall be present. If CKA_OTP_TIME_REQUIREMENT = CK_OTP_PARAM_OPTIONAL, then a CK_OTP_TIME parameter may be present. If it is not present, then the library may collect it (during the C_Sign call). If CKA_OTP_TIME_REQUIREMENT = CK_OTP_PARAM_IGNORED, then a provided CK_OTP_TIME parameter will always be ignored.

Additionally, a provided CK_OTP_TIME parameter will always be ignored if CKF_EXCLUDE_TIME is set in a CK_OTP_FLAGS parameter. Similarly, if this flag is set, a library will not attempt to collect the value itself, and it will also instruct the token not to make use of any internal value, subject to token policies. It is an error (CKR_MECHANISM_PARAM_INVALID) to set the CKF_EXCLUDE_TIME flag when the CKA_OTP_TIME_REQUIREMENT attribute is CK_OTP_PARAM_MANDATORY.

The above discussion holds for all CKA_OTP_ X_REQUIREMENT attributes (i.e., CKA_OTP_PIN_REQUIREMENT, CKA_OTP_CHALLENGE_REQUIREMENT, CKA_OTP_COUNTER_REQUIREMENT, CKA_OTP_TIME_REQUIREMENT). A library may set a particular CKA_OTP_ X_REQUIREMENT attribute to CK_OTP_PARAM_OPTIONAL even if it is required.
by the mechanism as long as the token (or the library itself) has the capability of providing the value to the
computation. One example of this is a token with an on-board clock.

In addition, applications may use the CK_OTP_FLAGS, the CK_OTP_FORMAT and the
CKA_OTP_LENGTH parameters to set additional parameters.

CK_OTP_SIGNATURE_INFO, CK_OTP_SIGNATURE_INFO_PTR

CK_OTP_SIGNATURE_INFO is a structure that is returned by all OTP mechanisms in successful calls to
C_Sign (C_SignFinal). The structure informs applications of actual parameter values used in particular
OTP computations in addition to the OTP value itself. It is used by all mechanisms for which the key
belongs to the class CKO_OTP_KEY and is defined as follows:

```c
typedef struct CK_OTP_SIGNATURE_INFO {
    CK_OTP_PARAM_PTR pParams;
    CK_ULONG ulCount;
} CK_OTP_SIGNATURE_INFO;
```

The fields of the structure have the following meanings:

- `pParams` pointer to an array of OTP parameter values
- `ulCount` the number of parameters in the array

After successful calls to C_Sign or C_SignFinal with an OTP mechanism, the `pSignature` parameter will
be set to point to a CK_OTP_SIGNATURE_INFO structure. One of the parameters in this structure will be
the OTP value itself, identified with the CK_OTP_VALUE tag. Other parameters may be present for
informational purposes, e.g. the actual time used in the OTP calculation. In order to simplify OTP
validations, authentication protocols may permit authenticating parties to send some or all of these
parameters in addition to OTP values themselves. Applications should therefore check for their presence
in returned CK_OTP_SIGNATURE_INFO values whenever such circumstances apply.

Since C_Sign and C_SignFinal follows the convention described in [PKCS11-Base] Section 5.2 on
producing output, a call to C_Sign (or C_SignFinal) with `pSignature` set to NULL_PTR will return (in the
`pulSignatureLen` parameter) the required number of bytes to hold the CK_OTP_SIGNATURE_INFO
structure as well as all the data in all its CK_OTP_PARAM components. If an application allocates a
memory block based on this information, it shall therefore not subsequently de-allocate components of
such a received value but rather de-allocate the complete CK_OTP_PARAMS structure itself. A Cryptoki
library that is called with a non-NULL `pSignature` pointer will assume that it points to a contiguous
memory block of the size indicated by the `pulSignatureLen` parameter.

When verifying an OTP value using an OTP mechanism, `pSignature` shall be set to the OTP value itself,
e.g. the value of the CK_OTP_VALUE component of a CK_OTP_PARAM structure returned by a call to
C_Sign. The CK_OTP_PARAM value supplied in the C_VerifyInit call sets the values to use in the
verification operation.

CK_OTP_SIGNATURE_INFO_PTR points to a CK_OTP_SIGNATURE_INFO.

2.53.8 RSA SecurID

2.53.8.1 RSA SecurID secret key objects

RSA SecurID secret key objects (object class CKO_OTP_KEY, key type CKK_SECURID) hold RSA
SecurID secret keys. The following table defines the RSA SecurID secret key object attributes, in
addition to the common attributes defined for this object class:
### Table 199, RSA SecurID secret key object attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_OTP_TIME_INTERVAL</td>
<td>CK_UULONG</td>
<td>Interval between OTP values produced with this key, in seconds. Default is 60.</td>
</tr>
</tbody>
</table>

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

The following is a sample template for creating an RSA SecurID secret key object:

```c
    CK_OBJECT_CLASS class = CKO_OTP_KEY;
    CK_KEY_TYPE keyType = CKK_SECURID;
    CK_DATE endDate = {...};
    CK_UTF8CHAR label[] = "RSA SecurID secret key object";
    CK_BYTE keyId[] = {...};
    CK_UULONG outputFormat = CK_OTP_FORMAT_DECIMAL;
    CK_UULONG outputLength = 6;
    CK_UULONG needPIN = CK_OTP_PARAM_MANDATORY;
    CK_UULONG timeInterval = 60;
    CK_BYTE value[] = {...};
    CK_BBOOL true = CK_TRUE;
    CK_ATTRIBUTE template[] = {
        {CKA_CLASS, &class, sizeof(class)},
        {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
        {CKA_END_DATE, &endDate, sizeof(endDate)},
        {CKA_TOKEN, &true, sizeof(true)},
        {CKA_SENSITIVE, &true, sizeof(true)},
        {CKA_LABEL, label, sizeof(label)-1},
        {CKA_SIGN, &true, sizeof(true)},
        {CKA_VERIFY, &true, sizeof(true)},
        {CKA_ID, keyId, sizeof(keyId)},
        {CKA_OTP_FORMAT, &outputFormat, sizeof(outputFormat)},
        {CKA_OTP_LENGTH, &outputLength, sizeof(outputLength)},
        {CKA_OTP_PIN_REQUIREMENT, &needPIN, sizeof(needPIN)},
        {CKA_OTP_TIME_INTERVAL, &timeInterval, sizeof(timeInterval)},
        {CKA_VALUE, value, sizeof(value)}
    };
```

#### 2.53.8.2 RSA SecurID key generation

The RSA SecurID key generation mechanism, denoted **CKM_SECURID_KEY_GEN**, is a key generation mechanism for the RSA SecurID algorithm.

It does not have a parameter.

The mechanism generates RSA SecurID keys with a particular set of attributes as specified in the template for the key.

The mechanism contributes at least the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_VALUE_LEN**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the RSA SecurID key type may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the **CK_MECHANISM_INFO** structure specify the supported range of SecurID key sizes, in bytes.
2.53.8.3 SecurID OTP generation and validation

**CKM_SECURID** is the mechanism for the retrieval and verification of RSA SecurID OTP values.

The mechanism takes a pointer to a **CK_OTP_PARAMS** structure as a parameter.

When signing or verifying using the **CKM_SECURID** mechanism, **pData** shall be set to NULL_PTR and **ulDataLen** shall be set to 0.

2.53.8.4 Return values

Support for the **CKM_SECURID** mechanism extends the set of return values for **C_Verify** with the following values:

- **CKR_NEW_PIN_MODE**: The supplied OTP was not accepted and the library requests a new OTP computed using a new PIN. The new PIN is set through means out of scope for this document.

- **CKR_NEXT_OTP**: The supplied OTP was correct but indicated a larger than normal drift in the token's internal state (e.g. clock, counter). To ensure this was not due to a temporary problem, the application should provide the next one-time password to the library for verification.

2.53.9 OATH HOTP

2.53.9.1 OATH HOTP secret key objects

HOTP secret key objects (object class **CKO_OTP_KEY**, key type **CKK_HOTP**) hold generic secret keys and associated counter values.

The **CKA_OTP_COUNTER** value may be set at key generation; however, some tokens may set it to a fixed initial value. Depending on the token's security policy, this value may not be modified and/or may not be revealed if the object has its **CKA_SENSITIVE** attribute set to **CK_TRUE** or its **CKA_EXTRACTABLE** attribute set to **CK_FALSE**.

For HOTP keys, the **CKA_OTP_COUNTER** value shall be an 8 bytes unsigned integer in big endian (i.e. network byte order) form. The same holds true for a **CK_OTP_COUNTER** value in a **CK_OTP_PARAM** structure.

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

```c
CK_OBJECT_CLASS class = CKO_OTP_KEY;
CK_KEY_TYPE keyType = CKK_HOTP;
CK_UTF8CHAR label[] = "HOTP secret key object";
CK_BYTE keyId[]={...};
CK_ULONGLONG outputFormat = CK_OTP_FORMAT_DECIMAL;
CK_ULONGLONG outputLength = 6;
CK_DATE endDate = {...};
CK_BYTE counterValue[8] = {0};
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_END_DATE, &endDate, sizeof(endDate)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_SENSITIVE, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_SIGN, &true, sizeof(true)},
    {CKA_VERIFY, &true, sizeof(true)},
    {CKA_ID, keyId, sizeof(keyId)},
    ```
});

2.53.9.2 HOTP key generation

The HOTP key generation mechanism, denoted **CKM_HOTP_KEY_GEN**, is a key generation mechanism for the HOTP algorithm.

It does not have a parameter.

The mechanism generates HOTP keys with a particular set of attributes as specified in the template for the key.

The mechanism contributes at least the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_OTP_COUNTER**, **CKA_VALUE** and **CKA_VALUE_LEN** attributes to the new key. Other attributes supported by the HOTP key type may be specified in the template for the key, or else are assigned default initial values.

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

2.53.9.3 HOTP OTP generation and validation

**CKM_HOTP** is the mechanism for the retrieval and verification of HOTP OTP values based on the current internal counter, or a provided counter.

The mechanism takes a pointer to a **CK_OTP_PARAMS** structure as a parameter.

As for the **CKM_SECURID** mechanism, when signing or verifying using the **CKM_HOTP** mechanism, **pData** shall be set to **NULL_PTR** and **ulDataLen** shall be set to 0.

For verify operations, the counter value **CK_OTP_COUNTER** must be provided as a **CK_OTP_PARAM** parameter to **C_VerifyInit**. When verifying an OTP value using the **CKM_HOTP** mechanism, **pSignature** shall be set to the OTP value itself, e.g. the value of the **CK_OTP_VALUE** component of a **CK_OTP_PARAM** structure in the case of an earlier call to **C_Sign**.

2.53.10 ActivIdentity ACTI

2.53.10.1 ACTI secret key objects

ACTI secret key objects (object class **CKO_OTP_KEY**, key type **CKK_ACTI**) hold ActivIdentity ACTI secret keys.

For ACTI keys, the **CKA_OTP_COUNTER** value shall be an 8 bytes unsigned integer in big endian (i.e. network byte order) form. The same holds true for the **CK_OTP_COUNTER** value in the **CK_OTP_PARAM** structure.

The **CKA_OTP_COUNTER** value may be set at key generation; however, some tokens may set it to a fixed initial value. Depending on the token’s security policy, this value may not be modified and/or may not be revealed if the object has its **CKA_SENSITIVE** attribute set to **CK_TRUE** or its **CKA_EXTRACTABLE** attribute set to **CK_FALSE**.

The **CKA_OTP_TIME** value may be set at key generation; however, some tokens may set it to a fixed initial value. Depending on the token’s security policy, this value may not be modified and/or may not be revealed if the object has its **CKA_SENSITIVE** attribute set to **CK_TRUE** or its **CKA_EXTRACTABLE** attribute set to **CK_FALSE**.

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

```c
CK_OBJECT_CLASS class = CKO_OTP_KEY;
CK_KEY_TYPE keyType = CKK_ACTI;
CK_UTF8CHAR label[] = “ACTI secret key object”;
```
CK_BYTE keyId[] = {...};
CK_ULONG outputFormat = CK_OTP_FORMAT_DECIMAL;
CK_ULONG outputLength = 6;
CK_DATE endDate = {...};
CK_BYTE counterValue[8] = {0};
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_END_DATE, &endDate, sizeof(endDate)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_SENSITIVE, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_SIGN, &true, sizeof(true)},
    {CKA_VERIFY, &true, sizeof(true)},
    {CKA_ID, keyId, sizeof(keyId)},
    {CKA_OTP_FORMAT, &outputFormat, sizeof(outputFormat)},
    {CKA_OTP_LENGTH, &outputLength, sizeof(outputLength)},
    {CKA_OTP_COUNTER, counterValue, sizeof(counterValue)},
    {CKA_VALUE, value, sizeof(value)}
};

2.53.10.2 ACTI key generation
The ACTI key generation mechanism, denoted CKM_ACTI_KEY_GEN, is a key generation mechanism for the ACTI algorithm.
It does not have a parameter.
The mechanism generates ACTI keys with a particular set of attributes as specified in the template for the key.
The mechanism contributes at least the CKA_CLASS, CKA_KEY_TYPE, CKA_VALUE and CKA_VALUE_LEN attributes to the new key. Other attributes supported by the ACTI key type may be specified in the template for the key, or else are assigned default initial values.
For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of ACTI key sizes, in bytes.

2.53.10.3 ACTI OTP generation and validation
CKM_ACTI is the mechanism for the retrieval and verification of ACTI OTP values.
The mechanism takes a pointer to a CK_OTP_PARAMS structure as a parameter.
When signing or verifying using the CKM_ACTI mechanism, pData shall be set to NULL_PTR and ulDataLen shall be set to 0.
When verifying an OTP value using the CKM_ACTI mechanism, pSignature shall be set to the OTP value itself, e.g. the value of the CK_OTP_VALUE component of a CK_OTP_PARAM structure in the case of an earlier call to C_Sign.
2.54 CT-KIP

2.54.1 Principles of Operation

Figure 4 shows an integration of PKCS #11 into an application that generates cryptographic keys through the use of CT-KIP. The application invokes `C_DeriveKey` to derive a key of a particular type on the token. The key may subsequently be used as a basis to e.g., generate one-time password values. The application communicates with a CT-KIP server that participates in the key derivation and stores a copy of the key in its database. The key is transferred to the server in wrapped form, after a call to `C_WrapKey`. The server authenticates itself to the client and the client verifies the authentication by calls to `C_Verify`.

2.54.2 Mechanisms

The following table shows, for the mechanisms defined in this document, their support by different cryptographic operations. For any particular token, of course, a particular operation may well support only a subset of the mechanisms listed. There is also no guarantee that a token that supports one mechanism for some operation supports any other mechanism for any other operation (or even supports that same mechanism for any other operation).
The remainder of this section will present in detail the mechanisms and the parameters that are supplied to them.

### 2.54.3 Definitions

#### Mechanisms:

- **CKM_KIP_DERIVE**
- **CKM_KIP_WRAP**
- **CKM_KIP_MAC**

### 2.54.4 CT-KIP Mechanism parameters

#### `CK_KIP_PARAMS; CK_KIP_PARAMS_PTR`

A structure that provides the parameters to all the CT-KIP related mechanisms: The **CKM_KIP_DERIVE** key derivation mechanism, the **CKM_KIP_WRAP** key wrap and key unwrap mechanism, and the **CKM_KIP_MAC** signature mechanism. The structure is defined as follows:

```c
typedef struct CK_KIP_PARAMS {
    CK_MECHANISM_PTR  pMechanism;
    CK_OBJECT_HANDLE  hKey;
    CK_BYTE_PTR       pSeed;
    CK_ULONG          ulSeedLen;
} CK_KIP_PARAMS;
```

The fields of the structure have the following meanings:

- **pMechanism**: pointer to the underlying cryptographic mechanism (e.g. AES, SHA-256), see further 0, Appendix D
- **hKey**: handle to a key that will contribute to the entropy of the derived key (CKM_KIP_DERIVE) or will be used in the MAC operation (CKM_KIP_MAC)
- **pSeed**: pointer to an input seed
- **ulSeedLen**: length in bytes of the input seed

**CK_KIP_PARAMS_PTR** is a pointer to a `CK_KIP_PARAMS` structure.

### 2.54.5 CT-KIP key derivation

The CT-KIP key derivation mechanism, denoted **CKM_KIP_DERIVE**, is a key derivation mechanism that is capable of generating secret keys of potentially any type, subject to token limitations.
It takes a parameter of type \texttt{CK_KIP_PARAMS} which allows for the passing of the desired underlying cryptographic mechanism as well as some other data. In particular, when the \texttt{hKey} parameter is a handle to an existing key, that key will be used in the key derivation in addition to the \texttt{hBaseKey} of \texttt{C_DeriveKey}.

The \texttt{pSeed} parameter may be used to seed the key derivation operation.

The mechanism derives a secret key with a particular set of attributes as specified in the attributes of the template for the key.

The mechanism contributes the \texttt{CKA_CLASS} and \texttt{CKA_VALUE} attributes to the new key. Other attributes supported by the key type may be specified in the template for the key, or else will be assigned default initial values. Since the mechanism is generic, the \texttt{CKA_KEY_TYPE} attribute should be set in the template, if the key is to be used with a particular mechanism.

### 2.54.6 CT-KIP key wrap and key unwrap

The CT-KIP key wrap and unwrap mechanism, denoted \texttt{CKM_KIP_WRAP}, is a key wrap mechanism that is capable of wrapping and unwrapping generic secret keys.

It takes a parameter of type \texttt{CK_KIP_PARAMS}, which allows for the passing of the desired underlying cryptographic mechanism as well as some other data. It does not make use of the \texttt{hKey} parameter of \texttt{CK_KIP_PARAMS}.

### 2.54.7 CT-KIP signature generation

The CT-KIP signature (MAC) mechanism, denoted \texttt{CKM_KIP_MAC}, is a mechanism used to produce a message authentication code of arbitrary length. The keys it uses are secret keys.

It takes a parameter of type \texttt{CK_KIP_PARAMS}, which allows for the passing of the desired underlying cryptographic mechanism as well as some other data. The mechanism does not make use of the \texttt{pSeed} and the \texttt{ulSeedLen} parameters of \texttt{CT_KIP_PARAMS}.

This mechanism produces a MAC of the length specified by \texttt{pulSignatureLen} parameter in calls to \texttt{C_Sign}.

If a call to \texttt{C_Sign} with this mechanism fails, then no output will be generated.

### 2.55 GOST 28147-89

GOST 28147-89 is a block cipher with 64-bit block size and 256-bit keys.

Table 201, GOST 28147-89 Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/ Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_GOST28147_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_GOST28147_ECB</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_GOST28147</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_GOST28147_MAC</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_GOST28147_KEY_WRAP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
2.55.1 Definitions

This section defines the key type “CKK_GOST28147” for type CK_KEY_TYPE as used in the
CKA_KEY_TYPE attribute of key objects and domain parameter objects.

Mechanisms:

- CKM_GOST28147_KEY_GEN
- CKM_GOST28147_ECB
- CKM_GOST28147
- CKM_GOST28147_MAC
- CKM_GOST28147_KEY_WRAP

2.55.2 GOST 28147-89 secret key objects

GOST 28147-89 secret key objects (object class CKO_SECRET_KEY, key type CKK_GOST28147) hold
GOST 28147-89 keys. The following table defines the GOST 28147-89 secret key object attributes, in
addition to the common attributes defined for this object class:

Table 202, GOST 28147-89 Secret Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE&lt;sup&gt;1,4,6,7&lt;/sup&gt;</td>
<td>Byte array</td>
<td>32 bytes in little endian order</td>
</tr>
<tr>
<td>CKA_GOST28147_PARAMS&lt;sup&gt;1,3,5&lt;/sup&gt;</td>
<td>Byte array</td>
<td>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</td>
</tr>
</tbody>
</table>

Refer to [PKCS11-Base] Table 11 for footnotes

The following is a sample template for creating a GOST 28147-89 secret key object:

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_GOST28147;
CK_UTF8CHAR label[] = "A GOST 28147-89 secret key object";
CK_BYTE value[32] = {..};
CK_BYTE params_oid[] = {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x1f, 0x00};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
  {CKA_CLASS, &class, sizeof(class)},
  {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
  {CKA_TOKEN, &true, sizeof(true)},
  {CKA_LABEL, label, sizeof(label)-1},
  {CKA_ENCRYPT, &true, sizeof(true)},
  {CKA_GOST28147_PARAMS, params_oid, sizeof(params_oid)},
  {CKA_VALUE, value, sizeof(value)}
};
```
2.55.3 GOST 28147-89 domain parameter objects

GOST 28147-89 domain parameter objects (object class `CKO_DOMAIN_PARAMETERS`, key type `CKK_GOST28147`) hold GOST 28147-89 domain parameters.

The following table defines the GOST 28147-89 domain parameter object attributes, in addition to the common attributes defined for this object class:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE(^1)</td>
<td>Byte array</td>
<td>DER-encoding of the domain parameters as it was introduced in [4] section 8.1 (type Gost28147-89-ParamSetParameters)</td>
</tr>
<tr>
<td>CKA_OBJECT_ID(^1)</td>
<td>Byte array</td>
<td>DER-encoding of the object identifier indicating the domain parameters</td>
</tr>
</tbody>
</table>

Refer to [PKCS11-Base] Table 11 for footnotes

For any particular token, there is no guarantee that a token supports domain parameters loading up and/or fetching out. Furthermore, applications, that make direct use of domain parameters objects, should take in account that CKA_VALUE attribute may be inaccessible.

The following is a sample template for creating a GOST 28147-89 domain parameter object:

```c
CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
CK_KEY_TYPE keyType = CKK_GOST28147;
CK_UTF8CHAR label[] = "A GOST 28147-89 cryptographic parameters object";
CK_BYTE oid[] = {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1f, 0x00};
CK_BYTE value[] = {
    0x30,0x62,0x04,0x40,0x4c,0xde,0x38,0x9c,0x29,0x89,0xef,0xb6,
    0xc5,0xe5,0xc2,0x9b,0x02,0x98,0x75,0x61,0x3b,
    0x11,0x3f,0x89,0x60,0x03,0x97,0x0c,0x79,0xa1,0xd5,0x5d,
    0xe2,0xad,0x43,0x37,0x5d,0xb3,0x8e,0xb4,0x2c,0x77,0xe7,
    0xcd,0x46,0xca,0xfa,0x9d,0x6a,0x20,0x1f,0x70,0xf4,0x1e,0xa4,
    0xab,0x03,0xf2,0x21,0x65,0xb8,0x44,0xd8,0x02,0x01,0x00,0x02,
    0x01,0x40,0x30,0x0b,0x06,0x07,0x2a,0x85,0x03,0x02,0x02,0x0e,
    0x00,0x05,0x00
};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_OBJECT_ID, oid, sizeof(oid)},
    {CKA_VALUE, value, sizeof(value)}
};
```

2.55.4 GOST 28147-89 key generation

The GOST 28147-89 key generation mechanism, denoted `CKM_GOST28147_KEY_GEN`, is a key generation mechanism for GOST 28147-89.

It does not have a parameter.
The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the GOST 28147-89 key type may be specified for objects of object class **CKO_SECRET_KEY**.

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the **CK_MECHANISM_INFO** are not used.

### 2.55.5 GOST 28147-89-ECB

GOST 28147-89-ECB, denoted **CKM_GOST28147_ECB**, is a mechanism for single and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on GOST 28147-89 and electronic codebook mode.

It does not have a parameter.

This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to wrap/unwrap every secret key that it supports.

For wrapping (**C_WrapKey**), the mechanism encrypts the value of the **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size so that the resulting length is a multiple of the block size.

For unwrapping (**C_UnwrapKey**), the mechanism decrypts the wrapped key, and truncates the result according to the **CKA_KEY_TYPE** attribute of the template and, if it has one, and the key type supports it, the **CKA_VALUE_LEN** attribute of the template. The mechanism contributes the result as the **CKA_VALUE** attribute of the new key.

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>C_Encrypt</strong></td>
<td>CKK_GOST28147</td>
<td>Multiple of block size</td>
<td>Same as input length</td>
</tr>
<tr>
<td><strong>C_Decrypt</strong></td>
<td>CKK_GOST28147</td>
<td>Multiple of block size</td>
<td>Same as input length</td>
</tr>
<tr>
<td><strong>C_WrapKey</strong></td>
<td>CKK_GOST28147</td>
<td>Any</td>
<td>Input length rounded up to multiple of block size</td>
</tr>
<tr>
<td><strong>C_UnwrapKey</strong></td>
<td>CKK_GOST28147</td>
<td>Multiple of block size</td>
<td>Determined by type of key being unwrapped</td>
</tr>
</tbody>
</table>

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the **CK_MECHANISM_INFO** structure are not used.

### 2.55.6 GOST 28147-89 encryption mode except ECB

GOST 28147-89 encryption mode except ECB, denoted **CKM_GOST28147** is a mechanism for single and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on GOST 28147-89 and CFB, counter mode, and additional CBC mode defined in [RFC 4357] section 2. Encryption's parameters are specified in object identifier of attribute **CKA_GOST28147_PARAMS**.

It has a parameter, which is an 8-byte initialization vector. This parameter may be omitted then a zero initialization vector is used.

This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to wrap/unwrap every secret key that it supports.

For wrapping (**C_WrapKey**), the mechanism encrypts the value of the **CKA_VALUE** attribute of the key that is wrapped.
For unwrapping (\textcode{C\_UnwrapKey}), the mechanism decrypts the wrapped key, and contributes the result as the \textcode{CKA\_VALUE} attribute of the new key.

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

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
\textbf{Function} & \textbf{Key type} & \textbf{Input length} & \textbf{Output length} \\
\hline
\textcode{C\_Encrypt} & \textcode{CKK\_GOST28147} & \text{Any} & \text{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} \\
\textcode{C\_Decrpt} & \textcode{CKK\_GOST28147} & \text{Any} & \\
\textcode{C\_WrapKey} & \textcode{CKK\_GOST28147} & \text{Any} & \\
\textcode{C\_UnwrapKey} & \textcode{CKK\_GOST28147} & \text{Any} & \\
\hline
\end{tabular}
\end{table}

For this mechanism, the \textcode{ul\_MinKeySize} and \textcode{ul\_MaxKeySize} fields of the \textcode{CK\_MECHANISM\_INFO} structure are not used.

\section*{2.55.7 GOST 28147-89-MAC}

GOST 28147-89-MAC, denoted \textcode{CKM\_GOST28147\_MAC}, is a mechanism for data integrity and authentication based on GOST 28147-89 and key meshing algorithms [RFC 4357] section 2.3.

MACing parameters are specified in object identifier of attribute \textcode{CKA\_GOST28147\_PARAMS}.

The output bytes from this mechanism are taken from the start of the final GOST 28147-89 cipher block produced in the MACing process.

It has a parameter, which is an 8-byte MAC initialization vector. This parameter may be omitted then a zero initialization vector is used.

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

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
\textbf{Function} & \textbf{Key type} & \textbf{Data length} & \textbf{Signature length} \\
\hline
\textcode{C\_Sign} & \textcode{CKK\_GOST28147} & \text{Any} & \text{4 bytes} \\
\textcode{C\_Verify} & \textcode{CKK\_GOST28147} & \text{Any} & \text{4 bytes} \\
\hline
\end{tabular}
\end{table}

For this mechanism, the \textcode{ul\_MinKeySize} and \textcode{ul\_MaxKeySize} fields of the \textcode{CK\_MECHANISM\_INFO} structure are not used.

\section*{2.55.8 GOST 28147-89 keys wrapping/unwrapping with GOST 28147-89}

GOST 28147-89 keys as a KEK (key encryption keys) for encryption GOST 28147-89 keys, denoted by \textcode{CKM\_GOST28147\_KEY\_WRAP}, is a mechanism for key wrapping; and key unwrapping, based on GOST 28147-89. Its purpose is to encrypt and decrypt keys have been generated by key generation mechanism for GOST 28147-89.

For wrapping (\textcode{C\_WrapKey}), the mechanism first computes MAC from the value of the \textcode{CKA\_VALUE} attribute of the key that is wrapped and then encrypts in ECB mode the value of the \textcode{CKA\_VALUE} attribute of the key that is wrapped. The result is 32 bytes of the key that is wrapped and 4 bytes of MAC.

For unwrapping (\textcode{C\_UnwrapKey}), the mechanism first decrypts in ECB mode the 32 bytes of the key that was wrapped and then computes MAC from the unwrapped key. Then compared together 4 bytes MAC.
has computed and 4 bytes MAC of the input. If these two MACs do not match the wrapped key is disallowed. The mechanism contributes the result as the **CKA_VALUE** attribute of the unwrapped key.

It has a parameter, which is an 8-byte MAC initialization vector. This parameter may be omitted then a zero initialization vector is used.

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

*Table 207, GOST 28147-89 keys as KEK: Key and Data Length*

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_WrapKey</td>
<td>CKK_GOST28147</td>
<td>32 bytes</td>
<td>36 bytes</td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>CKK_GOST28147</td>
<td>32 bytes</td>
<td>36 bytes</td>
</tr>
</tbody>
</table>

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure are not used.

**2.56 GOST R 34.11-94**

GOST R 34.11-94 is a mechanism for message digesting, following the hash algorithm with 256-bit message digest defined in [GOST R 34.11-94].

*Table 208, GOST R 34.11-94 Mechanisms vs. Functions*

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Functions</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/ Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_GOSTR3411</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_GOSTR3411_HMAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

**2.56.1 Definitions**

This section defines the key type “CKK_GOSTR3411” for type **CK_KEY_TYPE** as used in the **CKA_KEY_TYPE** attribute of domain parameter objects.

Mechanisms:

- CKM_GOSTR3411
- CKM_GOSTR3411_HMAC

**2.56.2 GOST R 34.11-94 domain parameter objects**

GOST R 34.11-94 domain parameter objects (object class **CKO_DOMAIN_PARAMETERS**, key type **CKK_GOSTR3411**) hold GOST R 34.11-94 domain parameters.

The following table defines the GOST R 34.11-94 domain parameter object attributes, in addition to the common attributes defined for this object class:
Table 209, GOST R 34.11-94 Domain Parameter Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE(^1)</td>
<td>Byte array</td>
<td>DER-encoding of the domain parameters as it was introduced in [4] section 8.2 (type GostR3411-94-ParamSetParameters)</td>
</tr>
<tr>
<td>CKA_OBJECT_ID(^1)</td>
<td>Byte array</td>
<td>DER-encoding of the object identifier indicating the domain parameters</td>
</tr>
</tbody>
</table>

Refer to [PKCS11-Base] Table 11 for footnotes

For any particular token, there is no guarantee that a token supports domain parameters loading up and/or fetching out. Furthermore, applications, that make direct use of domain parameters objects, should take in account that CKA_VALUE attribute may be inaccessible.

The following is a sample template for creating a GOST R 34.11-94 domain parameter object:

```c
CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
CK_KEY_TYPE keyType = CKK_GOSTR3411;
CK_UTF8CHAR label[] = "A GOST R34.11-94 cryptographic parameters object";
CK_BYTE oid[] = {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1e, 0x00};
CK_BYTE value[] = {
  0x30, 0x64, 0x04, 0x40, 0x4e, 0x57, 0x64, 0xd1, 0xab, 0x8d, 0xcb, 0xbf,
  0x94, 0x1a, 0x77, 0x4d, 0x2c, 0xd1, 0x10, 0x10, 0xd6, 0xa0, 0x57, 0x35,
  0x8d, 0x38, 0xf2, 0xf7, 0x0f, 0x49, 0x1d, 0x5a, 0xea, 0x2f, 0x8d, 0x94,
  0x62, 0xe6, 0x43, 0x09, 0xb3, 0xf4, 0xa6, 0x2a, 0x18, 0xc6, 0x98, 0xe3,
  0x55, 0x95, 0xbf, 0x28, 0x39, 0xb3, 0xe2, 0xc0, 0x04, 0x20, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
  {CKA_CLASS, &class, sizeof(class)},
  {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
  {CKA_TOKEN, &true, sizeof(true)},
  {CKA_LABEL, label, sizeof(label)-1},
  {CKA_OBJECT_ID, oid, sizeof(oid)},
  {CKA_VALUE, value, sizeof(value)}
};
```

2.56.3 GOST R 34.11-94 digest

GOST R 34.11-94 digest, denoted **CKM_GOSTR3411**, is a mechanism for message digesting based on GOST R 34.11-94 hash algorithm [GOST R 34.11-94].

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.

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.
For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the *CK_MECHANISM_INFO* structure are not used.

### 2.56.4 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>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>CKK GENERIC_SECRET or CKK_GOST28147</td>
<td>Any</td>
<td>32 bytes</td>
</tr>
<tr>
<td>C_Verify</td>
<td>CKK GENERIC_SECRET or CKK_GOST28147</td>
<td>Any</td>
<td>32 bytes</td>
</tr>
</tbody>
</table>

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the *CK_MECHANISM_INFO* structure are not used.

### 2.57 GOST R 34.10-2001

GOST R 34.10-2001 is a mechanism for single- and multiple-part signatures and verification, following the digital signature algorithm defined in [GOST R 34.10-2001].

Table 212, GOST R34.10-2001 Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Encrypt &amp; Decrypt</td>
</tr>
<tr>
<td>CKM_GOSTR3410_KEY_PAIR_GEN</td>
<td></td>
</tr>
<tr>
<td>CKM_GOSTR3410</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_GOSTR3410_WITH_GOST3411</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_GOSTR3410_KEY_WRAP</td>
<td></td>
</tr>
<tr>
<td>CKM_GOSTR3410_DERIVE</td>
<td></td>
</tr>
</tbody>
</table>

1 Single-part operations only
2.57.1 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:

- CKM_GOSTR3410_KEY_PAIR_GEN
- CKM_GOSTR3410
- CKM_GOSTR3410_WITH_GOSTR3411
- CKM_GOSTR3410
- CKM_GOSTR3410_KEY_WRAP
- CKM_GOSTR3410_DERIVE

2.57.2 GOST R 34.10-2001 public key objects

GOST R 34.10-2001 public key objects (object class CKO_PUBLIC_KEY, key type CKK_GOSTR3410) hold GOST R 34.10-2001 public keys.

The following table defines the GOST R 34.10-2001 public key object attributes, in addition to the common attributes defined for this object class:

Table 213, GOST R 34.10-2001 Public Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE</td>
<td>Byte array</td>
<td>64 bytes for public key; 32 bytes for each coordinates X and Y of elliptic curve point P(X, Y) in little endian order</td>
</tr>
<tr>
<td>CKA_GOSTR3410_PARAMS</td>
<td>Byte array</td>
<td>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</td>
</tr>
<tr>
<td>CKA_GOSTR3411_PARAMS</td>
<td>Byte array</td>
<td>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</td>
</tr>
<tr>
<td>CKA_GOST28147_PARAMS</td>
<td>Byte array</td>
<td>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</td>
</tr>
</tbody>
</table>

Refer to [PKCS11-Base] Table 11 for footnotes

The following is a sample template for creating an GOST R 34.10-2001 public key object:
CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_GOSTR3410;
CK_UTF8CHAR label[] = "A GOST R34.10-2001 public key object";
CK_BYTE gostR3410params_oid[] = 
    {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x23, 0x00};
CK_BYTE gostR3411params_oid[] =
    {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1e, 0x00};
CK_BYTE gost28147params_oid[] =
    {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1f, 0x00};
CK_BYTE value[64] = {...};
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_GOSTR3410_PARAMS, gostR3410params_oid, sizeof(gostR3410params_oid)},
    {CKA_GOSTR3411_PARAMS, gostR3411params_oid, sizeof(gostR3411params_oid)},
    {CKA_GOST28147_PARAMS, gost28147params_oid, sizeof(gost28147params_oid)},
    {CKA_VALUE, value, sizeof(value)}
};

2.57.3 GOST R 34.10-2001 private key objects

GOST R 34.10-2001 private key objects (object class CKO_PRIVATE_KEY, key type
CKK_GOSTR3410) hold GOST R 34.10-2001 private keys.
The following table defines the GOST R 34.10-2001 private key object attributes, in addition to the
common attributes defined for this object class:
Table 214, GOST R 34.10-2001 Private Key Object Attributes
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE¹,4,6,7</td>
<td>Byte array</td>
<td>32 bytes for private key in little endian order</td>
</tr>
<tr>
<td>CKA_GOSTR3410_PARAMS¹,4,6</td>
<td>Byte array</td>
<td>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.</td>
</tr>
<tr>
<td>CKA_GOSTR3411_PARAMS¹,4,6,8</td>
<td>Byte array</td>
<td>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.</td>
</tr>
<tr>
<td>CKA_GOST28147_PARAMS4²,6,8</td>
<td>Byte array</td>
<td>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.</td>
</tr>
</tbody>
</table>

Note that when generating an GOST R 34.10-2001 private key, the GOST R 34.10-2001 domain parameters are not specified in the key's template. This is because GOST R 34.10-2001 private keys are only generated as part of an GOST R 34.10-2001 key pair, and the GOST R 34.10-2001 domain parameters for the pair are specified in the template for the GOST R 34.10-2001 public key.

The following is a sample template for creating an GOST R 34.10-2001 private key object:

```c
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_GOSTR3410;
CK_UTF8CHAR label[] = "A GOST R34.10-2001 private key object";
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE gostR3410params_oid[] =
    {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x23, 0x00};
CK_BYTE gostR3411params_oid[] =
    {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1e, 0x00};
CK_BYTE gost28147params_oid[] =
    {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1f, 0x00};
CK_BYTE value[32] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
```

---

⁷⁶²⁴ Refer to [PKCS11-Base] Table 11 for footnotes

⁷⁶²⁵ Note that when generating an GOST R 34.10-2001 private key, the GOST R 34.10-2001 domain 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.

⁷⁶²⁶ The following is a sample template for creating an GOST R 34.10-2001 private key object:
{CKA_CLASS, &class, sizeof(class)},
{CKA_KEY_TYPE, &keyType, sizeof(keyType)},
{CKA_TOKEN, &true, sizeof(true)},
{CKA_LABEL, label, sizeof(label)-1},
{CKA_SUBJECT, subject, sizeof(subject)},
{CKA_ID, id, sizeof(id)},
{CKA_SENSITIVE, &true, sizeof(true)},
{CKA_SIGN, &true, sizeof(true)},
{CKA_GOSTR3410_PARAMS, gostR3410params_oid, sizeof(gostR3410params_oid)},
{CKA_GOSTR3411_PARAMS, gostR3411params_oid, sizeof(gostR3411params_oid)},
{CKA_GOST28147_PARAMS, gost28147params_oid, sizeof(gost28147params_oid)},
{CKA_VALUE, value, sizeof(value)}

2.57.4 GOST R 34.10-2001 domain parameter objects

GOST R 34.10-2001 domain parameter objects (object class CKO_DOMAIN_PARAMETERS, key type CKK_GOSTR3410) hold GOST R 34.10-2001 domain parameters. The following table defines the GOST R 34.10-2001 domain parameter object attributes, in addition to the common attributes defined for this object class:

Table 215, GOST R 34.10-2001 Domain Parameter Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE(^1)</td>
<td>Byte array</td>
<td>DER-encoding of the domain parameters as it was introduced in [4] section 8.4 (type GostR3410-2001-ParamSetParameters)</td>
</tr>
<tr>
<td>CKA_OBJECT_ID(^1)</td>
<td>Byte array</td>
<td>DER-encoding of the object identifier indicating the domain parameters</td>
</tr>
</tbody>
</table>

Refer to [PKCS11-Base] Table 11 for footnotes

For any particular token, there is no guarantee that a token supports domain parameters loading up and/or fetching out. Furthermore, applications, that make direct use of domain parameters objects, should take in account that CKA_VALUE attribute may be inaccessible.

The following is a sample template for creating a GOST R 34.10-2001 domain parameter object:

```c
CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
CK_KEY_TYPE keyType = CKK_GOSTR3410;
CK_UTF8CHAR label[] = “A GOST R34.10-2001 cryptographic parameters object”;
CK_BYTE oid[] = {
0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x23, 0x00,
0x30, 0x81, 0x90, 0x02, 0x01, 0x07, 0x02, 0x02, 0x20, 0x5f, 0xbf, 0xf4, 0x98,
0xaa, 0x93, 0x8c, 0xe7, 0x39, 0xb8, 0xe0, 0x22, 0xfb, 0xaf, 0xef, 0x40,
0x56, 0x3f, 0x6e, 0x6a, 0x34, 0x72, 0xfc, 0x2a, 0x4c, 0xe9,
0xda, 0xe2, 0x3b, 0x7e, 0x02, 0x21, 0x00, 0x80, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
... };
```
2.57.5 GOST R 34.10-2001 mechanism parameters

♦ CK_GOSTR3410_KEY_WRAP_PARAMS

CK_GOSTR3410_KEY_WRAP_PARAMS is a structure that provides the parameters to the CKM_GOSTR3410_KEY_WRAP mechanism. It is defined as follows:

```c
typedef struct CK_GOSTR3410_KEY_WRAP_PARAMS {
    CK_BYTE_PTR      pWrapOID;
    CK_ULONG         ulWrapOIDLen;
    CK_BYTE_PTR      pUKM;
    CK_ULONG         ulUKMLen;
    CK_OBJECT_HANDLE hKey;
} CK_GOSTR3410_KEY_WRAP_PARAMS;
```

The fields of the structure have the following meanings:

- `pWrapOID` 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.

- `ulWrapOIDLen` length of data with DER-encoding of the object identifier indicating the data object type of GOST 28147-89.

- `pUKM` 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

ulUKMLen length of UKM data. If pUKM-pointer is different from NULL_PTR then equal to 8

hKey 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

♦ CK_GOSTR3410_DERIVE_PARAMS

CK_GOSTR3410_DERIVE_PARAMS is a structure that provides the parameters to the CKM_GOSTR3410_DERIVE mechanism. It is defined as follows:

```c
typedef struct CK_GOSTR3410_DERIVE_PARAMS {
    CK_EC_KDF_TYPE kdf;
    CK_BYTE_PTR    pPublicData;
    CK_ULONG       ulPublicDataLen;
    CK_BYTE_PTR    pUKM;
    CK_ULONG       ulUKMLen;
} CK_GOSTR3410_DERIVE_PARAMS;
```

The fields of the structure have the following meanings:

- **kdf** additional key diversification algorithm identifier. Possible values are CKD_NULL and CKD_CPDIVERSIFY_KDF. In case of CKD_NULL, result of the key derivation function described in [RFC 4357], section 5.2 is used directly; In case of CKD_CPDIVERSIFY_KDF, the resulting key value is additionally processed with algorithm from [RFC 4357], section 6.5.

- **pPublicData** pointer to data with public key of a receiver

- **ulPublicDataLen** length of data with public key of a receiver (must be 64)

- **pUKM** pointer to a UKM data

- **ulUKMLen** length of UKM data in bytes (must be 8)

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 them is 32 bytes long and represented in little endian order.

2.57.6 GOST R 34.10-2001 key pair generation

The GOST R 34.10-2001 key pair generation mechanism, denoted CKM_GOSTR3410_KEY_PAIR_GEN, is a key pair generation mechanism for GOST R 34.10-2001. This mechanism does not have a parameter.

The mechanism generates GOST R 34.10-2001 public/private key pairs with particular GOST R 34.10-2001 domain parameters, as specified in the CKA_GOSTR3410_PARAMS.
2.57.7 GOST R 34.10-2001 without hashing

The GOST R 34.10-2001 without hashing mechanism, denoted CKM_GOSTR3410, is a mechanism for single-part signatures and verification for GOST R 34.10-2001. This mechanism corresponds only to the part of GOST R 34.10-2001 that processes the 32-bytes hash value; it does not compute the hash value.

This mechanism does not have a parameter.

For the purposes of these mechanisms, a GOST R 34.10-2001 signature is an octet string of 64 bytes long. The signature octets correspond to the concatenation of the GOST R 34.10-2001 values \( s \) and \( r' \), both represented as a 32 bytes octet string in big endian order with the most significant byte first [RFC 4490] section 3.2, and [RFC 4491] section 2.2.2.

The input for the mechanism is an octet string of 32 bytes long with digest has computed by means of GOST R 34.11-94 hash algorithm in the context of signed or should be signed message.

Table 216, GOST R 34.10-2001 without hashing: Key and Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign(^1)</td>
<td>CKK_GOSTR3410</td>
<td>32 bytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td>C_Verify(^1)</td>
<td>CKK_GOSTR3410</td>
<td>32 bytes</td>
<td>64 bytes</td>
</tr>
</tbody>
</table>

\(^1\) Single-part operations only.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure are not used.

2.57.8 GOST R 34.10-2001 with GOST R 34.11-94

The GOST R 34.10-2001 with GOST R 34.11-94, denoted CKM_GOSTR3410_WITH_GOSTR3411, is a mechanism for signatures and verification for GOST R 34.10-2001. This mechanism computes the entire GOST R 34.10-2001 specification, including the hashing with GOST R 34.11-94 hash algorithm.

As a parameter this mechanism utilizes a DER-encoding of the object identifier indicating GOST R 34.11-94 data object type. A mechanism parameter may be missed then parameters are specified in object identifier of attribute CKA_GOSTR3411_PARAMS must be used.

For the purposes of these mechanisms, a GOST R 34.10-2001 signature is an octet string of 64 bytes long. The signature octets correspond to the concatenation of the GOST R 34.10-2001 values \( s \) and \( r' \), both represented as a 32 bytes octet string in big endian order with the most significant byte first [RFC 4490] section 3.2, and [RFC 4491] section 2.2.2.

The input for the mechanism is signed or should be signed message of any length. Single- and multiple-part signature operations are available.

Table 217, GOST R 34.10-2001 with GOST R 34.11-94: Key and Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>CKK_GOSTR3410</td>
<td>Any</td>
<td>64 bytes</td>
</tr>
<tr>
<td>C_Verify</td>
<td>CKK_GOSTR3410</td>
<td>Any</td>
<td>64 bytes</td>
</tr>
</tbody>
</table>

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure are not used.
2.57.9 GOST 28147-89 keys wrapping/unwrapping with GOST R 34.10-2001

GOST R 34.10-2001 keys as a KEK (key encryption keys) for encryption GOST 28147 keys, denoted by \texttt{CKM\_GOSTR3410\_KEY\_WRAP}, is a mechanism for key wrapping; and key unwrapping, based on GOST R 34.10-2001. Its purpose is to encrypt and decrypt keys have been generated by key generation mechanism for GOST 28147-89. An encryption algorithm from [RFC 4490] (section 5.2) must be used.

Encrypted key is a DER-encoded structure of ASN.1 \texttt{GostR3410-KeyTransport} type [RFC 4490] section 4.2.

It has a parameter, a \texttt{CK\_GOSTR3410\_KEY\_WRAP\_PARAMS} structure defined in section 2.57.5.

For unwrapping (\texttt{C\_UnwrapKey}), the mechanism decrypts the wrapped key, and contributes the result as the \texttt{CKA\_VALUE} attribute of the new key.

For this mechanism, the \texttt{ulMinKeySize} and \texttt{ulMaxKeySize} fields of the \texttt{CK\_MECHANISM\_INFO} structure are not used.

2.57.10 Common key derivation with assistance of GOST R 34.10-2001 keys

Common key derivation, denoted \texttt{CKM\_GOSTR3410\_DERIVE}, is a mechanism for key derivation with assistance of GOST R 34.10-2001 private and public keys. The key of the mechanism must be of object class \texttt{CKO\_DOMAIN\_PARAMETERS} and key type \texttt{CKK\_GOSTR3410}. An algorithm for key derivation from [RFC 4357] (section 5.2) must be used.

The mechanism contributes the result as the \texttt{CKA\_VALUE} attribute of the new private key. All other attributes must be specified in a template for creating private key object.

2.58 ChaCha20

ChaCha20 is a secret-key stream cipher described in [CHACHA].

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; 1 VR</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_CHACHA20_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_CHACHA20</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

2.58.1 Definitions

This section defines the key type “CKK\_CHACHA20” for type \texttt{CK\_KEY\_TYPE} as used in the \texttt{CKA\_KEY\_TYPE} attribute of key objects.

Mechanisms:

- CKM\_CHACHA20\_KEY\_GEN
- CKM\_CHACHA20
### 2.58.2 ChaCha20 secret key objects

ChaCha20 secret key objects (object class CKO_SECRET_KEY, key type CKK_CHACHA) hold ChaCha20 keys. The following table defines the ChaCha20 secret key object attributes, in addition to the common attributes defined for this object class:

**Table 219, ChaCha20 Secret Key Object**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE⁰,⁴,⁶,⁷</td>
<td>Byte array</td>
<td>Key length is fixed at 256 bits. Bit length restricted to a byte array.</td>
</tr>
<tr>
<td>CKA_VALUE_LEN²,³</td>
<td>CK_ULONG</td>
<td>Length in bytes of key value</td>
</tr>
</tbody>
</table>

The following is a sample template for creating a ChaCha20 secret key object:

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_CHACHA20;
CK_UTF8CHAR label[] = "A ChaCha20 secret key object";
CK_BYTE value[32] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label) - 1},
    {CKA_ENCRYPT, &true, sizeof(true)},
    {CKA_VALUE, value, sizeof(value)}
};
```

**CKA_CHECK_VALUE**: The value of this attribute is derived from the key object by taking the first three bytes of the SHA-1 hash of the ChaCha20 secret key object's CKA_VALUE attribute.

### 2.58.3 ChaCha20 mechanism parameters

**CK_CHACHA20_PARAMS; CK_CHACHA20_PARAMS_PTR**

`CK_CHACHA20_PARAMS` provides the parameters to the `CKM_CHACHA20` mechanism. It is defined as follows:

```c
typedef struct CK_CHACHA20_PARAMS {
    CK_BYTE_PTR pBlockCounter;
    CK_ULONG blockCounterBits;
    CK_BYTE_PTR pNonce;
    CK_ULONG ulNonceBits;
} CK_CHACHA20_PARAMS;
```

The fields of the structure have the following meanings:

- `pBlockCounter` — pointer to block counter
- `ulblockCounterBits` — length of block counter in bits (can be either 32 or 64)
- `pNonce` — nonce (This should be never re-used with the same key.)
- `ulNonceBits` — length of nonce in bits (is 64 for original, 96 for IETF and 192 for xchacha20 variant)
The block counter is used to address 512 bit blocks in the stream. In certain settings (e.g. disk encryption) it is necessary to address these blocks in random order, thus this counter is exposed here.

### 2.58.4 ChaCha20 key generation

The ChaCha20 key generation mechanism, denoted **CKM_CHACHA20_KEY_GEN**, is a key generation mechanism for ChaCha20.

It does not have a parameter.

The mechanism generates ChaCha20 keys of 256 bits.

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

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of key sizes in bytes. As a practical matter, the key size for ChaCha20 is fixed at 256 bits.

### 2.58.5 ChaCha20 mechanism

ChaCha20, denoted **CKM_CHACHA20**, is a mechanism for single and multiple-part encryption and decryption based on the ChaCha20 stream cipher. It comes in 3 variants, which only differ in the size and handling of their nonces, affecting the safety of using random nonces and the maximum size that can be encrypted safely.

ChaCha20 has a parameter, **CK_CHACHA20_PARAMS**, which indicates the nonce and initial block counter value.

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

### Table 220, ChaCha20: Key and Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>ChaCha20</td>
<td>Any / only up to 256 GB in case of IETF variant</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>ChaCha20</td>
<td>Any / only up to 256 GB in case of IETF variant</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
</tbody>
</table>

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of ChaCha20 key sizes, in bits.

### Table 221, ChaCha20: Nonce and block counter lengths

<table>
<thead>
<tr>
<th>Variant</th>
<th>Nonce</th>
<th>Block counter</th>
<th>Maximum message</th>
<th>Nonce generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>64 bit</td>
<td>64 bit</td>
<td>Virtually unlimited</td>
<td>1(^{st}) msg: nonce(_0)=random \nn(^{th}) msg: nonce(_n-1)=++</td>
</tr>
<tr>
<td>Mechanism</td>
<td>Encrypt &amp; Decrypt</td>
<td>Sign &amp; Verify</td>
<td>SR &amp; 1 VR</td>
<td>Digest</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------</td>
<td>---------------</td>
<td>-----------</td>
<td>--------</td>
</tr>
<tr>
<td>CKM_SALSA20_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SALSA20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 2.59 Definitions

This section defines the key type “CKK_SALSA20” and “CKK_SALSA20” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

- CKM_SALSA20.Key.Gen
- CKM_SALSA20

### 2.59.2 Salsa20 Secret Key Objects

Salsa20 secret key objects (object class CKO_SECRET_KEY, key type CKK_SALSA) hold Salsa20 keys. The following table defines the Salsa20 secret key object attributes, in addition to the common attributes defined for this object class:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE</td>
<td>Byte array</td>
<td>Key length is fixed at 256 bits. Bit length restricted to a byte array.</td>
</tr>
<tr>
<td>CKA_VALUE_LEN</td>
<td>CK_ULONG</td>
<td>Length in bytes of key value</td>
</tr>
</tbody>
</table>

The following is a sample template for creating a Salsa20 secret key object:

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
```
CK_KEY_TYPE keyType = CKK_SALSA20;
CK_UTF8CHAR label[] = "A Salsa20 secret key object";
CK_BYTE value[32] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_ENCRYPT, &true, sizeof(true)},
    {CKA_VALUE, value, sizeof(value)}
};

CKA_CHECK_VALUE: The value of this attribute is derived from the key object by taking the first three bytes of the SHA-1 hash of the ChaCha20 secret key object's CKA_VALUE attribute.

2.59.3 Salsa20 mechanism parameters

♦ CK_SALSA20_PARAMS; CK_SALSA_PARAMS_PTR

CK_SALSA20_PARAMS provides the parameters to the CKM_SALSA20 mechanism. It is defined as follows:

typedef struct CK_SALSA20_PARAMS {
    CK_BYTE_PTR pBlockCounter;
    CK_BYTE_PTR pNonce;
    CK_ULONG ulNonceBits;
} CK_SALSA20_PARAMS;

The fields of the structure have the following meanings:

    pBlockCounter pointer to block counter (64 bits)
    pNonce nonce
    ulNonceBits size of the nonce in bits (64 for classic and 192 for XSalsa20)

The block counter is used to address 512 bit blocks in the stream. In certain settings (e.g. disk encryption) it is necessary to address these blocks in random order, thus this counter is exposed here.

2.59.4 Salsa20 key generation

The Salsa20 key generation mechanism, denoted CKM_SALSA20_KEY_GEN, is a key generation mechanism for Salsa20.

It does not have a parameter.

The mechanism generates Salsa20 keys of 256 bits.

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

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of key sizes in bytes. As a practical matter, the key size for Salsa20 is fixed at 256 bits.
2.59.5 Salsa20 mechanism

Salsa20, denoted **CKM_SALSA20**, is a mechanism for single and multiple-part encryption and decryption based on the Salsa20 stream cipher. Salsa20 comes in two variants which only differ in the size and handling of their nonces, affecting the safety of using random nonces.

Salsa20 has a parameter, **CK_SALSA20_PARAMS**, which indicates the nonce and initial block counter value.

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

Table 224, Salsa20: Key and Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>Salsa20</td>
<td>Any</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>Salsa20</td>
<td>Any</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
</tbody>
</table>

For this mechanism, the **ulMinKeySize** and **ulMaxKeySize** fields of the **CK_MECHANISM_INFO** structure specify the supported range of ChaCha20 key sizes, in bits.

Table 225, Salsa20: Nonce sizes

<table>
<thead>
<tr>
<th>Variant</th>
<th>Nonce</th>
<th>Maximum message</th>
<th>Nonce generation</th>
</tr>
</thead>
</table>
| original | 64 bit | Virtually unlimited | 1st msg: nonce0=Rand om 
| | | | n th msg: nonce n-1++ |
| XSalsa20 | 192 bit | Virtually unlimited | Each nonce can be randomly generated. |

Nonces must not ever be reused with the same key. However due to the birthday paradox the original variant cannot guarantee that randomly generated nonces are never repeating. Thus the recommended way to handle this is to generate the first nonce randomly, then increase this for follow-up messages. Only the XSalsa20 has large enough nonces so that it is virtually impossible to trigger with randomly generated nonces the birthday paradox.

2.60 Poly1305

Poly1305 is a message authentication code designed by D.J Bernstein [POLY1305]. Poly1305 takes a 256 bit key and a message and produces a 128 bit tag that is used to verify the message.

Table 226, Poly1305 Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR 1</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_POLY1305_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_POLY1305</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

29 May 2019
2.60.1 Definitions

This section defines the key type “CKK_POLY1305” for type CK_KEY_TYPE as used in the
CKA_KEY_TYPE attribute of key objects.

Mechanisms:
- CKM_POLY1305_KEY_GEN
- CKM_POLY1305_MAC

2.60.2 Poly1305 secret key objects

Poly1305 secret key objects (object class CKO_SECRET_KEY, key type CKK_POLY1305) hold
Poly1305 keys. The following table defines the Poly1305 secret key object attributes, in addition to the
common attributes defined for this object class:

Table 227, Poly1305 Secret Key Object

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE</td>
<td>Byte array</td>
<td>Key length is fixed at 256 bits. Bit length restricted to a byte array.</td>
</tr>
<tr>
<td>CKA_VALUE_LEN</td>
<td>CK_ULONG</td>
<td>Length in bytes of key value</td>
</tr>
</tbody>
</table>

The following is a sample template for creating a Poly1305 secret key object:

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_POLY1305;
CK_UTF8CHAR label[] = "A Poly1305 secret key object";
CK_BYTE value[32] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_SIGN, &true, sizeof(true)},
    {CKA_VALUE, value, sizeof(value)}
};
```

2.60.3 Poly1305 mechanism

Poly1305, denoted **CKM_POLY1305**, is a mechanism for producing an output tag based on a 256 bit key
and arbitrary length input.

- It has no parameters.
- Signatures (MACs) produced by this mechanism will be fixed at 128 bits in size.

Table 228, Poly1305: Key and Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>Poly1305</td>
<td>Any</td>
<td>128 bits</td>
</tr>
<tr>
<td>C_Verify</td>
<td>Poly1305</td>
<td>Any</td>
<td>128 bits</td>
</tr>
</tbody>
</table>
2.61 Chacha20/Poly1305 and Salsa20/Poly1305 Authenticated Encryption / Decryption

The stream ciphers Salsa20 and ChaCha20 are normally used in conjunction with the Poly1305 authenticator, in such a construction they also provide Authenticated Encryption with Associated Data (AEAD). This section defines the combined mechanisms and their usage in an AEAD setting.

2.61.1 Definitions

Mechanisms:

CKM_CHACHA20_POLY1305
CKM_SALSA20_POLY1305

2.61.2 Usage

Generic ChaCha20, Salsa20, Poly1305 modes are described in [CHACHA], [SALSA] and [POLY1305].

To set up for ChaCha20/Poly1305 or Salsa20/Poly1305 use the following process. ChaCha20/Poly1305 and Salsa20/Poly1305 both use CK_SALSA20_CHACHA20_POLY1305_PARAM for Encrypt, Decrypt and CK_SALSA20_CHACHA20_POLY1305_MSG_PARAM for MessageEncrypt, and MessageDecrypt.

Encrypt:

- Set the Nonce length ulNonceLen in the parameter block. (this affects which variant of ChaCha20 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)
- Set the Nonce data pNonce in the parameter block.
- Set the AAD data pAAD and size ulAADLen in the parameter block. pAAD may be NULL if ulAADLen is 0.
- Call C_EncryptInit() for CKM_CHACHA20_POLY1305 or CKM_SALSA20_POLY1305 mechanism with parameters and key K.
- Call C_Encrypt(), or C_EncryptUpdate()\*10 C_EncryptFinal(), for the plaintext obtaining ciphertext and authentication tag output.

Decrypt:

- Set the Nonce length ulNonceLen in the parameter block. (this affects which variant of ChaCha20 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)
- Set the Nonce data pNonce in the parameter block.
- Set the AAD data pAAD and size ulAADLen in the parameter block. pAAD may be NULL if ulAADLen is 0.
- Call C_DecryptInit() for CKM_CHACHA20_POLY1305 or CKM_SALSA20_POLY1305 mechanism with parameters and key K.
- Call C_Decrypt(), or C_DecryptUpdate()\*1 C_DecryptFinal(), for the ciphertext, including the appended tag, obtaining plaintext output. Note: since CKM_CHACHA20_POLY1305 and CKM_SALSA20_POLY1305 are AEAD ciphers, no data should be returned until C_Decrypt() or C_DecryptFinal().

MessageEncrypt::

\*10 **indicates 0 or more calls may be made as required
- Set the Nonce length `ulNonceLen` in the parameter block. (this affects which variant of Chacha20 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)

- Set the Nonce data `pNonce` in the parameter block.

- Set `pTag` to hold the tag data returned from `C_EncryptMessage()` or the final `C_EncryptMessageNext()`.

- Call `C_MessageEncryptInit()` for `CKM_CHACHA20_POLY1305` or `CKM_SALSA20_POLY1305` mechanism with key `K`.

- Call `C_EncryptMessage()`, or `C_EncryptMessageBegin` followed by `C_EncryptMessageNext()`\(^{11}\). The mechanism parameter is passed to all three of these functions.

- Call `C_MessageEncryptFinal()` to close the message decryption.

**MessageDecrypt:**

- Set the Nonce length `ulNonceLen` in the parameter block. (this affects which variant of Chacha20 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)

- Set the Nonce data `pNonce` in the parameter block.

- Set the tag data `pTag` in the parameter block before `C_DecryptMessage` or the final `C_DecryptMessageNext()`.

- Call `C_MessageDecryptInit()` for `CKM_CHACHA20_POLY1305` or `CKM_SALSA20_POLY1305` mechanism with key `K`.

- Call `C_DecryptMessage()`, or `C_DecryptMessageBegin` followed by `C_DecryptMessageNext()`\(^{12}\). The mechanism parameter is passed to all three of these functions.

- Call `C_MessageDecryptFinal()` to close the message decryption.

`ulNonceLen` is the length of the nonce in bits.

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.

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.

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.

### 2.61.3 ChaCha20/Poly1305 and Salsa20/Poly1305 Mechanism parameters

- **`CK_SALSA20_CHACHA20_POLY1305_PARAMS;`**
- `CK_SALSA20_CHACHA20_POLY1305_PARAMS_PTR`

`CK_SALSA20_CHACHA20_POLY1305_PARAMS` is a structure that provides the parameters to the `CKM_CHACHA20_POLY1305` and `CKM_SALSA20_POLY1305` mechanisms. It is defined as follows:

```c
typedef struct CK_SALSA20_CHACHA20_POLY1305_PARAMS {
```

\(^{11}\) “*” indicates 0 or more calls may be made as required

\(^{12}\) “*” indicates 0 or more calls may be made as required
CK_BYTE_PTR pNonce;
CK_ULONG ulNonceLen;
CK_BYTE_PTR pAAD;
CK_ULONG ulAADLen;
}

The fields of the structure have the following meanings:

pNonce  nonce (This should be never re-used with the same key.)

ulNonceLen length of nonce in bits (is 64 for original, 96 for IETF (only for chacha20) and 192 for xchacha20/xsalsa20 variant)

pAAD  pointer to additional authentication data. This data is authenticated but not encrypted.

ulAADLen length of pAAD in bytes.

CK_SALSA20_CHACHA20_POLY1305_PARAMS_PTR is a pointer to a CK_SALSA20_CHACHA20_POLY1305_PARAMS.

CK_SALSA20_CHACHA20_POLY1305_PARAMS;
CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS_PTR
CK_CHACHA20_POLY1305_PARAMS is a structure that provides the parameters to the CKM_CHACHA20_POLY1305 mechanism. It is defined as follows:

typedef struct CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS {
  CK_BYTE_PTR pNonce;
  CK_ULONG ulNonceLen;
  CK_BYTE_PTR pTag;
} CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS;

The fields of the structure have the following meanings:

pNonce  pointer to nonce

ulNonceLen length of nonce in bits. The length of the influences which variant of the ChaCha20 will be used (64 original, 96 IETF(only for ChaCha20), 192 XChaCha20/XSalsa20)

pTag  location of the authentication tag which is returned on MessageEncrypt, and provided on MessageDecrypt.

CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS_PTR is a pointer to a CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS.

2.62 HKDF Mechanisms

Details for HKDF key derivation mechanisms can be found in [RFC 5869].

Table 229, HKDF Mechanisms vs. Functions
2.62.1 Definitions

Mechanisms:

- CKM_HKDF_DERIVE
- CKM_HKDF_DATA
- CKM_HKDF_KEY_GEN

Key Types:

- CKK_HKDF

2.62.2 HKDF mechanism parameters

CKM_HKDF_DERIVE and CKM_HKDF_DATA mechanisms. It is defined as follows:

```c
typedef struct CK_HKDF_PARAMS {
    CK_BOOL bExtract;
    CK_BOOL bExpand;
    CK_MECHANISM_TYPE prfHashMechanism;
    CK_ULONG ulSaltType;
    CK_BYTE_PTR pSalt;
    CK_ULONG ulSaltLen;
    CK_HANDLE hSaltKey;
    CK_BYTE_PTR pInfo;
    CK_ULONG ulInfoLen;
} CK_HKDF_PARAMS;
```

The fields of the structure have the following meanings:

- **bExtract** execute the extract portion of HKDF.
- **bExpand** execute the expand portion of HKDF.
- **prfHashMechanism** base hash used for the HMAC in the underlying HKDF operation.
- **ulSaltType** specifies how the salt for the extract portion of the KDF is supplied.
- **CKF_HKDF_SALT_NULL** no salt is supplied.
- **CKF_HKDF_SALT_DATA** salt is supplied as a data in pSalt with length ulSaltLen.
CKF_HKDF_SALT_KEY salt is supplied as a key in hSaltKey.

pSalt    pointer to the salt.

ulSaltLen   length of the salt pointed to in pSalt.

hSaltKey   object handle to the salt key.

pInfo   info string for the expand stage.

ulInfoLen   length of the info string for the expand stage.

CK_HKDF_PARAMS_PTR is a pointer to a CK_HKDF_PARAMS.

2.62.3 HKDF derive

HKDF derivation implements the HKDF as specified in RFC 5869. The two booleans bExtract and bExpand control whether the extract section of the HKDF or the expand section of the HKDF is in use.

It has a parameter, a CK_HKDF_PARAMS structure, which allows for the passing of the salt and or the expansion info. The structure contains the bools bExtract and bExpand which control whether the extract or expand portions of the HKDF is to be used. This structure is defined in Section 2.62.2.

The input key must be of type CKK_HKDF or CKK_GENERIC_SECRET and the length must be the size of the underlying hash function specified in prfHashMechanism. The exception is a data object which has the same size as the underlying hash function, and which may be supplied as an input key. In this case bExtract should be true and non-null salt should be supplied.

Either bExtract or bExpand must be set to true. If they are both set to true, input key is first extracted then expanded. The salt is used in the extraction stage. If bExtract is set to true and no salt is given, a ‘zero’ salt (salt whose length is the same as the underlying hash and values all set to zero) is used as specified by the RFC. If bExpand is set to true, CKA_VALUE_LEN should be set to the desired key length. If it is false CKA_VALUE_LEN may be set to the length of the hash, but that is not necessary as the mechanism will supply this value. The salt should be ignored if bExtract is false. The pInfo should be ignored if bExpand is set to false.

The mechanism also contributes the CKA_CLASS, and CKA_VALUE attributes to the new key. Other attributes may be specified in the template, or else are assigned default values.

The template sent along with this mechanism during a C_DeriveKey call may indicate that the object class is CKO_SECRET_KEY. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

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

- The CKA_SENSITIVE and CKA_EXTRACTABLE attributes in the template for the new key can both be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some default value.

- If the base key has its CKA_ALWAYS_SENSITIVE attribute set to CK_FALSE, then the derived key will as well. If the base key has its CKA_ALWAYS_SENSITIVE attribute set to CK_TRUE, then the derived key has its CKA_ALWAYS_SENSITIVE attribute set to the same value as its CKA_SENSITIVE attribute.

- Similarly, if the base key has its CKA_NEVER_EXTRACTABLE attribute set to CK_FALSE, then the derived key will, too. If the base key has its CKA_NEVER_EXTRACTABLE attribute set to CK_TRUE, then the derived key has its CKA_NEVER_EXTRACTABLE attribute set to the opposite value from its CKA_EXTRACTABLE attribute.
2.62.4 HKDF Data
HKDF Data derive mechanism, denoted CKM_HKDF_DATA, is identical to HKDF Derive except the output is a CKO_DATA object whose value is the result to the derive operation. Some tokens may restrict what data may be successfully derived based on the pinfo portion of the CK_HKDF_PARAMS. All tokens must minimally support bExtract set to true and bInfo values which start with the value “tls1.3 vi”. Future additional required combinations may be specified in the profile document and applications could then query the appropriate profile before depending on the mechanism.

2.62.5 HKDF Key gen
HKDF key gen, denoted CKM_HKDF_KEYGEN generates a new random HKDF key. CKA_VALUE_LENGTH must be set in the template.

2.63 NULL Mechanism
CKM_NULL is a mechanism used to implement the trivial pass-through function.

Table 230, CKM_NULL Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/ Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_NULL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

1SR = SignRecover, VR = VerifyRecover

2.63.1 Definitions
Mechanisms:

CKM_NULL

2.63.2 CKM_NULL mechanism parameters
CKM_NULL does not have a parameter.

When used for encrypting / decrypting data, the input data is copied unchanged to the output data.
When used for signing, the input data is copied to the signature. When used for signature verification, it compares the input data and the signature, and returns CKR_OK (indicating that both are identical) or CKR_SIGNATURE_INVALID.
When used for digesting data, the input data is copied to the message digest.
When used for wrapping a private or secret key object, the wrapped key will be identical to the key to be wrapped. When used for unwrapping, a new object with the same value as the wrapped key will be created.
When used for deriving a key, the derived key has the same value as the base key.
3 PKCS #11 Implementation Conformance

An implementation is a conforming implementation if it meets the conditions specified in one or more server profiles specified in [PKCS11-Prof].

If a PKCS #11 implementation claims support for a particular profile, then the implementation SHALL conform to all normative statements within the clauses specified for that profile and for any subclauses to each of those clauses.
Appendix A. Acknowledgments

The following individuals have participated in the creation of this specification and are gratefully acknowledged:

Participants:

List needs to be pasted in here

Gil Abel, Athena Smartcard Solutions, Inc.
Warren Armstrong, QuintessenceLabs
Jeff Bartell, Semper Foris Solutions LLC
Peter Bartok, Venafi, Inc.
Anthony Berglas, Cryptsoft
Joseph Brand, Semper Fortis Solutions LLC
Kelley Burgin, National Security Agency
Robert Burns, Thales e-Security
Wan-Teh Chang, Google Inc.
Hai-May Chao, Oracle
Janice Cheng, Vormetric, Inc.
Sangrae Cho, Electronics and Telecommunications Research Institute (ETRI)
Doron Cohen, SafeNet, Inc.
Fadi Cotran, Futurex
Tony Cox, Cryptsoft
Christopher Duane, EMC
Chris Dunn, SafeNet, Inc.
Valerie Fenwick, Oracle
Terry Fletcher, SafeNet, Inc.
Susan Gleeson, Oracle
Sven Gossel, Charismathics
John Green, QuintessenceLabs
Robert Griffin, EMC
Paul Grojean, Individual
Peter Gutmann, Individual
Dennis E. Hamilton, Individual
Thomas Hardjono, M.I.T.
Tim Hudson, Cryptsoft
Gershon Janssen, Individual
Seunghun Jin, Electronics and Telecommunications Research Institute (ETRI)
Wang Jingman, Feitan Technologies
Andrey Jivsov, Symantec Corp.
Mark Joseph, P6R
Stefan Kaesar, Infineon Technologies
Greg Kazmierczak, Wave Systems Corp.
Mark Knight, Thales e-Security
Darren Krahn, Google Inc.
Alex Krasnov, Infineon Technologies AG
Dina Kurktchi-Nimeh, Oracle
Mark Lambiase, SecureAuth Corporation
Lawrence Lee, GoTrust Technology Inc.
John Leiseboer, QuintessenceLabs
Sean Leon, Infineon Technologies
Geoffrey Li, Infineon Technologies
Howie Liu, Infineon Technologies
Hal Lockhart, Oracle
Robert Lockhart, Thales e-Security
Dale Moberg, Axway Software
Darren Moffat, Oracle
Valery Osheter, SafeNet, Inc.
Sean Parkinson, EMC
Rob Philpott, EMC
Mark Powers, Oracle
Ajai Puri, SafeNet, Inc.
Robert Relyea, Red Hat
Saikat Saha, Oracle
Subhash Sankuratripati, NetApp
Anthony Scarpino, Oracle
Johann Schoetz, Infineon Technologies AG
Rayees Shamsuddin, Wave Systems Corp.
Radhika Siravara, Oracle
Brian Smith, Mozilla Corporation
David Smith, Venafi, Inc.
Ryan Smith, Futurex
Jerry Smith, US Department of Defense (DoD)
Oscar So, Oracle
Graham Steel, Cryptosense
Michael Stevens, QuintessenceLabs
Michael StJohns, Individual
Jim Susoy, P6R
Sander Temme, Thales e-Security
Kiran Thota, VMware, Inc.
Walter-John Turnes, Gemini Security Solutions, Inc.
Stef Walter, Red Hat
James Wang, Vormetric
Jeff Webb, Dell
Peng Yu, Feitian Technologies
Appendix B. Manifest Constants

The following definitions can be found in the appropriate computer language files referenced on the title page of this specification. Also, refer to [PKCS11_BASE] and [PKCS11_HIST] for additional definitions.

```c
/*
 * Copyright © Oasis Open 2019. All right reserved.
 * OASIS trademark, IPR and other policies apply.
 * https://www.oasis-open.org/policies-guidelines/ipr
 */
```

B.1 Object classes

```c
#define CKO_DATA 0x00000000
#define CKO_CERTIFICATE 0x00000001
#define CKO_PUBLIC_KEY 0x00000002
#define CKO_PRIVATE_KEY 0x00000003
#define CKO_SECRET_KEY 0x00000004
#define CKO_HW_FEATURE 0x00000005
#define CKO_DOMAIN_PARAMETERS 0x00000006
#define CKO_MECHANISM 0x00000007
#define CKO_OTP_KEY 0x00000008
#define CKO_PROFILE 0x00000009
#define CKO_VENDOR_DEFINED 0x80000000
```

B.2 Key types

```c
#define CKK_RSA 0x00000000
#define CKK_DSA 0x00000001
#define CKK_DH 0x00000002
#define CKK_EC 0x00000003
#define CKK_X9_42_DH 0x00000004
#define CKK_KEA 0x00000005
#define CKK_GENERIC_SECRET 0x00000006
#define CKK_RC2 0x00000007
#define CKK_RC4 0x00000008
#define CKK_DES 0x00000009
#define CKK_DES2 0x0000000A
#define CKK_DES3 0x0000000B
#define CKK_CAST 0x0000000C
#define CKK_CAST3 0x0000000D
#define CKK_CAST128 0x0000000E
#define CKK_RC5 0x0000000F
#define CKK_IDEA 0x00000010
#define CKK_SKIPJACK 0x00000011
```
B.3 Key derivation functions

#define CKD_NULL 0x00000001
#define CKD_SHA1_KDF 0x00000002
#define CKD_SHA1_KDF_ASN1 0x00000003
#define CKD_SHA1_KDF_CONCATENATE 0x00000004
#define CKD_SHA224_KDF 0x00000005
#define CKD_SHA256_KDF 0x00000006
#define CKD_SHA3_224_HMAC 0x00000022
#define CKD_SHA3_256_HMAC 0x00000027
#define CKD_SHA3_384_HMAC 0x0000002C
#define CKD_SHA3_512_HMAC 0x0000002D
#define CKD_SHA224_KDF 0x00000005
#define CKD_SHA256_KDF 0x00000006
#define CKD_SHA3_224_HMAC 0x00000022
#define CKD_SHA3_256_HMAC 0x00000027
#define CKD_SHA3_384_HMAC 0x0000002C
#define CKD_SHA3_512_HMAC 0x0000002D
#define CKD_SHA224_KDF 0x00000005
#define CKD_SHA256_KDF 0x00000006
#define CKD_SHA3_224_HMAC 0x00000022
#define CKD_SHA3_256_HMAC 0x00000027
#define CKD_SHA3_384_HMAC 0x0000002C
#define CKD_SHA3_512_HMAC 0x0000002D
#define CKD_SHA224_KDF 0x00000005
#define CKD_SHA256_KDF 0x00000006
#define CKD_SHA3_224_HMAC 0x00000022
#define CKD_SHA3_256_HMAC 0x00000027
#define CKD_SHA3_384_HMAC 0x0000002C
#define CKD_SHA3_512_HMAC 0x0000002D
#define CKD_SHA224_KDF 0x00000005
#define CKD_SHA256_KDF 0x00000006
#define CKD_SHA3_224_HMAC 0x00000022
#define CKD_SHA3_256_HMAC 0x00000027
#define CKD_SHA3_384_HMAC 0x0000002C
#define CKD_SHA3_512_HMAC 0x0000002D
#define CKD_SHA224_KDF 0x00000005
#define CKD_SHA256_KDF 0x00000006
#define CKD_SHA3_224_HMAC 0x00000022
#define CKD_SHA3_256_HMAC 0x00000027
#define CKD_SHA3_384_HMAC 0x0000002C
#define CKD_SHA3_512_HMAC 0x0000002D
#define CKD_SHA224_KDF 0x00000005
#define CKD_SHA256_KDF 0x00000006
#define CKD_SHA3_224_HMAC 0x00000022
#define CKD_SHA3_256_HMAC 0x00000027
#define CKD_SHA3_384_HMAC 0x0000002C
#define CKD_SHA3_512_HMAC 0x0000002D
#define CKD_SHA224_KDF 0x00000005
#define CKD_SHA256_KDF 0x00000006
#define CKD_SHA3_224_HMAC 0x00000022
#def
B.4 Mechanisms

```c
#define CKM_RSA_PKCS_KEY_PAIR_GEN 0x00000000
#define CKM_RSA_PKCS 0x00000001
#define CKM_RSA_9796 0x00000002
#define CKM_RSA_X_509 0x00000003
#define CKM_MD2_RSA_PKCS 0x00000004
#define CKM_MD5_RSA_PKCS 0x00000005
#define CKM_SHA1_RSA_PKCS 0x00000006
#define CKM_RIPEMD128_RSA_PKCS 0x00000007
#define CKM_RIPEMD160_RSA_PKCS 0x00000008
#define CKM_RSA_PKCS_OAEP 0x00000009
#define CKM_RSA_X9_31_KEY_PAIR_GEN 0x0000000A
#define CKM_RSA_X9_31 0x0000000B
#define CKM_SHA1_RSA_X9_31 0x0000000C
#define CKM_RSA_PKCS_PSS 0x0000000D
#define CKM_SHA1_RSA_PKCS_PSS 0x0000000E
#define CKM_DSA_KEY_PAIR_GEN 0x00000010
#define CKM_DSA 0x00000011
#define CKM_DSA_SHA1 0x00000012
#define CKM_DSA_FIPS_G_GEN 0x00000013
#define CKM_DSA_SHA224 0x00000014
#define CKM_DSA_SHA256 0x00000015
#define CKM_DSA_SHA384 0x00000016
#define CKM_DSA_SHA256_KEY_PAIR_GEN 0x00000017
#define CKM_DSA_SHA256 0x00000018
#define CKM_DSA_SHA384 0x00000019
#define CKM_DSA_SHA3_224 0x0000001A
#define CKM_DSA_SHA3_256 0x0000001B
#define CKM_DSA_SHA3_384 0x0000001C
#define CKM_DSA_SHA3_512 0x0000001D
#define CKM_DSA_SHA3_224_KEY_PAIR_GEN 0x0000001E
#define CKM_DSA_SHA3_256_KEY_PAIR_GEN 0x0000001F
#define CKM_DSA_SHA3_384_KEY_PAIR_GEN 0x00000020
#define CKM_DSA_SHA3_512_KEY_PAIR_GEN 0x00000021
#define CKM_DSA_SHA3_224 0x00000022
#define CKM_DSA_SHA3_256 0x00000023
#define CKM_DSA_SHA3_384 0x00000024
#define CKM_DSA_SHA3_512 0x00000025
```

#define CKM_DSA_SHA3_512 0x0000001B
#define CKM_DH_PKCS_KEY_PAIR_GEN 0x00000020
#define CKM_DH_PKCS_DERIVE 0x00000021
#define CKM_X9_42_DH_KEYPAIR_GEN 0x00000030
#define CKM_X9_42_DH_DERIVE 0x00000031
#define CKM_X9_42_DH_HYBRID_DERIVE 0x00000032
#define CKM_X9_42_MQV_DERIVE 0x00000033
#define CKM_SHA256_RSA_PKCS 0x00000040
#define CKM_SHA384_RSA_PKCS 0x00000041
#define CKM_SHA512_RSA_PKCS 0x00000042
#define CKM_SHA256_RSA_PKCS_PSS 0x00000043
#define CKM_SHA384_RSA_PKCS_PSS 0x00000044
#define CKM_SHA512_RSA_PKCS_PSS 0x00000045
#define CKM_SHA256_RSA_PKCS_PSS 0x00000046
#define CKM_SHA384_RSA_PKCS_PSS 0x00000047
#define CKM_SHA512_RSA_PKCS_PSS 0x00000048
#define CKM_SHA512_224 0x00000049
#define CKM_SHA512_224_HMAC 0x0000004A
#define CKM_SHA512_224_HMAC_GENERAL 0x0000004B
#define CKM_SHA512_224_KEY_DERIVATION 0x0000004C
#define CKM_SHA512_256 0x0000004D
#define CKM_SHA512_256_HMAC 0x0000004E
#define CKM_SHA512_256_HMAC_GENERAL 0x0000004F
#define CKM_SHA512_256_KEY_DERIVATION 0x00000050
#define CKM_SHA512_T 0x00000051
#define CKM_SHA512_T_HMAC 0x00000052
#define CKM_SHA512_T_HMAC_GENERAL 0x00000053
#define CKM_SHA3_256_RSA_PKCS 0x00000054
#define CKM_SHA3_384_RSA_PKCS 0x00000055
#define CKM_SHA3_512_RSA_PKCS 0x00000056
#define CKM_SHA3_256_RSA_PKCS_PSS 0x00000057
#define CKM_SHA3_384_RSA_PKCS_PSS 0x00000058
#define CKM_SHA3_512_RSA_PKCS_PSS 0x00000059
#define CKM_SHA3_224_RSA_PKCS 0x0000005A
#define CKM_SHA3_224_RSA_PKCS_PSS 0x0000005B
#define CKM_RC2_KEY_GEN 0x00000100
#define CKM_RC2_ECB 0x000000101
#define CKM_RC2_CBC 0x000000102
#define CKM_RC2_MAC 0x000000103
#define CKM_RC2_MAC_GENERAL 0x000000104
#define CKM_RC2_CBC_PAD 0x000000105
#define CKM_RC3_CBC 0x000000106
#define CKM_RC4_KEY_GEN 0x0000001010
#define CKM_DES_KEY_GEN 0x000000120
#define CKM_DES_ECB 0x000000121
#define CKM_DES_CBC 0x000000122
#define CKM_DES_MAC 0x000000123
#define CKM_DES_MAC_GENERAL 0x000000124
#define CKM_DES_CBC_PAD 0x00000125
#define CKM_DES2_KEY_GEN 0x00000130
#define CKM_DES3_KEY_GEN 0x00000131
#define CKM_DES3_ECB 0x00000132
#define CKM_DES3_CBC 0x00000133
#define CKM_DES3_MAC 0x00000134
#define CKM_DES3_MAC_GENERAL 0x00000135
#define CKM_DES3_CMAC 0x00000138
#define CKM_CDMF_KEY_GEN 0x00000140
#define CKM_CDMF_ECB 0x00000141
#define CKM_CDMF_CBC 0x00000142
#define CKM_CDMF_MAC 0x00000143
#define CKM_CDMF_MAC_GENERAL 0x00000144
#define CKM_DES_OFB64 0x00000150
#define CKM_DES_OFB8 0x00000151
#define CKM_DES_CFB64 0x00000152
#define CKM_DES_CFB8 0x00000153
#define CKM_MD2 0x00000200
#define CKM_MD2_HMAC 0x00000201
#define CKM_MD2_HMAC_GENERAL 0x00000202
#define CKM_MD5 0x00000210
#define CKM_MD5_HMAC 0x00000211
#define CKM_MD5_HMAC_GENERAL 0x00000212
#define CKM_SHA_1 0x00000220
#define CKM_SHA_1_HMAC 0x00000221
#define CKM_SHA_1_HMAC_GENERAL 0x00000222
#define CKM_RIPEMD128 0x00000230
#define CKM_RIPEMD128_HMAC 0x00000231
#define CKM_RIPEMD128_HMAC_GENERAL 0x00000232
#define CKM_RIPEMD160 0x00000240
#define CKM_RIPEMD160_HMAC 0x00000241
#define CKM_RIPEMD160_HMAC_GENERAL 0x00000242
#define CKM_SHA256 0x00000250
#define CKM_SHA256_HMAC 0x00000251
#define CKM_SHA256_HMAC_GENERAL 0x00000252
#define CKM_SHA384 0x00000260
#define CKM_SHA384_HMAC 0x00000261
#define CKM_SHA384_HMAC_GENERAL 0x00000262
#define CKM_SHA512 0x00000270
#define CKM_SHA512_HMAC 0x00000271
#define CKM_SHA512_HMAC_GENERAL 0x00000272
#define CKM_SECURID_KEY_GEN 0x00000280
#define CKM_SECURID 0x00000282
#define CKM_HOTP_KEY_GEN 0x00000290
#define CKM_HOTP 0x00000291
#define CKM_ACTI_KEY_GEN 0x000002A1
#define CKM_SHA3_256 0x000002B0
#define CKM_SHA3_256_HMAC 0x000002B1
#define CKM_SHA3_256_HMAC_GENERAL 0x000002B2
#define CKM_SHA3_256_KEY_GEN 0x000002B3
#define CKM_SHA3_224 0x000002B5
#define CKM_SHA3_224_HMAC 0x000002B6
#define CKM_SHA3_224_HMAC_GENERAL 0x000002B7
#define CKM_SHA3_224_KEY_GEN 0x000002B8
#define CKM_SHA3_384 0x000002B9
#define CKM_SHA3_384_HMAC 0x000002C0
#define CKM_SHA3_384_HMAC_GENERAL 0x000002C1
#define CKM_SHA3_384_KEY_GEN 0x000002C2
#define CKM_SHA3_512 0x000002C3
#define CKM_SHA3_512_HMAC 0x000002C4
#define CKM_SHA3_512_HMAC_GENERAL 0x000002C5
#define CKM_SHA3_512_KEY_GEN 0x000002C6
#define CKM_CAST_KEY_GEN 0x00000300
#define CKM_CAST_ECB 0x00000301
#define CKM_CAST_CBC 0x00000302
#define CKM_CAST_MAC 0x00000303
#define CKM_CAST_MAC_GENERAL 0x00000304
#define CKM_CAST_CBC_PAD 0x00000305
#define CKM_CAST3_KEY_GEN 0x00000310
#define CKM_CAST3_ECB 0x00000311
#define CKM_CAST3_CBC 0x00000312
#define CKM_CAST3_MAC 0x00000313
#define CKM_CAST3_MAC_GENERAL 0x00000314
#define CKM_CAST3_CBC_PAD 0x00000315
#define CKM_CAST128_KEY_GEN 0x00000320
#define CKM_CAST128_ECB 0x00000321
#define CKM_CAST128_CBC 0x00000322
#define CKM_CAST128_MAC 0x00000323
#define CKM_CAST128_MAC_GENERAL 0x00000324
#define CKM_CAST128_CBC_PAD 0x00000325
#define CKM_RC5_KEY_GEN 0x00000330
#define CKM_RC5_ECB 0x00000331
#define CKM_RC5_CBC 0x00000332
#define CKM_RC5_MAC 0x00000333
#define CKM_RC5_MAC_GENERAL 0x00000334
#define CKM_IDEA_KEY_GEN 0x00000340
#define CKM_IDEA_ECB 0x00000341
#define CKM_IDEA_CBC 0x00000342
#define CKM_IDEA_CBC_PAD 0x00000343
#define CKM_IDEA_MAC 0x00000343
#define CKM_IDEA_MAC_GENERAL 0x00000344
#define CKM_IDEA_CBC_PAD 0x00000345
#define CKM_GENERIC_SECRET_KEY_GEN 0x00000350
#define CKM_CONCATENATE_BASE_AND_KEY 0x00000360
#define CKM_CONCATENATE_BASE_AND_DATA 0x00000362
#define CKM_CONCATENATE_DATA_AND_BASE 0x00000363
#define CKM_XOR_BASE_AND_DATA 0x00000364
#define CKM_EXTRACT_KEY_FROM_KEY 0x00000365
#define CKM_SSL3_PRE_MASTER_KEY_GEN 0x00000370
#define CKM_SSL3_MASTER_KEY_DERIVE 0x00000371
#define CKM_SSL3_KEY_AND_MAC_DERIVE 0x00000372
#define CKM_SSL3_MASTER_KEY_DERIVE_DH 0x00000373
#define CKM_TLS_PRE_MASTER_KEY_GEN 0x00000374
#define CKM_TLS_MASTER_KEY_DERIVE 0x00000375
#define CKM_TLS_KEY_AND_MAC_DERIVE 0x00000376
#define CKM_TLS_MASTER_KEY_DERIVE_DH 0x00000377
#define CKM_TLS_PRF 0x00000378
#define CKM_SSL3_MD5_MAC 0x00000380
#define CKM_SSL3_SHA1_MAC 0x00000381
#define CKM_MD5_KEY_DERIVATION 0x00000390
#define CKM_MD2_KEY_DERIVATION 0x00000391
#define CKM_SHA1_KEY_DERIVATION 0x00000392
#define CKM_SHA256_KEY_DERIVATION 0x00000393
#define CKM_SHA384_KEY_DERIVATION 0x00000394
#define CKM_SHA512_KEY_DERIVATION 0x00000395
#define CKM_SHA3_256_KEY_DERIVE 0x00000396
#define CKM_SHA3_224_KEY_DERIVE 0x00000397
#define CKM_SHA3_384_KEY_DERIVE 0x00000398
#define CKM_SHA3_512_KEY_DERIVE 0x00000399
#define CKM_SHA3_128_KEY_DERIVE 0x0000039A
#define CKM_SHA3_256_KEY_DERIVE 0x0000039B
#define CKM_SHA3_224_KEY_DERIVE 0x0000039C
#define CKM_SHA3_384_KEY_DERIVE 0x0000039D
#define CKM_SHA3_512_KEY_DERIVE 0x0000039E
#define CKM_SHA3_128_KEY_DERIVE 0x0000039F
#define CKM_SHA3_256_KEY_DERIVE 0x000003A0
#define CKM_SHA3_224_KEY_DERIVE 0x000003A1
#define CKM_SHA3_384_KEY_DERIVE 0x000003A2
#define CKM_SHA3_512_KEY_DERIVE 0x000003A3
#define CKM_SHA3_128_KEY_DERIVE 0x000003A4
#define CKM_SHA3_256_KEY_DERIVE 0x000003A5
#define CKM_SHA3_224_KEY_DERIVE 0x000003A6
#define CKM_SHA3_384_KEY_DERIVE 0x000003A7
#define CKM_SHA3_512_KEY_DERIVE 0x000003A8
#define CKM_SHA3_128_KEY_DERIVE 0x000003A9
#define CKM_SHA3_256_KEY_DERIVE 0x000003AA
#define CKM_SHA3_224_KEY_DERIVE 0x000003AB
#define CKM_SHA3_384_KEY_DERIVE 0x000003AC
#define CKM_SHA3_512_KEY_DERIVE 0x000003AD
#define CKM_SP800_108_DOUBLE_PIPELINE_KDF 0x000003AE
#define CKM_PPKS5_PBKD2 0x000003B0
#define CKM_PBA_SHA1_WITH_SHA1_HMAC 0x000003C0
#define CKM_WTLS_PRE_MASTER_KEY_GEN 0x000003D0
#define CKM_WTLS_MASTER_KEY_DERIVE 0x000003D1
#define CKM_WTLS_MASTER_KEY_DERIVE_DH_ECC 0x000003D2
#define CKM_WTLS_PRF 0x000003D3
#define CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE 0x000003D4
#define CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE 0x000003D5
#define CKM_TLS12_MAC 0x000003D8
#define CKM_TLS12_KDF 0x000003D9
#define CKM_TLS12_MASTER_KEY_DERIVE 0x000003E0
#define CKM_TLS12_MASTER_KEY_DERIVE_DH 0x000003E2
#define CKM_TLS12_KEY_AND_MAC_DERIVE 0x000003E3
#define CKM_TLS12_MASTER_KEY_DERIVE_E 0x000003E4
#define CKM_TLS12_KEY_SAFE_DERIVE 0x000003E5
#define CKM_TLS_MAC 0x000003E6
#define CKM_TLS_KDF 0x000003E7
#define CKM_KEY_WRAP_LYNKS 0x00000400
#define CKM_KEY_WRAP_SET_OAEP 0x00000401
#define CKM_CMS_SIG 0x00000500
#define CKM_KIP_DERIVE 0x00000510
#define CKM_KIP_WRAP 0x00000511
#define CKM_KIP_MAC 0x00000512
#define CKM_CAMELLIA_KEY_GEN 0x00000550
#define CKM_CAMELLIA_ECB 0x00000551
#define CKM_CAMELLIA_CBC 0x00000552
#define CKM_CAMELLIA_MAC 0x00000553
#define CKM_CAMELLIA_MAC_GENERAL 0x00000554
#define CKM_CAMELLIA_CBC_PAD 0x00000555
#define CKM_CAMELLIA_ECB_ENCRYPT_DATA 0x00000556
#define CKM_CAMELLIA_CBC_ENCRYPT_DATA 0x00000557
#define CKM_CAMELLIA_CTR 0x00000558
#define CKM_ARIA_KEY_GEN 0x00000560
#define CKM_ARIA_ECB 0x00000561
#define CKM_ARIA_CBC 0x00000562
#define CKM_ARIA_MAC 0x00000563
#define CKM_ARIA_MAC_GENERAL 0x00000564
#define CKM_ARIA_CBC_PAD 0x00000565
#define CKM_ARIA_ECB_ENCRYPT_DATA 0x00000566
#define CKM_ARIA_CBC_ENCRYPT_DATA 0x00000567
#define CKM_SEED_KEY_GEN 0x00000560
#define CKM_SEED_ECB 0x00000561
#define CKM_SEED_CBC 0x00000562
#define CKM_SEED_MAC 0x00000563
#define CKM_SEED_MAC_GENERAL 0x00000564
#define CKM_SEED_CBC_PAD 0x00000565
#define CKM_SEED_ECB_ENCRYPT_DATA 0x00000566
#define CKM_SEED_CBC_ENCRYPT_DATA 0x00000567
#define CKM_SKIPJACK_KEY_GEN 0x00001000
#define CKM_SKIPJACK_ECB64 0x00001001
#define CKM_SKIPJACK_CBC64 0x00001002
#define CKM_SKIPJACK_OFB64 0x00001003
#define CKM_SKIPJACK_CFB64 0x00001004
#define CKM_SKIPJACK_CFB32 0x00001005
#define CKM_SKIPJACK_CFB16 0x00001006
#define CKM_SKIPJACK_CFB8 0x00001007
#define CKM_SKIPJACK_WRAP 0x00001008
#define CKM_SKIPJACK_PRIVATE_WRAP 0x00001009
#define CKM_SKIPJACK_RELAYX 0x0000100A
#define CKM_KEA_KEY_PAIR_GEN 0x00001010
#define CKM_KEA_KEY_DERIVE 0x00001011
#define CKM_KEA_DERIVE 0x00001012
#define CKM_FORTEZZA_TIMESTAMP 0x00001020
#define CKM_BATON_KEY_GEN 0x00001030
#define CKM_BATON_ECB128 0x00001031
#define CKM_BATON_ECB96 0x00001032
#define CKM_BATON_CBC128 0x00001033
#define CKM_BATON_CBC1 0x00001034
#define CKM_BATON_COUNTER 0x00001035
#define CKM_BATON_WRAP 0x00001036
#define CKM_BATON_SHUFFLE 0x00001037
#define CKM_EC_KEY_PAIR_GEN 0x00001040
#define CKM_ECDSA 0x00001041
#define CKM_ECDSA_SHA1 0x00001042
#define CKM_ECDSA_SHA224 0x00001043
#define CKM_ECDSA_SHA256 0x00001044
#define CKM_ECDSA_SHA384 0x00001045
#define CKM_ECDSA_SHA512 0x00001046
#define CKM_ECDSA_SHA3_224 0x00001047
#define CKM_ECDSA_SHA3_256 0x00001048
#define CKM_ECDSA_SHA3_384 0x00001049
#define CKM_ECDSA_SHA3_512 0x0000104A
#define CKM_ECDH1_DERIVE 0x00001050
#define CKM_ECDH1_COFACTOR_DERIVE 0x00001051
#define CKM_ECDH_AES_KEY_WRAP 0x00001052
#define CKM_RSA_AES_KEY_WRAP 0x00001053
#define CKM_JUNIPER_KEY_GEN 0x00001054
#define CKM_JUNIPER_ECB128 0x00001055
#define CKM_JUNIPER_CBC128 0x00001056
#define CKM_JUNIPER_COUNTER 0x00001057
#define CKM_JUNIPER_WRAP 0x00001058
#define CKM_JUNIPER_SHUFFLE 0x00001059
#define CKM_JUNIPER_KEY_PAIR_GEN 0x0000105A
#define CKM_JUNIPER_ECB1 0x0000105B
#define CKM_JUNIPER_CBC1 0x0000105C
#define CKM_JUNIPER_COUNTER 0x0000105D
#define CKM_JUNIPER_WRAP 0x0000105E
#define CKM_JUNIPER_SHUFFLE 0x0000105F
#define CKM_FASTHASH 0x00001060
#define CKM_AES_XTS 0x00001071
#define CKM_AES_XTS_KEY_GEN 0x00001072
#define CKM_AES_KEY_GEN 0x00001080
#define CKM_AES_ECB 0x00001081
#define CKM_AES_CBC 0x00001082
#define CKM_AES_MAC 0x00001083
#define CKM_AES_MAC_GENERAL 0x00001084
#define CKM_AES_CBC_PAD 0x00001085
#define CKM_AES_CTR 0x00001086
#define CKM_AES_GCM 0x00001087
#define CKM_AES_CCM 0x00001088
#define CKM_AES_MAC_GENERAL 0x00001089
#define CKM_AES_CMAC 0x0000108A
#define CKM_AES_CTS 0x0000108B
#define CKM_AES_XCBC_MAC 0x0000108C
#define CKM_AES_XCBC_MAC_96 0x0000108D
#define CKM_AES_GMAC 0x0000108E
#define CKM_BLOWFISH_KEY_GEN 0x00001090
#define CKM_BLOWFISH_CBC 0x00001091
#define CKM_TWOFISH_KEY_GEN 0x00001092
#define CKM_TWOFISH_CBC 0x00001093
#define CKM_TWOFISH_CBC_PAD 0x00001094
#define CKM_DES_ECB_ENCRYPT_DATA 0x00001100
#define CKM_DES_CBC_ENCRYPT_DATA 0x00001101
#define CKM_DES3_ECB_ENCRYPT_DATA 0x00001102
#define CKM_DES3_CBC_ENCRYPT_DATA 0x00001103
#define CKM_AES_ECB_ENCRYPT_DATA 0x00001104
#define CKM_AES_CBC_ENCRYPT_DATA 0x00001105
#define CKM_GOSTR3410_KEY_PAIR_GEN 0x00001200
#define CKM_GOSTR3410 0x00001201
#define CKM_GOSTR3410_WITH_GOSTR3411 0x00001202
#define CKM_GOSTR3410_KEY_WRAP 0x00001203
#define CKM_GOSTR3410_DERIVE 0x00001204
#define CKM_GOSTR3411 0x00001205
#define CKM_GOSTR3411_HMAC 0x00001206
#define CKM_GOST28147_KEY_GEN 0x00001210
#define CKM_GOST28147 0x00001211
#define CKM_GOST28147_MAC 0x00001212
#define CKM_GOST28147_KEY_WRAP 0x00001213
#define CKM_GOST28147_KEY_WRAP 0x00001214
#define CKM_CHACHA20_KEY_GEN 0x00001220
#define CKM_CHACHA20 0x00001221
#define CKM_X9_42_DH_PKCS_PARAMETER_GEN 0x00001230
#define CKM_X9_42_DH_PKCS_PARAMETER_GEN 0x00001231
#define CKM_POLY1305_KEY_GEN 0x00001232
#define CKM_POLY1305 0x00001233
#define CKM_DSA_PARAMETER_GEN 0x00001234
#define CKM_DH_PKCS_PARAMETER_GEN 0x00001235
#define CKM_X9_42_DH_PKCS_PARAMETER_GEN 0x00001236

#define CKM_DSA_PROBABLISTIC_PARAMETER_GEN 0x00002003
#define CKM_DSA_SHAWE_TAYLOR_PARAMETER_GEN 0x00002004
#define CKM_AES_OFB 0x00002104
#define CKM_AES_CFB64 0x00002105
#define CKM_AES_CFB8 0x00002106
#define CKM_AES_CFB128 0x00002107
#define CKM_AES_CFB1 0x00002108
#define CKM_AES_KEY_WRAP 0x00002109
#define CKM_AES_KEY_WRAP_PAD 0x0000210A
#define CKM_AES_KEY_WRAP_KWP 0x0000210B
#define CKM_RSA_PKCS_TPM_1_1 0x00004001
#define CKM_RSA_PKCS_OAEP_TPM_1_1 0x00004002
#define CKM_SHA_1_KEY_GEN 0x00004003
#define CKM_SHA224_KEY_GEN 0x00004004
#define CKM_SHA256_KEY_GEN 0x00004005
#define CKM_SHA384_KEY_GEN 0x00004006
#define CKM_SHA512_KEY_GEN 0x00004007
#define CKM_SHA512_224_KEY_GEN 0x00004008
#define CKM_SHA512_256_KEY_GEN 0x00004009
#define CKM_SHA512_256_KEY_GEN 0x0000400A
#define CKM_SHA512_256_KEY_GEN 0x0000400B
#define CKM_SHA224_KEY_GEN 0x0000400C
#define CKM_SHA256_KEY_GEN 0x0000400D
#define CKM_SHA384_KEY_GEN 0x0000400E
#define CKM_SHA512_256_KEY_GEN 0x0000400F
#define CKM_SHA512_KEY_GEN 0x00004010
#define CKM_SHA512_256_KEY_GEN 0x00004011
#define CKM_SHA512_256_KEY_GEN 0x00004012
#define CKM_SHA512_256_KEY_GEN 0x00004013
#define CKM_SHA512_256_KEY_GEN 0x00004014
#define CKM_SHA512_256_KEY_GEN 0x00004015
#define CKM_SHA512_256_KEY_GEN 0x00004016
#define CKM_SHA512_256_KEY_GEN 0x00004017
#define CKM_SHA512_256_KEY_GEN 0x00004018
#define CKM_SHA512_256_KEY_GEN 0x00004019
#define CKM_SHA512_256_KEY_GEN 0x0000401A
#define CKM_SHA512_256_KEY_GEN 0x0000401B
#define CKM_SHA512_256_KEY_GEN 0x0000401C
#define CKM_SHA512_256_KEY_GEN 0x0000401D
#define CKM_SHA512_256_KEY_GEN 0x0000401E
#define CKM_SHA512_256_KEY_GEN 0x0000401F
#define CKM_SHA512_256_KEY_GEN 0x00004020
#define CKM_SHA512_256_KEY_GEN 0x00004021
#define CKM_SHA512_256_KEY_GEN 0x00004022
#define CKM_SHA512_256_KEY_GEN 0x00004023
#define CKM_SHA512_256_KEY_GEN 0x00004024
#define CKM_SHA512_256_KEY_GEN 0x00004025
#define CKM_SHA512_256_KEY_GEN 0x00004026
### B.5 Attributes

<table>
<thead>
<tr>
<th>Line</th>
<th>Definition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>8816</td>
<td>#define CKA_CLASS</td>
<td>0x000000000</td>
</tr>
<tr>
<td>8817</td>
<td>#define CKA_TOKEN</td>
<td>0x000000001</td>
</tr>
<tr>
<td>8818</td>
<td>#define CKA_PRIVATE</td>
<td>0x000000002</td>
</tr>
<tr>
<td>8819</td>
<td>#define CKA_LABEL</td>
<td>0x000000003</td>
</tr>
<tr>
<td>8820</td>
<td>#define CKA_UNIQUE_ID</td>
<td>0x000000004</td>
</tr>
<tr>
<td>8821</td>
<td>#define CKA_APPLICATION</td>
<td>0x000000010</td>
</tr>
<tr>
<td>8822</td>
<td>#define CKA_VALUE</td>
<td>0x000000011</td>
</tr>
<tr>
<td>8823</td>
<td>#define CKA_OBJECT_ID</td>
<td>0x000000012</td>
</tr>
<tr>
<td>8824</td>
<td>#define CKA_CERTIFICATE_TYPE</td>
<td>0x000000080</td>
</tr>
<tr>
<td>8825</td>
<td>#define CKA_ISSUER</td>
<td>0x000000081</td>
</tr>
<tr>
<td>8826</td>
<td>#define CKA_SERIAL_NUMBER</td>
<td>0x000000082</td>
</tr>
<tr>
<td>8827</td>
<td>#define CKA_AC_ISSUER</td>
<td>0x000000083</td>
</tr>
<tr>
<td>8828</td>
<td>#define CKA_OWNER</td>
<td>0x000000084</td>
</tr>
<tr>
<td>8829</td>
<td>#define CKA_ATTR_TYPES</td>
<td>0x000000085</td>
</tr>
<tr>
<td>8830</td>
<td>#define CKA_TRUSTED</td>
<td>0x000000086</td>
</tr>
<tr>
<td>8831</td>
<td>#define CKA_CERTIFICATE_CATEGORY</td>
<td>0x000000087</td>
</tr>
<tr>
<td>8832</td>
<td>#define CKA_JAVA_MIDP_SECURITY_DOMAIN</td>
<td>0x000000088</td>
</tr>
<tr>
<td>8833</td>
<td>#define CKA_URL</td>
<td>0x000000089</td>
</tr>
<tr>
<td>8834</td>
<td>#define CKA_HASH_OF_SUBJECT_PUBLIC_KEY</td>
<td>0x00000008A</td>
</tr>
<tr>
<td>8835</td>
<td>#define CKA_HASH_OF_ISSUER_PUBLIC_KEY</td>
<td>0x00000008B</td>
</tr>
<tr>
<td>8836</td>
<td>#define CKA_NAME_HASH_ALGORITHM</td>
<td>0x00000008C</td>
</tr>
<tr>
<td>8837</td>
<td>#define CKA_CHECK_VALUE</td>
<td>0x000000090</td>
</tr>
<tr>
<td>8838</td>
<td>#define CKA_KEY_TYPE</td>
<td>0x000000100</td>
</tr>
<tr>
<td>8839</td>
<td>#define CKA_SUBJECT</td>
<td>0x000000101</td>
</tr>
<tr>
<td>8840</td>
<td>#define CKA_ID</td>
<td>0x000000102</td>
</tr>
<tr>
<td>8841</td>
<td>#define CKA_SENSITIVE</td>
<td>0x000000103</td>
</tr>
<tr>
<td>8842</td>
<td>#define CKA_ENCRYPT</td>
<td>0x000000104</td>
</tr>
<tr>
<td>8843</td>
<td>#define CKA_DECRYPT</td>
<td>0x000000105</td>
</tr>
<tr>
<td>8844</td>
<td>#define CKA_WRAP</td>
<td>0x000000106</td>
</tr>
<tr>
<td>8845</td>
<td>#define CKA_UNWRAP</td>
<td>0x000000107</td>
</tr>
<tr>
<td>8846</td>
<td>#define CKA_SIGN</td>
<td>0x000000108</td>
</tr>
<tr>
<td>8847</td>
<td>#define CKA_SIGN_RECOVER</td>
<td>0x000000109</td>
</tr>
<tr>
<td>8848</td>
<td>#define CKA_VERIFY</td>
<td>0x00000010A</td>
</tr>
<tr>
<td>8849</td>
<td>#define CKA_VERIFY_RECOVER</td>
<td>0x00000010B</td>
</tr>
<tr>
<td>8850</td>
<td>#define CKA_DERIVE</td>
<td>0x00000010C</td>
</tr>
<tr>
<td>8851</td>
<td>#define CKA_START_DATE</td>
<td>0x000000110</td>
</tr>
<tr>
<td>8852</td>
<td>#define CKA_END_DATE</td>
<td>0x000000111</td>
</tr>
<tr>
<td>8853</td>
<td>#define CKA_MODULUS</td>
<td>0x000000120</td>
</tr>
<tr>
<td>8854</td>
<td>#define CKA_MODULUS_BITS</td>
<td>0x000000121</td>
</tr>
</tbody>
</table>
#define CKA_PUBLIC_EXPONENT 0x00000122
#define CKA_PRIVATE_EXPONENT 0x00000123
#define CKA_PRIME_1 0x00000124
#define CKA_PRIME_2 0x00000125
#define CKA_EXPONENT_1 0x00000126
#define CKA_EXPONENT_2 0x00000127
#define CKA_COEFFICIENT 0x00000128
#define CKA_PUBLIC_KEY_INFO 0x00000129
#define CKA_PRIME 0x00000130
#define CKA_SUBPRIME 0x00000131
#define CKA_BASE 0x00000132
#define CKA_PRIME_BITS 0x00000133
#define CKA_SUB_PRIME_BITS CKA_SUBPRIME_BITS
#define CKA_VALUE_BITS 0x00000160
#define CKA_VALUE_LEN 0x00000161
#define CKA_EXTRACTABLE 0x00000162
#define CKA_LOCAL 0x00000163
#define CKA_NEVER_EXTRACTABLE 0x00000164
#define CKA_ALWAYS_SENSITIVE 0x00000165
#define CKA_KEY_GEN_MECHANISM 0x00000166
#define CKA_MODIFIABLE 0x00000170
#define CKA_COPYABLE 0x00000171
#define CKA_DESTROYABLE 0x00000172
#define CKA_EC_PARAMS 0x00000180
#define CKA_EC_POINT 0x00000181
#define CKA_WRAP_WITH_TRUSTED 0x00000210
#define CKA_WRAP_TEMPLATE (CKF_ARRAY_ATTRIBUTE|0x00000211)
#define CKA_UNWRAP_TEMPLATE (CKF_ARRAY_ATTRIBUTE|0x00000212)
#define CKA_DERIVE_TEMPLATE (CKF_ARRAY_ATTRIBUTE|0x00000213)
#define CKA_OTP_FORMAT 0x00000220
#define CKA_OTP_LENGTH 0x00000221
#define CKA_OTP_TIME_INTERVAL 0x00000222
#define CKA_OTP_USER_FRIENDLY_MODE 0x00000223
#define CKA_OTP_CHALLENGE_REQUIREMENT 0x00000224
#define CKA_OTP_TIME_REQUIREMENT 0x00000225
#define CKA_OTP_COUNTER_REQUIREMENT 0x00000226
#define CKA_OTP_PIN_REQUIREMENT 0x00000227
#define CKA_OTP_USER_IDENTIFIER 0x0000022A
#define CKA_OTP_SERVICE_IDENTIFIER 0x0000022B
#define CKA_OTP_SERVICE_LOGO 0x0000022C
#define CKA_OTP_SERVICE_LOGO_TYPE 0x0000022D
#define CKA_OTP_COUNTER 0x0000022E
#define CKA_OTP_TIME 0x0000022F
#define CKA_GOSTR3410_PARAMS 0x00000250
#define CKA_GOSTR3411_PARAMS 0x00000251
#define CKA_GOST28147_PARAMS 0x00000252
#define CKA_HW_FEATURE_TYPE 0x00000300
8903  #define CKA_RESET_ON_INIT          0x00000301
8904  #define CKA_HAS_RESET            0x00000302
8905  #define CKA_PIXEL_X              0x00000400
8906  #define CKA_PIXEL_Y              0x00000401
8907  #define CKA_RESOLUTION           0x00000402
8908  #define CKA_CHAR_ROWS            0x00000403
8909  #define CKA_CHAR_COLUMNS         0x00000404
8910  #define CKA_COLOR                0x00000405
8911  #define CKA_BITS_PER_PIXEL       0x00000406
8912  #define CKA_PIXELSETS            0x00000480
8913  #define CKA_ENCODING_METHODS      0x00000481
8914  #define CKA_MIME_TYPES           0x00000482
8915  #define CKA_MECHANISM_TYPE        0x00000500
8916  #define CKA_REQUIRED_CMS_ATTRIBUTES 0x00000501
8917  #define CKA_DEFAULT_CMS_ATTRIBUTES 0x00000502
8918  #define CKA_SUPPORTED_CMS_ATTRIBUTES 0x00000503
8919  #define CKA_ALLOWED_MECHANISMS     (CKF_ARRAY_ATTRIBUTE|0x00000600)
8920  #define CKA_VENDOR_DEFINED        0x80000000

8922  B.6 Attribute constants
8923  #define CK_OTP_FORMAT_DECIMAL      0x00000000
8924  #define CK_OTP_FORMAT_HEXADECIMAL   0x00000001
8925  #define CK_OTP_FORMAT_ALPHANUMERIC  0x00000002
8926  #define CK_OTP_FORMAT_BINARY        0x00000003
8927  #define CK_OTP_PARAM_IGNORED        0x00000000
8928  #define CK_OTP_PARAM_OPTIONAL       0x00000001
8929  #define CK_OTP_PARAM_MANDATORY      0x00000002
8930  #define CK_OTP_VALUE                0x00000000
8931  #define CK_OTP_PIN                  0x00000001
8932  #define CK_OTP_CHALLENGE           0x00000002
8933  #define CK_OTP_TIME                 0x00000003
8934  #define CK_OTP_COUNTER              0x00000004
8935  #define CK_OTP_FLAGS                0x00000005
8936  #define CK_OTP_OUTPUT_LENGTH        0x00000006
8937  #define CK_OTP_FORMAT               0x00000007

8939  B.7 Other constants
8940  #define CKF_NEXT_OTP               0x00000001
8941  #define CKF_EXCLUDE_TIME           0x00000002
8942  #define CKF_EXCLUDE_COUNTER        0x00000004
8943  #define CKF_EXCLUDE_CHALLENGE      0x00000008
8944  #define CKF_EXCLUDE_PIN            0x00000010
8945  #define CKF_USER_FRIENDLY_OTP      0x00000020
#define CKF_HKDF_SALT_NULL 0x00000001
#define CKF_HKDF_SALT_DATA 0x00000002
#define CKF_HKDF_SALT_KEY 0x00000004

## B.8 Notifications

#define CKN_OTP_CHANGED 0x00000001

## B.9 Return values

#define CKR_OK 0x00000000
#define CKR_CANCEL 0x00000001
#define CKR_HOST_MEMORY 0x00000002
#define CKR_SLOT_ID_INVALID 0x00000003
#define CKR_GENERAL_ERROR 0x00000005
#define CKR_FUNCTION_FAILED 0x00000006
#define CKR_ARGUMENTS_BAD 0x00000007
#define CKR_NO_EVENT 0x00000008
#define CKR_NEED_TO_CREATE_THREADS 0x00000009
#define CKR_CANT_LOCK 0x0000000A
#define CKR_ATTRIBUTE_READ_ONLY 0x00000010
#define CKR_ATTRIBUTE_SENSITIVE 0x00000011
#define CKR_ATTRIBUTE_TYPE_INVALID 0x00000012
#define CKR_ATTRIBUTE_VALUE_INVALID 0x00000013
#define CKR_ACTION_PROHIBITED 0x0000001B
#define CKR_DATA_INVALID 0x00000020
#define CKR_DATA_LEN_RANGE 0x00000021
#define CKR_DEVICE_ERROR 0x00000030
#define CKR_DEVICE_MEMORY 0x00000031
#define CKR_DEVICE_REMOVED 0x00000032
#define CKR_ENCRYPTED_DATA_INVALID 0x00000040
#define CKR_ENCRYPTED_DATA_LEN_RANGE 0x00000041
#define CKR_AEAD_DECRYPT_FAILED 0x00000042
#define CKR_FUNCTION_CANCELED 0x00000050
#define CKR_FUNCTION_NOT_PARALLEL 0x00000051
#define CKR_FUNCTION_NOT_SUPPORTED 0x00000054
#define CKR_KEY_HANDLE_INVALID 0x00000060
#define CKR_KEY_SIZE_RANGE 0x00000062
#define CKR_KEY_TYPE_INCONSISTENT 0x00000063
#define CKR_KEY_NOT_NEEDED 0x00000064
#define CKR_KEY_CHANGED 0x00000065
#define CKR_KEY_NEEDED 0x00000066
#define CKR_KEY_INDIGESTIBLE 0x00000067
#define CKR_KEY_FUNCTION_NOT_PERMITTED 0x00000068
#define CKR_KEY_NOT_WRAPPABLE 0x00000069
#define CKR_MECHANISM_INVALID 0x00000070
#define CKR_MECHANISM_PARAM_INVALID 0x00000071
#define CKR_OBJECT_HANDLE_INVALID 0x00000082
#define CKR_OPERATION_ACTIVE          0x00000090
#define CKR_OPERATION_NOT_INITIALIZED 0x00000091
#define CKR_PIN_INCORRECT            0x000000A0
#define CKR_PIN_INVALID              0x000000A1
#define CKR_PIN_LEN_RANGE            0x000000A2
#define CKR_PIN_EXPIRED              0x000000A3
#define CKR_PIN_LOCKED               0x000000A4
#define CKR_SESSION_CLOSED           0x000000B0
#define CKR_SESSION_COUNT            0x000000B1
#define CKR_SESSION_HANDLE_INVALID   0x000000B3
#define CKR_SESSION_PARALLEL_NOT_SUPPORTED 0x000000B4
#define CKR_SESSION_READ_ONLY        0x000000B5
#define CKR_SESSION_EXISTS           0x000000B6
#define CKR_SESSION_READ_ONLY_EXISTS 0x000000B7
#define CKR_SESSION_READ_WRITE_SO_EXISTS 0x000000B8
#define CKR_SIGNATURE_INVALID        0x000000C0
#define CKR_SIGNATURE_LEN_RANGE      0x000000C1
#define CKR_TEMPLATE_INCOMPLETE      0x000000D0
#define CKR_TEMPLATE_INCONSISTENT    0x000000D1
#define CKR_TOKEN_NOT_PRESENT        0x000000E0
#define CKR_TOKEN_NOT_RECOGNIZED     0x000000E1
#define CKR_TOKEN_WRITE_PROTECTED    0x000000E2
#define CKR_UNWRAPPING_KEY_INVALID   0x000000F0
#define CKR_UNWRAPPING_KEY_SIZE_RANGE 0x000000F1
#define CKR_UNWRAPPING_KEY_TYPE_INCONSISTENT 0x000000F2
#define CKR_USER_ALREADY_LOGGED_IN    0x00000100
#define CKR_USER_NOT_LOGGED_IN       0x00000101
#define CKR_USER_PIN_NOT_INITIALIZED 0x00000102
#define CKR_USER_TYPE_INVALID        0x00000103
#define CKR_USER_ANOTHER_ALREADY_LOGGED_IN 0x00000104
#define CKR_USER_TOO_MANY_TYPES      0x00000105
#define CKR_WRAPPED_KEY_INVALID      0x00000110
#define CKR_WRAPPED_KEY_LEN_RANGE    0x00000112
#define CKR_WRAPPING_KEY_HANDLE_INVALID 0x00000113
#define CKR_WRAPPING_KEY_SIZE_RANGE  0x00000114
#define CKR_WRAPPING_KEY_TYPE_INCONSISTENT 0x00000115
#define CKR_RANDOM_SEED_NOT_SUPPORTED 0x00000120
#define CKR_RANDOM_NO_RNG            0x00000121
#define CKR_DOMAIN_PARAMS_INVALID    0x00000130
#define CKR_CURVE_NOT_SUPPORTED      0x00000140
#define CKR_BUFFER_TOO_SMALL         0x00000150
#define CKR_SAVED_STATE_INVALID      0x00000160
#define CKR_INFORMATION_SENSITIVE    0x00000170
#define CKR_STATE_UNSAVEABLE        0x00000180
#define CKR_CRYPTOKI_NOT_INITIALIZED 0x00000190
#define CKR_CRYPTOKI_ALREADY_INITIALIZED 0x00000191
#define CKR_MUTEX_BAD                0x000001A0
#define CKR_MUTEX_NOT_LOCKED         0x000001A1
#define CKR_NEW_PIN_MODE       0x000001B0
#define CKR_NEXT_OTP           0x000001B1
#define CKR_EXCEEDED_MAX_ITERATIONS 0x000001B5
#define CKR_FIPS_SELF_TEST_FAILED 0x000001B6
#define CKR_LIBRARY_LOAD_FAILED 0x000001B7
#define CKR_PIN_TOO_WEAK        0x000001B8
#define CKR_PUBLIC_KEY_INVALID  0x000001B9
#define CKR_FUNCTION_REJECTED   0x00000200
#define CKR_VENDOR_DEFINED      0x80000000
## Appendix C. Revision History

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Editor</th>
<th>Changes Made</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wd02</td>
<td>18-Jun-2017</td>
<td>Chris Zimman</td>
<td>Initial version incorporating recent changes since 2.41</td>
</tr>
<tr>
<td>WD07</td>
<td>16-Oct-2018</td>
<td>Dieter Bong</td>
<td>See pkcs11-curr-v3.0-updates-from-wd06-to-wd07.docx</td>
</tr>
<tr>
<td>WD08</td>
<td>22-Oct-2018</td>
<td>Dieter Bong</td>
<td>Added references [SALSA] to section 1.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Split GOST Mechanisms vs. Functions table for GOST into separate tables for the respective GOST nnn algorithms.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Changed format of Salsa20 to Heading2, making it section 2.59</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Removed section B.1 OTP definitions (OTP definitions have become part of the standard header files and are now covered in B.6 Attribute constants)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Copyright updated from “2013” to “2018”</td>
</tr>
<tr>
<td>WD09</td>
<td>26-Mar-2019</td>
<td>Daniel Minder</td>
<td>Added CKF_EC_CURVENAME to table 34</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Changed CK_GCM_AED_Params to CK_GCM_MESSAGE_PARAMS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Reworked section 2.13 (additional AES mechanisms) during F2F</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Removed Derive for CKM_AES_GMAC in Table 80 since this is not defined in section 2.13.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Removed solved comments of Chris typos and formatting</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Edwards curves and RFC 8410:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Added reference to RFC 8410 at several places in 2.3.5 - 2.3.14</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Clarified that Edwards/Montgomery curves specified with curveName are incompatible with curves specified with old (since RFC 8410 is designed like this)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Added explanation for CKM_TOKENRESOURCEEXCEEDED error in 2.3.14</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Changed sample template for edwards public key objects in 2.3.5 since the parameter spec was in ecPoint instead of ecParams</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Added “allowed key types” table in 2.3.17 - 2.3.20 (they were lost when copying from the proposal), but corrected it for ECDH with cofactor since this is not possible according to RFC8032.</td>
</tr>
<tr>
<td>Version</td>
<td>Date</td>
<td>Author</td>
<td>Changes</td>
</tr>
<tr>
<td>-----------</td>
<td>------------</td>
<td>-------------</td>
<td>---------</td>
</tr>
</tbody>
</table>
| WD10      | 29 Apr 2019| Dieter Bong | - Updated section Related work  
- Added Dieter Bong as Editor  
- Put year 2019 in Copyright  
- Added section 2.62 HKDF Mechanisms; HKDF constants in Appendix B; RFC5869 in section 1.4 Non-Normative References  
- Section 2.40: added CKM_NULL, removed CKM_TLS10_MAC_*  
- Section 2.52.7.1 reference to base specification corrected  
- Replaced reference to [PKCS11-Base] table 10 by [PKCS11-Base] table 11 throughout whole document  
- Removed all occurrences of CKK_ECDSA and CKA_ECDSA_PARAMS and added notices that they are deprecated  
- Removed #define's for CKA_SECONDARY_AUTH, CKA_AUTH_PIN_FLAGS and CKA_ALWAYS_AUTHENTICATE  
- Removed #define's for CAST5 mechanisms |
| WD10 Rev. 2| 7 May 2019  | Dieter Bong | - Moved CKM_NULL to own section 2.63  
- Removed 2 remaining occurrences of CKA_ECDSA_PARAMS |
| WD11      | May 28, 2019| Tony Cox    | - Final cleanup of front introductory texts and links prior to CSPRD |