PKCS #11 Cryptographic Token Interface
Historical Mechanisms Specification
Version 3.0

Committee Specification 01

19 December 2019

This stage:
https://docs.oasis-open.org/pkcs11/pkcs11-hist/v3.0/cs01/pkcs11-hist-v3.0-cs01.docx (Authoritative)
https://docs.oasis-open.org/pkcs11/pkcs11-hist/v3.0/cs01/pkcs11-hist-v3.0-cs01.html
https://docs.oasis-open.org/pkcs11/pkcs11-hist/v3.0/cs01/pkcs11-hist-v3.0-cs01.pdf

Previous stage:
N/A

Latest stage:
https://docs.oasis-open.org/pkcs11/pkcs11-hist/v3.0/pkcs11-hist-v3.0.docx (Authoritative)
https://docs.oasis-open.org/pkcs11/pkcs11-hist/v3.0/pkcs11-hist-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-hist/v3.0/cs01/include/pkcs11-v3.0/

Related work:
This specification replaces or supersedes:

This specification is related to:
Abstract:
This document defines mechanisms for PKCS #11 that are no longer in general use.

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

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

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

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

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

[PKCS11-Historical-v3.0]
## Table of Contents

1 Introduction ......................................................................................................................... 8
  1.1 IPR Policy ......................................................................................................................... 8
  1.2 Description of this Document .......................................................................................... 8
  1.3 Terminology ..................................................................................................................... 8
  1.4 Definitions ....................................................................................................................... 8
  1.5 Normative References ...................................................................................................... 9
  1.6 Non-Normative References .............................................................................................. 9
2 Mechanisms .......................................................................................................................... 12
  2.1 PKCS #11 Mechanisms .................................................................................................... 12
  2.2 FORTEZZA timestamp ................................................................................................... 14
  2.3 KEA .................................................................................................................................. 15
    2.3.1 Definitions .................................................................................................................. 15
    2.3.2 KEA mechanism parameters ...................................................................................... 15
          2.3.2.1 CK KEA_DERIVE_PARAMS; CK KEA_DERIVE_PARAMS_PTR ................. 15
    2.3.3 KEA public key objects .............................................................................................. 16
    2.3.4 KEA private key objects ............................................................................................ 16
    2.3.5 KEA key pair generation ............................................................................................ 17
    2.3.6 KEA key derivation ..................................................................................................... 18
  2.4 RC2 .................................................................................................................................... 19
    2.4.1 Definitions ................................................................................................................. 19
    2.4.2 RC2 secret key objects ............................................................................................... 19
    2.4.3 RC2 mechanism parameters ...................................................................................... 20
          2.4.3.1 CK RC2_PARAMS; CK RC2_PARAMS_PTR .............................................. 20
          2.4.3.2 CK RC2_CBC_PARAMS; CK RC2_CBC_PARAMS_PTR .................................. 20
          2.4.3.3 CK RC2_MAC_GENERAL_PARAMS; CK RC2_MAC_GENERAL_PARAMS_PTR  20
    2.4.4 RC2 key generation ..................................................................................................... 21
    2.4.5 RC2-ECB .................................................................................................................... 21
    2.4.6 RC2-CBC .................................................................................................................... 22
    2.4.7 RC2-CBC with PKCS padding .................................................................................... 22
    2.4.8 General-length RC2-MAC .......................................................................................... 23
    2.4.9 RC2-MAC .................................................................................................................... 23
  2.5 RC4 ..................................................................................................................................... 24
    2.5.1 Definitions ................................................................................................................. 24
    2.5.2 RC4 secret key objects ............................................................................................... 24
    2.5.3 RC4 key generation ..................................................................................................... 24
    2.5.4 RC4 mechanism ......................................................................................................... 25
  2.6 RC5 ..................................................................................................................................... 25
    2.6.1 Definitions ................................................................................................................. 25
    2.6.2 RC5 secret key objects ............................................................................................... 25
    2.6.3 RC5 mechanism parameters ...................................................................................... 26
          2.6.3.1 CK RC5_PARAMS; CK RC5_PARAMS_PTR .............................................. 26
          2.6.3.2 CK RC5_CBC_PARAMS; CK RC5_CBC_PARAMS_PTR .................................. 26
          2.6.3.3 CK RC5_MAC_GENERAL_PARAMS; CK RC5_MAC_GENERAL_PARAMS_PTR  26
    2.6.4 RC5 key generation ..................................................................................................... 27
2.6.5 RC5-ECB .......................................................... 27
2.6.6 RC5-CBC .......................................................... 28
2.6.7 RC5-CBC with PKCS padding ................................. 28
2.6.8 General-length RC5-MAC ...................................... 29
2.6.9 RC5-MAC .......................................................... 29

2.7 General block cipher ................................................ 30
  2.7.1 Definitions ....................................................... 30
  2.7.2 DES secret key objects ...................................... 31
  2.7.3 CAST secret key objects .................................... 31
  2.7.4 CAST3 secret key objects .................................. 32
  2.7.5 CAST128 secret key objects ................................ 32
  2.7.6 IDEA secret key objects ..................................... 33
  2.7.7 CDMF secret key objects .................................... 33
  2.7.8 General block cipher mechanism parameters .......... 34
     2.7.8.1 CK_MAC_GENERAL_PARAMS; CK_MAC_GENERAL_PARAMS_PTR ............... 34
  2.7.9 General block cipher key generation .................... 34
  2.7.10 General block cipher ECB ................................ 35
  2.7.11 General block cipher CBC ................................ 35
  2.7.12 General block cipher CBC with PKCS padding ......... 36
  2.7.13 General-length general block cipher MAC .................. 36
  2.7.14 General block cipher MAC ................................ 37

2.8 SKIPJACK .......................................................... 37
  2.8.1 Definitions ....................................................... 37
  2.8.2 SKIPJACK secret key objects ................................ 38
  2.8.3 SKIPJACK Mechanism parameters .......................... 39
     2.8.3.1 CK_SKIPJACK_PRIVATE_WRAP_PARAMS; CK_SKIPJACK_PRIVATE_WRAP_PARAMS_PTR .. 39
     2.8.3.2 CK_SKIPJACK_RELAYX_PARAMS; CK_SKIPJACK_RELAYX_PARAMS_PTR .................. 39
  2.8.4 SKIPJACK key generation ................................... 40
  2.8.5 SKIPJACK-ECB64 .............................................. 41
  2.8.6 SKIPJACK-CBC64 ............................................. 41
  2.8.7 SKIPJACK-OFB64 ............................................. 41
  2.8.8 SKIPJACK-CFB64 ............................................. 41
  2.8.9 SKIPJACK-CFB32 ............................................. 42
  2.8.10 SKIPJACK-CFB16 .......................................... 42
  2.8.11 SKIPJACK-CFB8 ............................................. 42
  2.8.12 SKIPJACK-WRAP ............................................ 43
  2.8.13 SKIPJACK-PRIVATE-WRAP .................................. 43
  2.8.14 SKIPJACK-RELAYX ........................................ 43

2.9 BATON ............................................................ 43
  2.9.1 Definitions ....................................................... 43
  2.9.2 BATON secret key objects .................................. 43
  2.9.3 BATON key generation ....................................... 44
  2.9.4 BATON-ECB128 .............................................. 44
  2.9.5 BATON-ECB96 ................................................ 45
  2.9.6 BATON-CBC128 .............................................. 45
  2.9.7 BATON-COUNTER ........................................... 45
2.16.4 RIPE-MD 128-HMAC .................................................................57
2.16.5 RIPE-MD 160 .................................................................57
2.16.6 General-length RIPE-MD 160-HMAC ........................................57
2.16.7 RIPE-MD 160-HMAC ............................................................58
2.17 SET ..........................................................................................58
  2.17.1 Definitions .........................................................................58
  2.17.2 SET mechanism parameters ............................................58
    2.17.2.1 CK_KEY_WRAP_SET_OAEP_PARAMS; CK_KEY_WRAP_SET_OAEP_PARAMS_PTR ..58
  2.17.3 OAEP key wrapping for SET ..................................................58
2.18 LYNKS .......................................................................................59
  2.18.1 Definitions .........................................................................59
  2.18.2 LYNKS key wrapping ..........................................................59
3 PKCS #11 Implementation Conformance ..................................................60
Appendix A. Acknowledgments .................................................................61
Appendix B. Manifest constants .................................................................63
Appendix C. Revision History ..................................................................64
1 Introduction

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 Description of this Document
This document defines historical PKCS#11 mechanisms, that is, mechanisms that were defined for earlier versions of PKCS #11 but are no longer in general use.
All text is normative unless otherwise labeled.

1.3 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].

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

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BATON</td>
<td>MISSI's BATON block cipher.</td>
</tr>
<tr>
<td>CAST</td>
<td>Entrust Technologies’ proprietary symmetric block cipher</td>
</tr>
<tr>
<td>CAST3</td>
<td>Entrust Technologies’ proprietary symmetric block cipher</td>
</tr>
<tr>
<td>CAST128</td>
<td>Entrust Technologies’ symmetric block cipher.</td>
</tr>
<tr>
<td>CDMF</td>
<td>Commercial Data Masking Facility, a block encipherment method specified by International Business Machines Corporation and based on DES.</td>
</tr>
<tr>
<td>CMS</td>
<td>Cryptographic Message Syntax (see RFC 3369)</td>
</tr>
<tr>
<td>DES</td>
<td>Data Encryption Standard, as defined in FIPS PUB 46-3</td>
</tr>
<tr>
<td>ECB</td>
<td>Electronic Codebook mode, as defined in FIPS PUB 81.</td>
</tr>
<tr>
<td>FASTHASH</td>
<td>MISSI’s FASTHASH message-digesting algorithm.</td>
</tr>
<tr>
<td>IDEA</td>
<td>Ascom Systec’s symmetric block cipher.</td>
</tr>
<tr>
<td>IV</td>
<td>Initialization Vector.</td>
</tr>
<tr>
<td>JUNIPER</td>
<td>MISSI’s JUNIPER block cipher.</td>
</tr>
<tr>
<td>KEA</td>
<td>MISSI’s Key Exchange Algorithm.</td>
</tr>
<tr>
<td>LYNKS</td>
<td>A smart card manufactured by SPYRUS.</td>
</tr>
<tr>
<td>MAC</td>
<td>Message Authentication Code</td>
</tr>
<tr>
<td>MD2</td>
<td>RSA Security’s MD2 message-digest algorithm, as defined in RFC 6149.</td>
</tr>
</tbody>
</table>
1.5 Normative References


1.6 Non-Normative References


## 2 Mechanisms

### 2.1 PKCS #11 Mechanisms

A mechanism specifies precisely how a certain cryptographic process is to be performed. PKCS #11 implementations MAY use one or 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 support only a subset of the mechanisms listed. There is also no guarantee that a token which supports one mechanism for some operation 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 MAY also perform RSA encryption with **CKM_RSA_PKCS**.

*Table 1, 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_FORTEZZA_TIMESTAMP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_KEA_KEY_PAIR_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CKM_KEA_KEY_DERIVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CKM_RC2_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CKM_RC2_ECB</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CKM_RC2_CBC</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CKM_RC2_CBC_PAD</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CKM_RC2_MAC_GENERAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CKM_RC2_MAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CKM_RC4_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CKM_RC4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CKM_RC5_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CKM_RC5_ECB</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CKM_RC5_CBC</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CKM_RC5_CBC_PAD</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CKM_RC5_MAC_GENERAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CKM_RC5_MAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CKM_DES_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CKM_DES_ECB</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CKM_DES_CBC</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CKM_DES_CBC_PAD</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CKM_DES_MAC_GENERAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CKM_DES_MAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CKM_CAST_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CKM_CAST_ECB</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CKM_CAST_CBC</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CKM_CAST_CBC_PAD</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CKM_CAST_MAC_GENERAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CKM_CAST_MAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Mechanism</td>
<td>Functions</td>
<td>Encrypt &amp; Decrypt</td>
<td>Sign &amp; Verify</td>
<td>SR &amp; VR¹</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>
<td>---------------</td>
</tr>
<tr>
<td>CKM CAST3 KEY GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM CAST3 ECB</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM CAST3 CBC</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM CAST3 CBC_PAD</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM CAST3 MAC_GENERAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM CAST3 MAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM CAST128 KEY GEN</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM CAST128 ECB</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM CAST128 CBC</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM CAST128_CBC_PAD</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM CAST128 MAC_GENERAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM CAST128 MAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM IDEA KEY GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM IDEA ECB</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM IDEA CBC</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM IDEA_CBC_PAD</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM IDEA_MAC_GENERAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM IDEA MAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM CDMF KEY GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM CDMF ECB</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM CDMF_CBC</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM CDMF_CBC_PAD</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM CDMF MAC_GENERAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM CDMF MAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM SKIPJACK KEY GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM SKIPJACK ECB64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM SKIPJACK_CBC64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM SKIPJACK_OFB64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM SKIPJACK CFB64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM SKIPJACK_CFB32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM SKIPJACK CFB16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM SKIPJACK CFB8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM SKIPJACK WRAP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM SKIPJACK_PRIVATE_WRAP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM SKIPJACK RELAYX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM BATON KEY GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM BATON ECB128</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM BATON ECB96</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM BATON_CBC128</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM BATON_COUNTER</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM BATON_SHUFFLE</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM BATON_WRAP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM JUNIPER KEY GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM JUNIPER ECB128</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM JUNIPER_CBC128</td>
<td></td>
<td>X</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&lt;sup&gt;1&lt;/sup&gt;</td>
<td>SR &amp; VR&lt;sup&gt;2&lt;/sup&gt;</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>
<td>---------------</td>
</tr>
<tr>
<td>CKM_JUNIPER_COUNTER</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_JUNIPER_SHUFFLE</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_JUNIPER_WRAP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_MD2</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_MD2_HMAC_GENERAL</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_MD2_HMAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_MD2_KEY_DERIVATION</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_MD5</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_MD5_HMAC_GENERAL</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_MD5_HMAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_MD5_KEY_DERIVATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_RIPEMD128</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_RIPEMD128_HMAC_GENERAL</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_RIPEMD128_HMAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_RIPEMD160</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_RIPEMD160_HMAC_GENERAL</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_RIPEMD160_HMAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_FASTHASH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_PBE_MD2_DES_CBC</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_PBE_MD5_DES_CBC</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_PBE_MD5_CAST_CBC</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_PBE_MD5_CAST3_CBC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_PBE_MD5_CAST128_CBC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_PBE_SHA1_CAST128_CBC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_PBE_SHA1_RC4_128</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_PBE_SHA1_RC4_40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_PBE_SHA1_RC2_128_CBC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_PBE_SHA1_RC2_40_CBC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_PBA_SHA1_WITH_SHA1_HMAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_KEY_WRAP_SET_OAEP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>CKM_KEY_WRAP_LYNKS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<sup>1</sup> SR = SignRecover, VR = VerifyRecover.
<sup>2</sup> Single-part operations only.
<sup>3</sup> Mechanism MUST only be used for wrapping, not unwrapping.

The remainder of this section presents 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.

### 2.2 FORTEZZA timestamp

The FORTEZZA timestamp mechanism, denoted **CKM_FORTEZZA_TIMESTAMP**, is a mechanism for single-part signatures and verification. The signatures it produces and verifies are DSA digital signatures over the provided hash value and the current time.
It has no parameters.

Constraints on key types and the length of data are summarized in the following table. The input and output data MAY begin at the same location in memory.

Table 2, FORTEZZA Timestamp: 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>20</td>
<td>40</td>
</tr>
<tr>
<td>C_Verify</td>
<td>DSA public key</td>
<td>20,40²</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.

2.3 KEA

2.3.1 Definitions

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

Mechanisms:

CKM_KEA_KEY_PAIR_GEN

CKM_KEA_KEY_DERIVE

2.3.2 KEA mechanism parameters

2.3.2.1 CK freshmen KEA_DERIVE_PARAMS; CK KEA_DERIVE_PARAMS_PTR

CK freshmen KEA_DERIVE_PARAMS is a structure that provides the parameters to the CKM freshmen KEA_DERIVE mechanism. It is defined as follows:

```c
typedef struct CK freshmen KEA_DERIVE_PARAMS {
    CK_BBOOL isSender;
    CK_ULONG ulRandomLen;
    CK_BYTE_PTR pRandomA;
    CK_BYTE_PTR pRandomB;
    CK_ULONG ulPublicDataLen;
    CK_BYTE_PTR pPublicData;
} CK freshmen KEA_DERIVE_PARAMS;
```

The fields of the structure have the following meanings:

- **isSender**: Option for generating the key (called a TEK). The value is CK_TRUE if the sender (originator) generates the TEK, CK_FALSE if the recipient is regenerating the TEK.
- **ulRandomLen**: the size of random Ra and Rb in bytes
- **pRandomA**: pointer to Ra data
- **pRandomB**: pointer to Rb data
- **ulPublicDataLen**: other party’s KEA public key size
$p_{PublicData}$ pointer to other party’s KEA public key value

$\text{CK\_KEA\_DERIVE\_PARAMS\_PTR}$ is a pointer to a $\text{CK\_KEA\_DERIVE\_PARAMS}$.

### 2.3.3 KEA public key objects

KEA public key objects (object class $\text{CKO\_PUBLIC\_KEY}$, key type $\text{CKK\_KEA}$) hold KEA public keys.

The following table defines the KEA 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>$\text{CKA_PRIME}^{1,3}$</td>
<td>Big integer</td>
<td>Prime $p$ (512 to 1024 bits, in steps of 64 bits)</td>
</tr>
<tr>
<td>$\text{CKA_SUBPRIME}^{1,3}$</td>
<td>Big integer</td>
<td>Subprime $q$ (160 bits)</td>
</tr>
<tr>
<td>$\text{CKA_BASE}^{1,3}$</td>
<td>Big integer</td>
<td>Base $g$ (512 to 1024 bits, in steps of 64 bits)</td>
</tr>
<tr>
<td>$\text{CKA_VALUE}^{1,4}$</td>
<td>Big integer</td>
<td>Public value $y$</td>
</tr>
</tbody>
</table>

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

The $\text{CKA\_PRIME}$, $\text{CKA\_SUBPRIME}$ and $\text{CKA\_BASE}$ attribute values are collectively the “KEA domain parameters”.

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

```c
CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK KEA;
CK_UTF8CHAR label[] = "A KEA public key object";
CK_BYTE prime[] = {...};
CK_BYTE subprime[] = {...};
CK_BYTE base[] = {...};
CK_BYTE value[] = {...};
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)}
};
```

### 2.3.4 KEA private key objects

KEA private key objects (object class $\text{CKO\_PRIVATE\_KEY}$, key type $\text{CKK\_KEA}$) hold KEA private keys.

The following table defines the KEA 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>$\text{CKA_PRIME}^{1,4,6}$</td>
<td>Big integer</td>
<td>Prime $p$ (512 to 1024 bits, in steps of 64 bits)</td>
</tr>
<tr>
<td>$\text{CKA_SUBPRIME}^{1,4,6}$</td>
<td>Big integer</td>
<td>Subprime $q$ (160 bits)</td>
</tr>
<tr>
<td>$\text{CKA_BASE}^{1,4,6}$</td>
<td>Big integer</td>
<td>Base $g$ (512 to 1024 bits, in steps of 64 bits)</td>
</tr>
</tbody>
</table>
The **CKA_PRIME**, **CKA_SUBPRIME** and **CKA_BASE** attribute values are collectively the “KEA domain parameters". Note that when generating a KEA private key, the KEA parameters are not specified in the key’s template. This is because KEA private keys are only generated as part of a KEA key pair, and the KEA parameters for the pair are specified in the template for the KEA public key.

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

```c
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_KEA;
CK_UTF8CHAR label[] = "A KEA 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)},
    {Algorithm, as defined by NISTS
    {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_SUBPRIME, subprime, sizeof(subprime)},
    {CKA_BASE, base, sizeof(base)},
    {CKA_VALUE, value, sizeof(value)}
};
```

### 2.3.5 KEA key pair generation

The KEA key pair generation mechanism, denoted **CKM_KEA_KEY_PAIR_GEN**, generates key pairs for the Key Exchange Algorithm, as defined by NIST’s “SKIPJACK and KEA Algorithm Specification Version 2.0”, 29 May 1998.

It does not have a parameter.

The mechanism generates KEA 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. Note that this version of Cryptoki does not include a mechanism for generating these KEA domain parameters.

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 KEA 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 KEA prime sizes, in bits.
2.3.6 KEA key derivation

The KEA key derivation mechanism, denoted **CKM_KEA_DERIVE**, is a mechanism for key derivation based on KEA, the Key Exchange Algorithm, as defined by NIST’s “SKIPJACK and KEA Algorithm Specification Version 2.0”, 29 May 1998.

It has a parameter, a **CK_KEA_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.

As defined in the Specification, KEA MAY be used in two different operational modes: full mode and e-mail mode. Full mode is a two-phase key derivation sequence that requires real-time parameter exchange between two parties. E-mail mode is a one-phase key derivation sequence that does not require real-time parameter exchange. By convention, e-mail mode is designated by use of a fixed value of (1) for the KEA parameter Rb (**pRandomB**).

The operation of this mechanism depends on two of the values in the supplied **CK_KEA_DERIVE_PARAMS** structure, as detailed in the table below. Note that in all cases, the data buffers pointed to by the parameter structure fields **pRandomA** and **pRandomB** must be allocated by the caller prior to invoking **C_DeriveKey**. Also, the values pointed to by **pRandomA** and **pRandomB** are represented as Cryptoki “Big integer” data (i.e., a sequence of bytes, most significant byte first).

<table>
<thead>
<tr>
<th>Value of boolean isSender</th>
<th>Value of big integer pRandomB</th>
<th>Token Action (after checking parameter and template values)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CK_TRUE</td>
<td>0</td>
<td>Compute KEA Ra value, store it in <strong>pRandomA</strong>, return CKR_OK. No derived key object is created.</td>
</tr>
<tr>
<td>CK_TRUE</td>
<td>1</td>
<td>Compute KEA Ra value, store it in <strong>pRandomA</strong>, derive key value using e-mail mode, create key object, return CKR_OK.</td>
</tr>
<tr>
<td>CK_TRUE</td>
<td>&gt;1</td>
<td>Compute KEA Ra value, store it in <strong>pRandomA</strong>, derive key value using full mode, create key object, return CKR_OK.</td>
</tr>
<tr>
<td>CK_FALSE</td>
<td>0</td>
<td>Compute KEA Rb value, store it in <strong>pRandomB</strong>, return CKR_OK. No derived key object is created.</td>
</tr>
<tr>
<td>CK_FALSE</td>
<td>1</td>
<td>Derive key value using e-mail mode, create key object, return CKR_OK.</td>
</tr>
<tr>
<td>CK_FALSE</td>
<td>&gt;1</td>
<td>Derive key value using full mode, create key object, return CKR_OK.</td>
</tr>
</tbody>
</table>

Note that the parameter value **pRandomB** == 0 is a flag that the KEA mechanism is being invoked to compute the party’s public random value (Ra or Rb, for sender or recipient, respectively), not to derive a key. In these cases, any object template supplied as the **C_DeriveKey** pTemplate argument should be ignored.

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

* Note that the rules regarding the **CKA_SENSITIVE**, **CKA_EXTRACTABLE**, **CKA_ALWAYS_SENSITIVE**, and **CKA_NEVER_EXTRACTABLE** attributes have changed in version
The `CKA_SENSITIVE` and `CKA_EXTRACTABLE` attributes in the template for the new key MAY 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 MUST as well. If the base key has its `CKA_ALWAYS_SENSITIVE` attribute set to `CK_TRUE`, then the derived 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 MUST, 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 KEA prime sizes, in bits.

### 2.4 RC2

#### 2.4.1 Definitions

RC2 is a block cipher which is trademarked by RSA Security. It has a variable keysizse and an additional parameter, the "effective number of bits in the RC2 search space", which MAY take on values in the range 1-1024, inclusive. The effective number of bits in the RC2 search space is sometimes specified by an RC2 "version number"; this "version number" is not the same thing as the "effective number of bits", however. There is a canonical way to convert from one to the other.

This section defines the key type “CKK_RC2” for type `CK_KEY_TYPE` as used in the `CKA_KEY_TYPE` attribute of key objects.

Mechanisms:

- `CKM_RC2_KEY_GEN`
- `CKM_RC2_ECB`
- `CKM_RC2_CBC`
- `CKM_RC2_MAC`
- `CKM_RC2_MAC_GENERAL`
- `CKM_RC2_CBC_PAD`

#### 2.4.2 RC2 secret key objects

RC2 secret key objects (object class `CKO_SECRET_KEY`, key type `CKK_RC2`) hold RC2 keys. The following table defines the RC2 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 (1 to 128 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 [PKCS #11-Base] table 11 for footnotes

2.11 to match the policy used by other key derivation mechanisms such as `CKM_SSL3_MASTER_KEY_DERIVE`.

---

2.4 RC2

2.4.1 Definitions

RC2 is a block cipher which is trademarked by RSA Security. It has a variable keysizse and an additional parameter, the “effective number of bits in the RC2 search space”, which MAY take on values in the range 1-1024, inclusive. The effective number of bits in the RC2 search space is sometimes specified by an RC2 “version number”; this “version number” is not the same thing as the “effective number of bits”, however. There is a canonical way to convert from one to the other.

This section defines the key type “CKK_RC2” for type `CK_KEY_TYPE` as used in the `CKA_KEY_TYPE` attribute of key objects.

Mechanisms:

- `CKM_RC2_KEY_GEN`
- `CKM_RC2_ECB`
- `CKM_RC2_CBC`
- `CKM_RC2_MAC`
- `CKM_RC2_MAC_GENERAL`
- `CKM_RC2_CBC_PAD`

2.4.2 RC2 secret key objects

RC2 secret key objects (object class `CKO_SECRET_KEY`, key type `CKK_RC2`) hold RC2 keys. The following table defines the RC2 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 (1 to 128 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 [PKCS #11-Base] table 11 for footnotes

2.11 to match the policy used by other key derivation mechanisms such as `CKM_SSL3_MASTER_KEY_DERIVE`. 
The following is a sample template for creating an RC2 secret key object:

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_RC2;
CK_UTF8CHAR label[] = "An RC2 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)}
};
```

### 2.4.3 RC2 mechanism parameters

#### 2.4.3.1 CK_RC2_PARAMS; CK_RC2_PARAMS_PTR

CK_RC2_PARAMS provides the parameters to the CKM_RC2_ECB and CMK_RC2_MAC mechanisms. It holds the effective number of bits in the RC2 search space. It is defined as follows:

```c
typedef CK_ULONG CK_RC2_PARAMS;
```

CK_RC2_PARAMS_PTR is a pointer to a CK_RC2_PARAMS.

#### 2.4.3.2 CK_RC2_CBC_PARAMS; CK_RC2_CBC_PARAMS_PTR

CK_RC2_CBC_PARAMS is a structure that provides the parameters to the CKM_RC2_CBC and CKM_RC2_CBC_PAD mechanisms. It is defined as follows:

```c
typedef struct CK_RC2_CBC_PARAMS {
    CK_ULONG ulEffectiveBits;
    CK_BYTE iv[8];
} CK_RC2_CBC_PARAMS;
```

The fields of the structure have the following meanings:

- **ulEffectiveBits**: the effective number of bits in the RC2 search space
- **iv**: the initialization vector (IV) for cipher block chaining mode

CK_RC2_CBC_PARAMS_PTR is a pointer to a CK_RC2_CBC_PARAMS.

#### 2.4.3.3 CK_RC2_MAC_GENERAL_PARAMS; CK_RC2_MAC_GENERAL_PARAMS_PTR

CK_RC2_MAC_GENERAL_PARAMS is a structure that provides the parameters to the CKM_RC2_MAC_GENERAL mechanism. It is defined as follows:

```c
typedef struct CK_RC2_MAC_GENERAL_PARAMS {
    CK_ULONG ulEffectiveBits;
    CK_ULONG ulMacLength;
    CK_RC2_MAC_GENERAL_PARAMS;
};
```

The fields of the structure have the following meanings:

- **ulEffectiveBits**: the effective number of bits in the RC2 search space
- **ulMacLength**: length of the MAC produced, in bytes
2.4.4 RC2 key generation

The RC2 key generation mechanism, denoted **CKM_RC2_KEY_GEN**, is a key generation mechanism for RSA Security’s block cipher RC2.

It does not have a parameter.

The mechanism generates RC2 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 RC2 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 RC2 key sizes, in bits.

2.4.5 RC2-ECB

RC2-ECB, denoted **CKM_RC2_ECB**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on RSA Security’s block cipher RC2 and electronic codebook mode as defined in FIPS PUB 81.

It has a parameter, a **CK_RC2_PARAMS**, which indicates the effective number of bits in the RC2 search space.

This mechanism MAY 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 seven null bytes so that the resulting length is a multiple of eight. 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, 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>RC2</td>
<td>Multiple of 8</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>RC2</td>
<td>Multiple of 8</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>RC2</td>
<td>Any</td>
<td>Input length rounded up to multiple of 8</td>
<td></td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>RC2</td>
<td>Multiple of 8</td>
<td>Determined by type of key being unwrapped or <strong>CKA_VALUE_LEN</strong></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 RC2 effective number of bits.
2.4.6 RC2-CBC

RC2_CBC, denoted **CKM_RC2_CBC**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on RSA Security’s block cipher RC2 and cipher-block chaining mode as defined in FIPS PUB 81.

It has a parameter, a **CK_RC2_CBC_PARAMS** structure, where the first field indicates the effective number of bits in the RC2 search space, and the next field is the initialization vector for cipher block chaining mode.

This mechanism MAY 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 seven null bytes so that the resulting length is a multiple of eight. 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_VALUE** 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>RC2</td>
<td>Multiple of 8</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>RC2</td>
<td>Multiple of 8</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>RC2</td>
<td>Any</td>
<td>Input length rounded up to multiple of 8</td>
<td></td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>RC2</td>
<td>Multiple of 8</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 RC2 effective number of bits.

2.4.7 RC2-CBC with PKCS padding

RC2-CBC with PKCS padding, denoted **CKM_RC2_CBC_PAD**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on RSA Security’s block cipher RC2; cipher-block chaining mode as defined in FIPS PUB 81; and the block cipher padding method detailed in PKCS #7.

It has a parameter, a **CK_RC2_CBC_PARAMS** structure, where the first field indicates the effective number of bits in the RC2 search space, and the next field is the 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 MAY wrap and unwrap RSA, Diffie-Hellman, X9.42 Diffie-Hellman, EC (also related to ECDSA) and DSA private keys (see [PKCS #11-Curr], Miscellaneous simple key derivation mechanisms 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 9, RC2-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>RC2</td>
<td>Any</td>
<td>Input length rounded up to multiple of 8</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>RC2</td>
<td>Multiple of 8</td>
<td>Between 1 and 8 bytes shorter than input length</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>RC2</td>
<td>Any</td>
<td>Input length rounded up to multiple of 8</td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>RC2</td>
<td>Multiple of 8</td>
<td>Between 1 and 8 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 RC2 effective number of bits.

2.4.8 General-length RC2-MAC

General-length RC2-MAC, denoted CKM_RC2_MAC_GENERAL, is a mechanism for single-and multiple-part signatures and verification, based on RSA Security's block cipher RC2 and data authorization as defined in FIPS PUB 113.

It has a parameter, a CK_RC2_MAC_GENERAL_PARAMS structure, which specifies the effective number of bits in the RC2 search space and the output length desired from the mechanism.

The output bytes from this mechanism are taken from the start of the final RC2 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>RC2</td>
<td>Any</td>
<td>0-8, as specified in parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>RC2</td>
<td>Any</td>
<td>0-8, 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 RC2 effective number of bits.

2.4.9 RC2-MAC

RC2-MAC, denoted by CKM_RC2_MAC, is a special case of the general-length RC2-MA mechanism (see Section 2.4.8). Instead of taking a CK_RC2_MAC_GENERAL_PARAMS parameter, it takes a CK_RC2_PARAMS parameter, which only contains the effective number of bits in the RC2 search space.

RC2-MAC produces and verifies 4-byte MACs.

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>RC2</td>
<td>Any</td>
<td>4</td>
</tr>
<tr>
<td>C_Verify</td>
<td>RC2</td>
<td>Any</td>
<td>4</td>
</tr>
</tbody>
</table>

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of RC2 effective number of bits.
2.5 RC4

2.5.1 Definitions

This section defines the key type “CKK_RC4” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms

CKM_RC4_KEY_GEN

CKM_RC4

2.5.2 RC4 secret key objects

RC4 secret key objects (object class CKO_SECRET_KEY, key type CKK_RC4) hold RC4 keys. The following table defines the RC4 secret key object attributes, in addition to the common attributes defined for this object class:

Table 12, RC4 Secret Key Object

<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 (1 to 256 bytes)</td>
</tr>
<tr>
<td>CKA_VALUE_LEN2,3,6</td>
<td>CK_ULONG</td>
<td>Length in bytes of key value</td>
</tr>
</tbody>
</table>

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

The following is a sample template for creating an RC4 secret key object:

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_RC4;
CK_UTF8CHAR label[] = "An RC4 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)}
};
```

2.5.3 RC4 key generation

The RC4 key generation mechanism, denoted CKM_RC4_KEY_GEN, is a key generation mechanism for RSA Security’s proprietary stream cipher RC4.

It does not have a parameter.

The mechanism generates RC4 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 RC4 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 RC4 key sizes, in bits.
2.5.4 RC4 mechanism

RC4, denoted `CKM_RC4`, is a mechanism for single- and multiple-part encryption and decryption based on RSA Security’s proprietary stream cipher RC4.

It does not have a parameter.

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

Table 13, RC4: 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>RC4</td>
<td>Any</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>RC4</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 RC4 key sizes, in bits.

2.6 RC5

2.6.1 Definitions

RC5 is a parameterizable block cipher patented by RSA Security. It has a variable wordsize, a variable keysize, and a variable number of rounds. The blocksize of RC5 is equal to twice its wordsize.

This section defines the key type “CKK_RC5” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

- `CKM_RC5_KEY_GEN`
- `CKM_RC5_ECB`
- `CKM_RC5_CBC`
- `CKM_RC5_MAC`
- `CKM_RC5_MAC_GENERAL`
- `CMK_RC5_CBC_PAD`

2.6.2 RC5 secret key objects

RC5 secret key objects (object class `CKO_SECRET_KEY`, key type `CKK_RC5`) hold RC5 keys. The following table defines the RC5 secret key object attributes, in addition to the common attributes defined for this object class.

Table 14, RC5 Secret Key Object

<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 (0 to 255 bytes)</td>
</tr>
<tr>
<td>CKA_VALUE_LEN^2,3,6</td>
<td>CK_ULONG</td>
<td>Length in bytes of key value</td>
</tr>
</tbody>
</table>

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

The following is a sample template for creating an RC5 secret key object:

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_RC5;
CK_UTF8CHAR label[] = "An RC5 secret key object";
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
```
2.6.3 RC5 mechanism parameters

2.6.3.1 CK_RC5_PARAMS; CK_RC5_PARAMS_PTR

CK_RC5_PARAMS provides the parameters to the CKM_RC5_ECB and CKM_RC5_MAC mechanisms. It is defined as follows:

```c
typedef struct CK_RC5_PARAMS {
    CK_ULONG ulWordsize;
    CK_ULONG ulRounds;
} CK_RC5_PARAMS;
```

The fields of the structure have the following meanings:

- `ulWordsize` wordsize of RC5 cipher in bytes
- `ulRounds` number of rounds of RC5 encipherment

CK_RC5_PARAMS_PTR is a pointer to a CK_RC5_PARAMS.

2.6.3.2 CK_RC5_CBC_PARAMS; CK_RC5_CBC_PARAMS_PTR

CK_RC5_CBC_PARAMS is a structure that provides the parameters to the CKM_RC5_CBC and CKM_RC5_CBC_PAD mechanisms. It is defined as follows:

```c
typedef struct CK_RC5_CBC_PARAMS {
    CK_ULONG ulWordsize;
    CK_ULONG ulRounds;
    CK_BYTE_PTR pIv;
    CK_ULONG ulIvLen;
} CK_RC5_CBC_PARAMS;
```

The fields of the structure have the following meanings:

- `ulwordSize` wordsize of RC5 cipher in bytes
- `ulRounds` number of rounds of RC5 encipherment
- `pIV` pointer to initialization vector (IV) for CBC encryption
- `ulIVLen` length of initialization vector (must be same as blocksize)

CK_RC5_CBC_PARAMS_PTR is a pointer to a CK_RC5_CBC_PARAMS.

2.6.3.3 CK_RC5_MAC_GENERAL_PARAMS; CK_RC5_MAC_GENERAL_PARAMS_PTR

CK_RC5_MAC_GENERAL_PARAMS is a structure that provides the parameters to the CKM_RC5_MAC_GENERAL mechanism. It is defined as follows:
typedef struct CK_RC5_MAC_GENERAL_PARAM {
    CK_ULONG ulWordsize;
    CK_ULONG ulRounds;
    CK_ULONG ulMacLength;
} CK_RC5_MAC_GENERAL_PARAMS;

The fields of the structure have the following meanings:

- **ulwordSize**  
  wordsize of RC5 cipher in bytes

- **ulRounds**  
  number of rounds of RC5 encipherment

- **ulMacLength**  
  length of the MAC produced, in bytes

**CK_RC5_MAC_GENERAL_PARAMS_PTR** is a pointer to a **CK_RC5_MAC_GENERAL_PARAMS**.

### 2.6.4 RC5 key generation

The RC5 key generation mechanism, denoted **CKM_RC5_KEY_GEN**, is a key generation mechanism for RSA Security’s block cipher RC5.

It does not have a parameter.

The mechanism generates RC5 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 RC5 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 RC5 key sizes, in bytes.

### 2.6.5 RC5-ECB

RC5-ECB, denoted **CKM_RC5_ECB**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on RSA Security’s block cipher RC5 and electronic codebook mode as defined in FIPS PUB 81.

It has a parameter, **CK_RC5_PARAMS**, which indicates the wordsize and number of rounds of encryption to use.

This mechanism MAY 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 null bytes so that the resulting length is a multiple of the cipher blocksize (twice the wordsize). 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** attributes 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 15, RC5-ECB 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>RC5</td>
<td>Multiple of blocksize</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 RC5 key sizes, in bytes.

### 2.6.6 RC5-CBC

RC5-CBC, denoted `CKM_RC5_CBC`, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on RSA Security's block cipher RC5 and cipher-block chaining mode as defined in FIPS PUB 81.

It has a parameter, a `CK_RC5_CBC_PARAMS` structure, which specifies the wordsize and number of rounds of encryption to use, as well as the initialization vector for cipher block chaining mode.

This mechanism MAY 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 seven null bytes so that the resulting length is a multiple of eight. 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 for 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>RC5</td>
<td>Multiple of blocksize</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>RC5</td>
<td>Multiple of blocksize</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>RC5</td>
<td>Any</td>
<td>Input length rounded up to multiple of blocksize</td>
<td></td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>RC5</td>
<td>Multiple of blocksize</td>
<td>Determined by type of key being unwrapped or <code>CKA_VALUE_LEN</code></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 RC5 key sizes, in bytes.

### 2.6.7 RC5-CBC with PKCS padding

RC5-CBC with PKCS padding, denoted `CKM_RC5_CBC_PAD`, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on RSA Security's block cipher RC5; cipher block chaining mode as defined in FIPS PUB 81; and the block cipher padding method detailed in PKCS #7.
It has a parameter, a `CK_RC5_CBC_PARAMS` structure, which specifies the wordsize and number of rounds of encryption to use, as well as the initialization vector for cipher block chaining mode.

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 MAY wrap and unwrap RSA, Diffie-Hellman, X9.42 Diffie-Hellman, EC (also related to ECDSA) and DSA private keys. 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>
<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>RC5</td>
<td>Any</td>
<td>Input length rounded up to multiple of blocksize</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>RC5</td>
<td>Multiple of blocksize</td>
<td>Between 1 and blocksize bytes shorter than input length</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>RC5</td>
<td>Any</td>
<td>Input length rounded up to multiple of blocksize</td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>RC5</td>
<td>Multiple of blocksize</td>
<td>Between 1 and blocksize 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 RC5 key sizes, in bytes.

2.6.8 General-length RC5-MAC

General-length RC5-MAC, denoted `CKM_RC5_MAC_GENERAL`, is a mechanism for single- and multiple-part signatures and verification, based on RSA Security’s block cipher RC5 and data authentication as defined in FIPS PUB 113.

It has a parameter, a `CK_RC5_MAC_GENERAL_PARAMS` structure, which specifies the wordsize and number of rounds of encryption to use and the output length desired from the mechanism.

The output bytes from this mechanism are taken from the start of the final RC5 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>RC5</td>
<td>Any</td>
<td>0-blocksize, as specified in parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>RC5</td>
<td>Any</td>
<td>0-blocksize, 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 RC5 key sizes, in bytes.

2.6.9 RC5-MAC

RC5-MAC, denoted by `CKM_RC5_MAC`, is a special case of the general-length RC5-MAC mechanism. Instead of taking a `CK_RC5_MAC_GENERAL_PARAMS` parameter, it takes a `CK_RC5_PARAMS` parameter. RC5-MAC produces and verifies MACs half as large as the RC5 blocksize.

Constraints on key types and the length of data are summarized in the following table:
Table 19, RC5-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>RC5</td>
<td>Any</td>
<td>RC5 wordsize = [blocksize/2]</td>
</tr>
<tr>
<td>C_Verify</td>
<td>RC5</td>
<td>Any</td>
<td>RC5 wordsize = [blocksize/2]</td>
</tr>
</tbody>
</table>

For this mechanism, the \textit{ulMinKeySize} and \textit{ulMaxKeySize} fields of the \texttt{CK_MECHANISM_INFO} structure specify the supported range of RC5 key sizes, in bytes.

2.7 General block cipher

2.7.1 Definitions

For brevity’s sake, the mechanisms for the DES, CAST, CAST3, CAST128, IDEA and CDMF block ciphers are described together here. Each of these ciphers has the following mechanisms, which are described in a templatized form.

This section defines the key types “CKK_DES”, “CKK_CAST”, “CKK_CAST3”, “CKK_CAST128”, “CKK_IDEA” and “CKK_CDMF” for type \texttt{CK_KEY_TYPE} as used in the \texttt{CA_KEY_TYPE} attribute of key objects.

Mechanisms:

- \texttt{CKM_DES_KEY_GEN}
- \texttt{CKM_DES_ECB}
- \texttt{CKM_DES_CBC}
- \texttt{CKM_DES_MAC}
- \texttt{CKM_DES_MAC_GENERAL}
- \texttt{CKM_DES_CBC_PAD}
- \texttt{CKM_CDMF_KEY_GEN}
- \texttt{CKM_CDMF_ECB}
- \texttt{CKM_CDMF_CBC}
- \texttt{CKM_CDMF_MAC}
- \texttt{CKM_CDMF_MAC_GENERAL}
- \texttt{CKM_CDMF_CBC_PAD}
- \texttt{CKM_DES_OFB64}
- \texttt{CKM_DES_OFB8}
- \texttt{CKM_DES_CFB64}
- \texttt{CKM_DES_CFB8}
- \texttt{CKM_CAST_KEY_GEN}
- \texttt{CKM_CAST_ECB}
- \texttt{CKM_CAST_CBC}
- \texttt{CKM_CAST_MAC}
- \texttt{CKM_CAST_MAC_GENERAL}
- \texttt{CKM_CAST_CBC_PAD}
- \texttt{CKM_CAST3_KEY_GEN}
- \texttt{CKM_CAST3_ECB}
- \texttt{CKM_CAST3_CBC}
- \texttt{CKM_CAST3_MAC}
2.7.2 DES secret key objects

DES secret key objects (object class CKO_SECRET_KEY, key type CKK_DES) hold single-length DES keys. The following table defines the DES 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 (8 bytes long)</td>
</tr>
</tbody>
</table>

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

DES keys MUST have their parity bits properly set as described in FIPS PUB 46-3. Attempting to create or unwrap a DES key with incorrect parity MUST return an error.

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

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_DES;
CK_UTF8CHAR label[] = "A DES secret key object";
CK_BYTE value[8] = {…};
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.

2.7.3 CAST secret key objects

CAST secret key objects (object class CKO_SECRET_KEY, key type CKK_CAST) hold CAST keys. The following table defines the CAST secret key object attributes, in addition to the common attributes defined for this object class:
Table 21, CAST 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 (1 to 8 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 [PKCS #11-Base] table 11 for footnotes.

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

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_CAST;
CK_UTF8CHAR label[] = "A CAST 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)}
};
```

2.7.4 CAST3 secret key objects

CAST3 secret key objects (object class CKO_SECRET_KEY, key type CKK_CAST3) hold CAST3 keys. The following table defines the CAST3 secret key object attributes, in addition to the common attributes defines for this object class:

Table 22, CAST3 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 (1 to 8 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 [PKCS #11-Base] table 11 for footnotes.

The following is a sample template for creating a CAST3 secret key object:

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_CAST3;
CK_UTF8CHAR label[] = "A CAST3 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)}
};
```

2.7.5 CAST128 secret key objects

CAST128 secret key objects (object class CKO_SECRET_KEY, key type CKK_CAST128) hold CAST128 keys. The following table defines the CAST128 secret key object attributes, in addition to the common attributes defines for this object class:

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_CAST128;
CK_UTF8CHAR label[] = "A CAST128 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)}
};
```
Table 23, CAST128 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 (1 to 16 bytes)</td>
</tr>
<tr>
<td>CKA_VALUE_LEN²,³,⁶</td>
<td>CK_ULONG</td>
<td>Length in bytes of key value</td>
</tr>
</tbody>
</table>

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

The following is a sample template for creating a CAST128 secret key object:

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_CAST128;
CK_UTF8CHAR label[] = "A CAST128 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)}
};
```

2.7.6 IDEA secret key objects

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

Table 24, IDEA Secret Key Object

<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 (16 bytes long)</td>
</tr>
</tbody>
</table>

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

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

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_IDEA;
CK_UTF8CHAR label[] = "An IDEA 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, value, sizeof(value)}
};
```

2.7.7 CDMF secret key objects

IDEA secret key objects (object class CKO_SECRET_KEY, key type CKK_CDMF) hold CDMF keys. The following table defines the CDMF secret key object attributes, in addition to the common attributes defines for this object class:

Table 25, CDMF Secret Key Object

<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 (8 bytes long)</td>
</tr>
</tbody>
</table>
Refer to [PKCS #11-Base] table 11 for footnotes
CDMF keys MUST have their parity bits properly set in exactly the same fashion described for DES keys in FIPS PUB 46-3. Attempting to create or unwrap a CDMF key with incorrect parity MUST return an error.

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

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_CDMF;
CK_UTF8CHAR label[] = "A CDMF secret key object";
CK_BYTE value[8] = {...};
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)}
};
```

### 2.7.8 General block cipher mechanism parameters

#### 2.7.8.1 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), CAST, CAST3, CAST128, IDEA, CDMF and AES ciphers. It also provides the parameters to the general-length HMACing mechanisms (i.e., MD2, MD5, SHA-1, SHA-256, SHA-384, SHA-512, RIPEMD-128 and RIPEMD-160) 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:

```c
typedef CK_ULONG CK_MAC_GENERAL_PARAMS;
```

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

### 2.7.9 General block cipher key generation

Cipher `<NAME>` has a key generation mechanism, "<NAME> key generation", denoted by **CKM_<NAME>_KEY_GEN**.

This mechanism 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 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.

When DES keys or CDMF keys are generated, their parity bits are set properly, as specified in FIPS PUB 46-3. Similarly, when a triple-DES key is generated, each of the DES keys comprising it has its parity bits set properly.

When DES or CDMF keys are generated, it is token-dependent whether or not it is possible for "weak" or "semi-weak" keys to be generated. Similarly, when triple-DES keys are generated, it is token-dependent whether or not it is possible for any of the component DES keys to be "weak" or "semi-weak" keys.

When CAST, CAST3, or CAST128 keys are generated, the template for the secret key must specify a **CKA_VALUE_LEN** attribute.

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the **CK_MECHANISM_INFO** structure MAY be used. The CAST, CAST3, and CAST128 ciphers have variable key sizes, and so for the key generation mechanisms for these ciphers, the `ulMinKeySize` and `ulMaxKeySize` fields of the **CK_MECHANISM_INFO** structure specify the supported range of key sizes, in bytes. For the DES, DES3 (triple-DES), IDEA and CDMF ciphers, these fields and not used.
2.7.10 General block cipher ECB

Cipher <NAME> has an electronic codebook mechanism, “<NAME>-ECB”, denoted CKM_<NAME>_ECB. It is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping with <NAME>.

It does not have a parameter.

This mechanism MAY 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 null bytes so that the resulting length is a multiple of <NAME>’s blocksize. 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 must be specified in the template.

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

Table 26, General Block Cipher ECB: 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>&lt;NAME&gt;</td>
<td>Multiple of blocksize</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>&lt;NAME&gt;</td>
<td>Multiple of blocksize</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>&lt;NAME&gt;</td>
<td>Any</td>
<td>Input length rounded up to multiple of blocksize</td>
<td></td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>&lt;NAME&gt;</td>
<td>Any</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 MAY be used. The CAST, CAST3, and CAST128 ciphers have variable key sizes, and so for these ciphers, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of key sizes, in bytes. For the DES, DES3 (triple-DES), IDEA and CDMF ciphers, these fields are not used.

2.7.11 General block cipher CBC

Cipher <NAME> has a cipher-block chaining mode, “<NAME>-CBC”, denoted CKM_<NAME>_CBC. It is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping with <NAME>.

It has a parameter, an initialization vector for cipher block chaining mode. The initialization vector has the same length as <NAME>’s blocksize.

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

Table 27, General Block Cipher 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>&lt;NAME&gt;</td>
<td>Multiple of blocksize</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>Function</td>
<td>Key type</td>
<td>Input length</td>
<td>Output length</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>----------</td>
<td>--------------------------------</td>
<td>----------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>&lt;NAME&gt;</td>
<td>Multiple of blocksize</td>
<td>Same as input length</td>
<td></td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>&lt;NAME&gt;</td>
<td>Any</td>
<td>Input length rounded up to multiple of blocksize</td>
<td></td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>&lt;NAME&gt;</td>
<td>Any</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 MAY be used. The CAST, CAST3, and CAST128 ciphers have variable key sizes, and so for these ciphers, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure specify the supported range of key sizes, in bytes. For the DES, DES3 (triple-DES), IDEA, and CDMF ciphers, these fields are not used.

### 2.7.12 General block cipher CBC with PKCS padding

Cipher `<NAME>` has a cipher-block chaining mode with PKCS padding, “<NAME>-CBC with PKCS padding”, denoted `CKM_<NAME>_CBC_PAD`. It is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping with `<NAME>`. All ciphertext is padded with PKCS padding.

It has a parameter, an initialization vector for cipher block chaining mode. The initialization vector has the same length as `<NAME>`'s blocksize.

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 MAY wrap and unwrap RSA, Diffie-Hellman, X9.42 Diffie-Hellman, EC (also related to ECDSA) and DSA private keys. The entries in the table below for data length constraints when wrapping and unwrapping keys to not apply to wrapping and unwrapping private keys.

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

### Table 28, General Block Cipher 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>&lt;NAME&gt;</td>
<td>Any</td>
<td>Input length rounded up to multiple of blocksize</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>&lt;NAME&gt;</td>
<td>Multiple of blocksize</td>
<td>Between 1 and blocksize bytes shorter than input length</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>&lt;NAME&gt;</td>
<td>Any</td>
<td>Input length rounded up to multiple of blocksize</td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>&lt;NAME&gt;</td>
<td>Multiple of blocksize</td>
<td>Between 1 and blocksize bytes shorter than input length</td>
</tr>
</tbody>
</table>

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure MAY be used. The CAST, CAST3 and CAST128 ciphers have variable key sizes, and so for these ciphers, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure specify the supported range of key sizes, in bytes. For the DES, DES3 (triple-DES), IDEA, and CDMF ciphers, these fields are not used.

### 2.7.13 General-length general block cipher MAC

Cipher `<NAME>` has a general-length MACing mode, “General-length `<NAME>`-MAC”, denoted `CKM_<NAME>_MAC_GENERAL`. It is a mechanism for single- and multiple-part signatures and...
verification, based on the <NAME> encryption algorithm and data authentication as defined in FIPS PUB
113.
It has a parameter, a CK_MAC_GENERAL_PARAMS, which specifies the size of the output.
The output bytes from this mechanism are taken from the start of the final cipher block produced in the
MACing process.
Constraints on key types and the length of input and output data are summarized in the following table:

Table 29, General-length General Block Cipher 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>&lt;NAME&gt;</td>
<td>Any</td>
<td>0-blocksize, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>&lt;NAME&gt;</td>
<td>Any</td>
<td>0-blocksize, depending on parameters</td>
</tr>
</tbody>
</table>

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure
MAY be used. The CAST, CAST3, and CAST128 ciphers have variable key sizes, and so for these
ciphers, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the
supported range of key sizes, in bytes. For the DES, DES3 (triple-DES), IDEA and CDMF ciphers, these
fields are not used.

2.7.14 General block cipher MAC

Cipher <NAME> has a MACing mechanism, “<NAME>-MAC”, denoted CKM_<NAME>_MAC. This
mechanism is a special case of the CKM_<NAME>_MAC_GENERAL mechanism described above. It
produces an output of size half as large as <NAME>‘s blocksize.
This mechanism has no parameters.
Constraints on key types and the length of data are summarized in the following table:

Table 30, General Block cipher 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>&lt;NAME&gt;</td>
<td>Any</td>
<td>[blocksize/2]</td>
</tr>
<tr>
<td>C_Verify</td>
<td>&lt;NAME&gt;</td>
<td>Any</td>
<td>[blocksize/2]</td>
</tr>
</tbody>
</table>

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure
MAY be used. The CAST, CAST3, and CAST128 ciphers have variable key sizes, and so for these
ciphers, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the
supported range of key sizes, in bytes. For the DES, DES3 (triple-DES), IDEA and CDMF ciphers, these
fields are not used.

2.8 SKIPJACK

2.8.1 Definitions

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

CKM_SKIPJACK_KEY_GEN
CKM_SKIPJACK_ECB64
CKM_SKIPJACK_CBC64
CKM_SKIPJACK_OFB64
CKM_SKIPJACK_CFB64
2.8.2 SKIPJACK secret key objects

SKIPJACK secret key objects (object class CKO_SECRET_KEY, key type CKK_SKIPJACK) holds a single-length MEK or a TEK. The following table defines the SKIPJACK secret object attributes, in addition to the common attributes defined for this object class:

Table 31, SKIPJACK Secret Key Object

<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 (12 bytes long)</td>
</tr>
</tbody>
</table>

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

SKIPJACK keys have 16 checksum bits, and these bits must be properly set. Attempting to create or unwrap a SKIPJACK key with incorrect checksum bits MUST return an error.

It is not clear that any tokens exist (or ever will exist) which permit an application to create a SKIPJACK key with a specified value. Nonetheless, we provide templates for doing so.

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

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_SKIPJACK;
CK_UTF8CHAR label[] = "A SKIPJACK MEK secret key object";
CK_BYTE value[12] = {…};
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)}
};
```

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

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_SKIPJACK;
CK_UTF8CHAR label[] = "A SKIPJACK TEK secret key object";
CK_BYTE value[12] = {…};
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_WRAP, &true, sizeof(true)},
    {CKA_VALUE, value, sizeof(value)}
};
```
2.8.3 SKIPJACK Mechanism parameters

2.8.3.1 CK_SKIPJACK_PRIVATE_WRAP_PARAM;

CK_SKIPJACK_PRIVATE_WRAP_PARAM_PTR

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

```c
typedef struct CK_SKIPJACK_PRIVATE_WRAP_PARAM {
    CK_ULONG ulPasswordLen;
    CK_BYTE_PTR pPassword;
    CK_ULONG ulPublicDataLen;
    CK_BYTE_PTR pPublicData;
    CK_ULONG ulPandGLen;
    CK_ULONG ulQLen;
    CK_ULONG ulRandomLen;
    CK_BYTE_PTR pRandomA;
    CK_BYTE_PTR pPrimeP;
    CK_BYTE_PTR pBaseG;
    CK_BYTE_PTR pSubprimeQ;
} CK_SKIPJACK_PRIVATE_WRAP_PARAM;
```

The fields of the structure have the following meanings:

- `ulPasswordLen` length of the password
- `pPassword` pointer to the buffer which contains the user-supplied password
- `ulPublicDataLen` other party’s key exchange public key size
- `pPublicData` pointer to other party’s key exchange public key value
- `ulPandGLen` length of prime and base values
- `ulQLen` length of subprime value
- `ulRandomLen` size of random Ra, in bytes
- `pPrimeP` pointer to Prime, p, value
- `pBaseG` pointer to Base, b, value
- `pSubprimeQ` pointer to Subprime, q, value

CK_SKIPJACK_PRIVATE_WRAP_PARAM_PTR is a pointer to a CK_PRIVATE_WRAP_PARAM.

2.8.3.2 CK_SKIPJACK_RELAYX_PARAM;

CK_SKIPJACK_RELAYX_PARAM_PTR

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

```c
typedef struct CK_SKIPJACK_RELAYX_PARAM {
    CK_ULONG ulOldWrappedXLen;
    CK_BYTE_PTR pOldWrappedX;
    CK_ULONG ulNewWrappedXLen;
    CK_BYTE_PTR pNewWrappedX;
} CK_SKIPJACK_RELAYX_PARAM;
```
CK ULONG ulOldPasswordLen;
CK_BYTE_PTR pOldPassword;
CK ULONG ulOldPublicDataLen;
CK_BYTE_PTR pOldPublicData;
CK ULONG ulOldRandomLen;
CK_BYTE_PTR pOldRandomA;
CK ULONG ulNewPasswordLen;
CK_BYTE_PTR pNewPassword;
CK ULONG ulNewPublicDataLen;
CK_BYTE_PTR pNewPublicData;
CK ULONG ulNewRandomLen;
CK_BYTE_PTR pNewRandomA;
} CK_SKIPJACK_RELAYX_PARAMS;

The fields of the structure have the following meanings:

- **ulOldWrappedLen**: length of old wrapped key in bytes
- **pOldWrappedX**: pointer to old wrapper key
- **ulOldPasswordLen**: length of the old password
- **pOldPassword**: pointer to the buffer which contains the old user-supplied password
- **ulOldPublicDataLen**: old key exchange public key size
- **pOldPublicData**: pointer to old key exchange public key value
- **ulOldRandomLen**: size of old random Ra in bytes
- **pOldRandomA**: pointer to old Ra data
- **ulNewPasswordLen**: length of the new password
- **pNewPassword**: pointer to the buffer which contains the new user-supplied password
- **ulNewPublicDataLen**: new key exchange public key size
- **pNewPublicData**: pointer to new key exchange public key value
- **ulNewRandomLen**: size of new random Ra in bytes
- **pNewRandomA**: pointer to new Ra data

**CK_SKIPJACK_RELAYX_PARAMS_PTR** is a pointer to a **CK_SKIPJACK_RELAYX_PARAMS**.

### 2.8.4 SKIPJACK key generation

The SKIPJACK key generation mechanism, denoted **CKM_SKIPJACK_KEY_GEN**, is a key generation mechanism for SKIPJACK. The output of this mechanism is called a Message Encryption Key (MEK).

It does not have a parameter.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key.
2.8.5 SKIPJACK-ECB64

SKIPJACK-ECB64, denoted `CKM_SKIPJACK_ECB64`, is a mechanism for single- and multiple-part encryption and decryption with SKIPJACK in 64-bit electronic codebook mode as defined in FIPS PUB 185.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token – in other words, the application can’t specify a particular IV when encrypting. It MAY, of course, specify a particular IV when decrypting.

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>SKIPJACK</td>
<td>Multiple of 8</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>SKIPJACK</td>
<td>Multiple of 8</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
</tbody>
</table>

2.8.6 SKIPJACK-CBC64

SKIPJACK-CBC64, denoted `CKM_SKIPJACK_CBC64`, is a mechanism for single- and multiple-part encryption and decryption with SKIPJACK in 64-bit output feedback mode as defined in FIPS PUB 185.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token – in other words, the application MAY NOT specify a particular IV when encrypting. It MAY, of course, specify a particular IV when decrypting.

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>SKIPJACK</td>
<td>Multiple of 8</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>SKIPJACK</td>
<td>Multiple of 8</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
</tbody>
</table>

2.8.7 SKIPJACK-OFB64

SKIPJACK-OFB64, denoted `CKM_SKIPJACK_OFB64`, is a mechanism for single- and multiple-part encryption and decryption with SKIPJACK in 64-bit output feedback mode as defined in FIPS PUB 185.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token – in other words, the application MAY NOT specify a particular IV when encrypting. It MAY, of course, specify a particular IV when decrypting.

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>SKIPJACK</td>
<td>Multiple of 8</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>SKIPJACK</td>
<td>Multiple of 8</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
</tbody>
</table>

2.8.8 SKIPJACK-CFB64

SKIPJACK-CFB64, denoted `CKM_SKIPJACK_CFB64`, is a mechanism for single- and multiple-part encryption and decryption with SKIPJACK in 64-bit cipher feedback mode as defined in FIPS PUB 185.
It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token – in other words, the application MAY NOT specify a particular IV when encrypting. It MAY, of course, specify a particular IV when decrypting. Constraints on key types and the length of data are summarized in the following table:

Table 35, SKIPJACK-CFB64: Data and 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>SKIPJACK</td>
<td>Multiple of 8</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>SKIPJACK</td>
<td>Multiple of 8</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
</tbody>
</table>

2.8.9 SKIPJACK-CFB32

SKIPJACK-CFB32, denoted CKM_SKIPJACK_CFB32, is a mechanism for single- and multiple-part encryption and decryption with SKIPJACK in 32-bit cipher feedback mode as defined in FIPS PUB 185. It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token – in other words, the application MAY NOT specify a particular IV when encrypting. It MAY, of course, specify a particular IV when decrypting. Constraints on key types and the length of data are summarized in the following table:

Table 36, SKIPJACK-CFB32: Data and 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>SKIPJACK</td>
<td>Multiple of 4</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>SKIPJACK</td>
<td>Multiple of 4</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
</tbody>
</table>

2.8.10 SKIPJACK-CFB16

SKIPJACK-CFB16, denoted CKM_SKIPJACK_CFB16, is a mechanism for single- and multiple-part encryption and decryption with SKIPJACK in 16-bit cipher feedback mode as defined in FIPS PUB 185. It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token – in other words, the application MAY NOT specify a particular IV when encrypting. It MAY, of course, specify a particular IV when decrypting. Constraints on key types and the length of data are summarized in the following table:

Table 37, SKIPJACK-CFB16: Data and 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>SKIPJACK</td>
<td>Multiple of 4</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>SKIPJACK</td>
<td>Multiple of 4</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
</tbody>
</table>

2.8.11 SKIPJACK-CFB8

SKIPJACK-CFB8, denoted CKM_SKIPJACK_CFB8, is a mechanism for single- and multiple-part encryption and decryption with SKIPJACK in 8-bit cipher feedback mode as defined in FIPS PUB 185. It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token – in other words, the application MAY NOT specify a particular IV when encrypting. It MAY, of course, specify a particular IV when decrypting. Constraints on key types and the length of data are summarized in the following table:
Table 38, SKIPJACK-CFB8: Data and 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>SKIPJACK</td>
<td>Multiple of 4</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>SKIPJACK</td>
<td>Multiple of 4</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
</tbody>
</table>

2.8.12 SKIPJACK-WRAP

The SKIPJACK-WRAP mechanism, denoted CKM_SKIPJACK_WRAP, is used to wrap and unwrap a secret key (MEK). It MAY wrap or unwrap SKIPJACK, BATON, and JUNIPER keys. It does not have a parameter.

2.8.13 SKIPJACK-PRIVATE-WRAP

The SKIPJACK-PRIVATE-WRAP mechanism, denoted CKM_SKIPJACK_PRIVATE_WRAP, is used to wrap and unwrap a private key. It MAY wrap KEA and DSA private keys. It has a parameter, a CK_SKIPJACK_PRIVATE_WRAP_PARAMS structure.

2.8.14 SKIPJACK-RELAYX

The SKIPJACK-RELAYX mechanism, denoted CKM_SKIPJACK_RELAYX, is used with the C_WrapKey function to “change the wrapping” on a private key which was wrapped with the SKIPJACK-PRIVATE-WRAP mechanism (See Section 2.8.13). It has a parameter, a CK_SKIPJACK_RELAYX_PARAMS structure.

Although the SKIPJACK-RELAYX mechanism is used with C_WrapKey, it differs from other key-wrapping mechanisms. Other key-wrapping mechanisms take a key handle as one of the arguments to C_WrapKey; however for the SKIPJACK_RELAYX mechanism, the [always invalid] value 0 should be passed as the key handle for C_WrapKey, and the already-wrapped key should be passed in as part of the CK_SKIPJACK_RELAYX_PARAMS structure.

2.9 BATON

2.9.1 Definitions

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

Mechanisms:

<table>
<thead>
<tr>
<th>Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_BATON_KEY_GEN</td>
</tr>
<tr>
<td>CKM_BATON_ECB128</td>
</tr>
<tr>
<td>CKM_BATON_ECB96</td>
</tr>
<tr>
<td>CKM_BATON_CBC128</td>
</tr>
<tr>
<td>CKM_BATON_COUNTER</td>
</tr>
<tr>
<td>CKM_BATON_SHUFFLE</td>
</tr>
<tr>
<td>CKM_BATON_WRAP</td>
</tr>
</tbody>
</table>

2.9.2 BATON secret key objects

BATON secret key objects (object class CKO_SECRET_KEY, key type CKK_BATON) hold single-length BATON keys. The following table defines the BATON secret key object attributes, in addition to the common attributes defined for this object class:
Table 39, BATON Secret Key Object

<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 (40 bytes long)</td>
</tr>
</tbody>
</table>

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

BATON keys have 160 checksum bits, and these bits must be properly set. Attempting to create or unwrap a BATON key with incorrect checksum bits MUST return an error.

It is not clear that any tokens exist (or will ever exist) which permit an application to create a BATON key with a specified value. Nonetheless, we provide templates for doing so.

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

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_BATON;
CK_UTF8CHAR label[] = "A BATON MEK secret key object";
CK_BYTE value[40] = {..};
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)}
};
```

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

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_BATON;
CK_UTF8CHAR label[] = "A BATON TEK secret key object";
CK_BYTE value[40] = {..};
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_WRAP, &true, sizeof(true)},
    {CKA_VALUE, value, sizeof(value)}
};
```

2.9.3 BATON key generation

The BATON key generation mechanism, denoted **CKM_BATON_KEY_GEN**, is a key generation mechanism for BATON. The output of this mechanism is called a Message Encryption Key (MEK).

It does not have a parameter.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key.

2.9.4 BATON-ECB128

BATON-ECB128, denoted **CKM_BATON_ECB128**, is a mechanism for single- and multiple-part encryption and decryption with BATON in 128-bit electronic codebook mode.
It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token – in other words, the application MAY NOT specify a particular IV when encrypting. It MAY, of course, specify a particular IV when decrypting.

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

### Table 40, BATON-ECB128: Data and 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>BATON</td>
<td>Multiple of 16</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>BATON</td>
<td>Multiple of 16</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
</tbody>
</table>

### 2.9.5 BATON-ECB96

BATON-ECB96, denoted **CKM_BATON_ECB96**, is a mechanism for single- and multiple-part encryption and decryption with BATON in 96-bit electronic codebook mode.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token – in other words, the application MAY NOT specify a particular IV when encrypting. It MAY, of course, specify a particular IV when decrypting.

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

### Table 41, BATON-ECB96: Data and 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>BATON</td>
<td>Multiple of 12</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>BATON</td>
<td>Multiple of 12</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
</tbody>
</table>

### 2.9.6 BATON-CBC128

BATON-CBC128, denoted **CKM_BATON_CBC128**, is a mechanism for single- and multiple-part encryption and decryption with BATON in 128-bit cipher-block chaining mode.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token – in other words, the application MAY NOT specify a particular IV when encrypting. It MAY, of course, specify a particular IV when decrypting.

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

### Table 42, BATON-CBC128

<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>BATON</td>
<td>Multiple of 16</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>BATON</td>
<td>Multiple of 16</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
</tbody>
</table>

### 2.9.7 BATON-COUNTER

BATON-COUNTER, denoted **CKM_BATON_COUNTER**, is a mechanism for single- and multiple-part encryption and decryption with BATON in counter mode.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token – in other words, the application MAY NOT specify a particular IV when encrypting. It MAY, of course, specify a particular IV when decrypting.

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>BATON</td>
<td>Multiple of 16</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>BATON</td>
<td>Multiple of 16</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
</tbody>
</table>

#### 2.9.8 BATON-SHUFFLE

BATON-SHUFFLE, denoted **CKM_BATON_SHUFFLE**, is a mechanism for single- and multiple-part encryption and decryption with BATON in shuffle mode.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token – in other words, the application MAY NOT specify a particular IV when encrypting. It MAY, of course, specify a particular IV when decrypting.

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>BATON</td>
<td>Multiple of 16</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>BATON</td>
<td>Multiple of 16</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
</tbody>
</table>

#### 2.9.9 BATON WRAP

The BATON wrap and unwrap mechanism, denoted **CKM_BATON_WRAP**, is a function used to wrap and unwrap a secret key (MEK). It MAY wrap and unwrap SKIPJACK, BATON and JUNIPER keys.

It has no parameters.

When used to unwrap a key, this mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to it.

#### 2.10 JUNIPER

#### 2.10.1 Definitions

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

Mechanisms:

- **CKM_JUNIPER_KEY_GEN**
- **CKM_JUNIPER_ECB128**
- **CKM_JUNIPER_CBC128**
- **CKM_JUNIPER_COUNTER**
- **CKM_JUNIPER_SHUFFLE**
- **CKM_JUNIPER_WRAP**

#### 2.10.2 JUNIPER secret key objects

JUNIPER secret key objects (object class **CKO_SECRET_KEY**, key type **CKK_JUNIPER**) hold single-length JUNIPER keys. The following table defines the BATON secret key object attributes, in addition to the common attributes defined for this object class:
Table 45, JUNIPER Secret Key Object

<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 (40 bytes long)</td>
</tr>
</tbody>
</table>

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

JUNIPER keys have 160 checksum bits, and these bits must be properly set. Attempting to create or unwrap a BATON key with incorrect checksum bits MUST return an error.

It is not clear that any tokens exist (or will ever exist) which permit an application to create a BATON key with a specified value. Nonetheless, we provide templates for doing so.

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

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_JUNIPER;
CK_UTF8CHAR label[] = "A JUNIPER MEK secret key object";
CK_BYTE value[40] = {...};
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)}
};
```

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

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_JUNIPER;
CK_UTF8CHAR label[] = "A JUNIPER TEK secret key object";
CK_BYTE value[40] = {...};
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_WRAP, &true, sizeof(true)},
    {CKA_VALUE, value, sizeof(value)}
};
```

2.10.3 JUNIPER key generation

The JUNIPER key generation mechanism, denoted CKM_JUNIPER_KEY_GEN, is a key generation mechanism for JUNIPER. The output of this mechanism is called a Message Encryption Key (MEK).

It does not have a parameter.

The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, and CKA_VALUE attributes to the new key.

2.10.4 JUNIPER-ECB128

JUNIPER-ECB128, denoted CKM_JUNIPER_ECB128, is a mechanism for single- and multiple-part encryption and decryption with JUNIPER in 128-bit electronic codebook mode.
It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token – in other words, the application MAY NOT specify a particular IV when encrypting. It MAY, of course, specify a particular IV when decrypting.

Constraints on key types and the length of data are summarized in the following table. For encryption and decryption, the input and output data (parts) MAY begin at the same location in memory.

<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>JUNIPER</td>
<td>Multiple of 16</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>JUNIPER</td>
<td>Multiple of 16</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
</tbody>
</table>

2.10.5 JUNIPER-CBC128

JUNIPER-CBC128, denoted **CKM_JUNIPER_CBC128**, is a mechanism for single- and multiple-part encryption and decryption with JUNIPER in 128-bit cipher block chaining mode.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token – in other words, the application MAY NOT specify a particular IV when encrypting. It MAY, of course, specify a particular IV when decrypting.

Constraints on key types and the length of data are summarized in the following table. For encryption and decryption, the input and output data (parts) MAY begin at the same location in memory.

<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>JUNIPER</td>
<td>Multiple of 16</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>JUNIPER</td>
<td>Multiple of 16</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
</tbody>
</table>

2.10.6 JUNIPER-COUNTER

JUNIPER-COUNTER, denoted **CKM_JUNIPER_COUNTER**, is a mechanism for single- and multiple-part encryption and decryption with JUNIPER in counter mode.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token – in other words, the application MAY NOT specify a particular IV when encrypting. It MAY, of course, specify a particular IV when decrypting.

Constraints on key types and the length of data are summarized in the following table. For encryption and decryption, the input and output data (parts) MAY begin at the same location in memory.

<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>JUNIPER</td>
<td>Multiple of 16</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>JUNIPER</td>
<td>Multiple of 16</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
</tbody>
</table>

2.10.7 JUNIPER-SHUFFLE

JUNIPER-SHUFFLE, denoted **CKM_JUNIPER_SHUFFLE**, is a mechanism for single- and multiple-part encryption and decryption with JUNIPER in shuffle mode.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token – in other words, the application MAY NOT specify a particular IV when encrypting. It MAY, of course, specify a particular IV when decrypting.
Constraints on key types and the length of data are summarized in the following table. For encryption and decryption, the input and output data (parts) MAY begin at the same location in memory.

Table 49, JUNIPER-SHUFFLE: Data and 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>JUNIPER</td>
<td>Multiple of 16</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>JUNIPER</td>
<td>Multiple of 16</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
</tbody>
</table>

2.10.8 JUNIPER WRAP

The JUNIPER wrap and unwrap mechanism, denoted CKM_JUNIPER_WRAP, is a function used to wrap and unwrap an MEK. It MAY wrap or unwrap SKIPJACK, BATON and JUNIPER keys. It has no parameters. When used to unwrap a key, this mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, and CKA_VALUE attributes to it.

2.11 MD2

2.11.1 Definitions

Mechanisms:

- CKM_MD2
- CKM_MD2_HMAC
- CKM_MD2_HMAC_GENERAL
- CKM_MD2_KEY_DERIVATION

2.11.2 MD2 digest

The MD2 mechanism, denoted CKM_MD2, is a mechanism for message digesting, following the MD2 message-digest algorithm defined in RFC 6149. It does not have a parameter. Constraints on the length of data are summarized in the following table:

Table 50, MD2: Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Data length</th>
<th>Digest Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>Any</td>
<td>16</td>
</tr>
</tbody>
</table>

2.11.3 General-length MD2-HMAC

The general-length MD2-HMAC mechanism, denoted CKM_MD2_HMAC_GENERAL, is a mechanism for signatures and verification. It uses the HMAC construction, based on the MD2 hash function. The keys it uses are generic secret keys. 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 0-16 (the output size of MD2 is 16 bytes). Signatures (MACs) produced by this mechanism MUST be taken from the start of the full 16-byte HMAC output.

Table 51, General-length MD2-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>0-16, depending on parameters</td>
</tr>
</tbody>
</table>
C_Verify  Generic secret  Any  0-16, depending on parameters

2.11.4 MD2-HMAC
The MD2-HMAC mechanism, denoted **CKM_MD2_HMAC**, is a special case of the general-length MD2-HMAC mechanism in Section 2.11.3.
It has no parameter, and produces an output of length 16.

2.11.5 MD2 key derivation
MD2 key derivation, denoted **CKM_MD2_KEY_DERIVATION**, is a mechanism which provides the capability of deriving a secret key by digesting the value of another secret key with MD2.
The value of the base key is digested once, and the result is used to make the value of the derived secret key.

- If no length or key type is provided in the template, then the key produced by this mechanism MUST be a generic secret key. Its length MUST be 16 bytes (the output size of MD2).
- If no key type is provided in the template, but a length is, then the key produced by this mechanism MUST 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 MUST be of the type specified in the template. If it doesn’t, an error MUST 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 MUST 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 MUST be set properly.
If the requested type of key requires more than 16 bytes, such as DES2, 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 MAY 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 MUST 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 MUST, 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.

2.12 MD5

2.12.1 Definitions
Mechanisms:
- **CKM_MD5**
- **CKM_MD5_HMAC**
- **CKM_MD5_HMAC_GENERAL**
- **CKM_MD5_KEY_DERIVATION**
2.12.2 MD5 Digest

The MD5 mechanism, denoted CKM_MD5, is a mechanism for message digesting, following the MD5 message-digest algorithm defined in RFC 1321.

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>Data length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>Any</td>
<td>16</td>
</tr>
</tbody>
</table>

2.12.3 General-length MD5-HMAC

The general-length MD5-HMAC mechanism, denoted CKM_MD5_HMAC_GENERAL, is a mechanism for signatures and verification. It uses the HMAC construction, based on the MD5 hash function. The keys it uses are generic secret keys.

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 0-16 (the output size of MD5 is 16 bytes). Signatures (MACs) produced by this mechanism MUST be taken from the start of the full 16-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</td>
<td>Any</td>
<td>0-16, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>Generic secret</td>
<td>Any</td>
<td>0-16, depending on parameters</td>
</tr>
</tbody>
</table>

2.12.4 MD5-HMAC

The MD5-HMAC mechanism, denoted CKM_MD5_HMAC, is a special case of the general-length MD5-HMAC mechanism in Section 2.12.3.

It has no parameter, and produces an output of length 16.

2.12.5 MD5 key derivation

MD5 key derivation denoted CKM_MD5_KEY_DERIVATION, is a mechanism which provides the capability of deriving a secret key by digesting the value of another secret key with MD5.

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 MUST be a generic secret key. Its length MUST be 16 bytes (the output size of MD5).
- If no key type is provided in the template, but a length is, then the key produced by this mechanism MUST 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 MUST be of the type specified in the template. If it doesn’t, an error MUST 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 MUST 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 MUST be set properly.
- If the requested type of key requires more than 16 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 MAY both be specified to 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 MUST 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 MUST, 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.

### 2.13 FASTHASH

#### 2.13.1 Definitions
Mechanisms:

- CKM_FASTHASH

#### 2.13.2 FASTHASH digest

The FASTHASH mechanism, denoted CKM_FASTHASH, is a mechanism for message digesting, following the U.S. government's algorithm. It does not have a parameter. Constraints on the length of input and output data are summarized in the following table:

<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>40</td>
</tr>
</tbody>
</table>

### 2.14 PKCS #5 and PKCS #5-style password-based encryption (PBD)

#### 2.14.1 Definitions

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.

Mechanisms:

- CKM_PBE_MD2_DES_CBC
- CKM_PBE_MD5_DES_CBC
- CKM_PBE_MD5_CAST_CBC
- CKM_PBE_MD5_CAST3_CBC
- CKM_PBE_MD5_CAST128_CBC
- CKM_PBE_SHA1_CAST128_CBC
- CKM_PBE_SHA1_RC4_128
- CKM_PBE_SHA1_RC4_40
- CKM_PBE_SHA1_RC2_128_CBC
- CKM_PBE_SHA1_RC2_40_CBC
2.14.2 Password-based encryption/authentication mechanism parameters

2.14.2.1 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
- `ulIteration`: number of iterations required for the generation

CK_PBE_PARAMS_PTR is a pointer to a CK_PBE_PARAMS.

2.14.3 MD2-PBE for DES-CBC

MD2-PBE for DES-CBC, denoted CKM_PBE_MD2_DES_CBC, is a mechanism used for generating a DES secret key and an IV from a password and a salt value by using the MD2 digest algorithm and an iteration count. This functionality is defined in PKCS #5 as PBKDF1.

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 receives the 8-byte IV generated by the mechanism.

2.14.4 MD5-PBE for DES-CBC

MD5-PBE for DES-CBC, denoted CKM_PBE_MD5_DES_CBC, is a mechanism used for generating a DES secret key and an IV from a password and a salt value by using the MD5 digest algorithm and an iteration count. This functionality is defined in PKCS #5 as PBKDF1.

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 receives the 8-byte IV generated by the mechanism.
2.14.5 MD5-PBE for CAST-CBC

MD5-PBE for CAST-CBC, denoted CKM_PBE_MD5_CAST_CBC, is a mechanism used for generating a CAST secret key and an IV from a password and a salt value by using the MD5 digest algorithm and an iteration count. This functionality is analogous to that defined in PKCS #5 PBKDF1 for MD5 and DES.

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 receives the 8-byte IV generated by the mechanism.

The length of the CAST key generated by this mechanism MAY be specified in the supplied template; if it is not present in the template, it defaults to 8 bytes.

2.14.6 MD5-PBE for CAST3-CBC

MD5-PBE for CAST3-CBC, denoted CKM_PBE_MD5_CAST3_CBC, is a mechanism used for generating a CAST3 secret key and an IV from a password and a salt value by using the MD5 digest algorithm and an iteration count. This functionality is analogous to that defined in PKCS #5 PBKDF1 for MD5 and DES.

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 receives the 8-byte IV generated by the mechanism.

The length of the CAST3 key generated by this mechanism MAY be specified in the supplied template; if it is not present in the template, it defaults to 8 bytes.

2.14.7 MD5-PBE for CAST128-CBC

MD5-PBE for CAST128-CBC, denoted CKM_PBE_MD5_CAST128_CBC, is a mechanism used for generating a CAST128 secret key and an IV from a password and a salt value by using the MD5 digest algorithm and an iteration count. This functionality is analogous to that defined in PKCS #5 PBKDF1 for MD5 and DES.

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 receives the 8-byte IV generated by the mechanism.

The length of the CAST128 key generated by this mechanism MAY be specified in the supplied template; if it is not present in the template, it defaults to 8 bytes.

2.14.8 SHA-1-PBE for CAST128-CBC

SHA-1-PBE for CAST128-CBC, denoted CKM_PBE_SHA1_CAST128_CBC, is a mechanism used for generating a CAST128 secret key and an IV from a password and salt value using the SHA-1 digest algorithm and an iteration count. This functionality is analogous to that defined in PKCS #5 PBKDF1 for MD5 and DES.

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 receives the 8-byte IV generated by the mechanism.

The length of the CAST128 key generated by this mechanism MAY be specified in the supplied template; if it is not present in the template, it defaults to 8 bytes.

2.15 PKCS #12 password-based encryption/authentication mechanisms

2.15.1 Definitions

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, $ID$, described at the end of this section.

Let $H$ be a hash function built around a compression function $f: Z^r \times Z^s \rightarrow Z^v$ (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 $l=S||P$ to be the concatenation of $S$ and $P$.

5. Set $j=\lceil n/u \rceil$.

6. For $i=1, 2, \ldots, j$, do the following:
   a. Set $A_i=Hc^i(D||l)$. That is, compute the hash of $D||l$; 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 $l_0, l_1, \ldots, l_k$ of $v$-bit blocks, where $k=\lceil s/v \rceil \lceil p/v \rceil$, modify $l$ by setting $l_i=(l_i+l_{i}+1) \mod 2^v$ for each $i$. 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 $ID$ is set to the value 3.

### 2.15.2 SHA-1-PBE for 128-bit RC4

SHA-1-PBE for 128-bit RC4, denoted **CKM_PBE_SHA1_RC4_128**, is a mechanism used for generating a 128-bit RC4 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 receives an IV; for this mechanism, the contents of this field are ignored, since RC4 does not require an IV.

The key produced by this mechanism will typically be used for performing password-based encryption.
2.15.3 SHA-1_PBE for 40-bit RC4

SHA-1-PBE for 40-bit RC4, denoted \texttt{CKM\_PBE\_SHA1\_RC4\_40}, is a mechanism used for generating a 40-bit RC4 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 \texttt{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 receives an IV; for this mechanism, the contents of this field are ignored, since RC4 does not require an IV.

The key produced by this mechanism will typically be used for performing password-based encryption.

2.15.4 SHA-1_PBE for 128-bit RC2-CBC

SHA-1-PBE for 128-bit RC2-CBC, denoted \texttt{CKM\_PBE\_SHA1\_RC2\_128\_CBC}, is a mechanism used for generating a 128-bit RC2 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 and IV is described above.

It has a parameter, a \texttt{CK\_PBE\_PARAMS} structure. The parameter specifies the input information for the key generation process and the location of an application-supplied buffer which receives the 8-byte IV generated by the mechanism.

When the key and IV generated by this mechanism are used to encrypt or decrypt, the effective number of bits in the RC2 search space should be set to 128. This ensures compatibility with the ASN.1 Object Identifier \texttt{pbeWithSHA1And128BitRC2-CBC}.

The key and IV produced by this mechanism will typically be used for performing password-based encryption.

2.15.5 SHA-1_PBE for 40-bit RC2-CBC

SHA-1-PBE for 40-bit RC2-CBC, denoted \texttt{CKM\_PBE\_SHA1\_RC2\_40\_CBC}, is a mechanism used for generating a 40-bit RC2 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 and IV is described above.

It has a parameter, a \texttt{CK\_PBE\_PARAMS} structure. The parameter specifies the input information for the key generation process and the location of an application-supplied buffer which receives the 8-byte IV generated by the mechanism.

When the key and IV generated by this mechanism are used to encrypt or decrypt, the effective number of bits in the RC2 search space should be set to 40. This ensures compatibility with the ASN.1 Object Identifier \texttt{pbeWithSHA1And40BitRC2-CBC}.

The key and IV produced by this mechanism will typically be used for performing password-based encryption.

2.16 RIPE-MD

2.16.1 Definitions

Mechanisms:

- \texttt{CKM\_RIPEMD128}
- \texttt{CKM\_RIPEMD128\_HMAC}
- \texttt{CKM\_RIPEMD128\_HMAC\_GENERAL}
- \texttt{CKM\_RIPEMD160}
- \texttt{CKM\_RIPEMD160\_HMAC}
- \texttt{CKM\_RIPEMD160\_HMAC\_GENERAL}
2.16.2 RIPE-MD 128 Digest

The RIPE-MD 128 mechanism, denoted CKM_RIPEMD128, is a mechanism for message digesting, following the RIPE-MD 128 message-digest algorithm.

It does not have a parameter.

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Data length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>Any</td>
<td>16</td>
</tr>
</tbody>
</table>

2.16.3 General-length RIPE-MD 128-HMAC

The general-length RIPE-MD 128-HMAC mechanism, denoted CKM_RIPEMD128_HMAC_GENERAL, is a mechanism for signatures and verification. It uses the HMAC construction, based on the RIPE-MD 128 hash function. The keys it uses are generic secret keys.

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 0-16 (the output size of RIPE-MD 128 is 16 bytes). Signatures (MACs) produced by this mechanism MUST be taken from the start of the full 16-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</td>
<td>Any</td>
<td>0-16, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>Generic secret</td>
<td>Any</td>
<td>0-16, depending on parameters</td>
</tr>
</tbody>
</table>

2.16.4 RIPE-MD 128-HMAC

The RIPE-MD 128-HMAC mechanism, denoted CKM_RIPEMD128_HMAC, is a special case of the general-length RIPE-MD 128-HMAC mechanism in Section 2.16.3.

It has no parameter, and produces an output of length 16.

2.16.5 RIPE-MD 160

The RIPE-MD 160 mechanism, denoted CKM_RIPEMD160, is a mechanism for message digesting, following the RIPE-MD 160 message-digest defined in ISO-10118.

It does not have a parameter.

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Data length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>Any</td>
<td>20</td>
</tr>
</tbody>
</table>

2.16.6 General-length RIPE-MD 160-HMAC

The general-length RIPE-MD 160-HMAC mechanism, denoted CKM_RIPEMD160_HMAC_GENERAL, is a mechanism for signatures and verification. It uses the HMAC construction, based on the RIPE-MD 160 hash function. The keys it uses are generic secret keys.
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 0-20 (the output size of RIPE-MD 160 is 20 bytes). Signatures (MACs) produced by this mechanism MUST be taken from the start of the full 20-byte HMAC output.

*Table 58, General-length RIPE-MD 160-HMAC: Data and 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>0-20, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>Generic secret</td>
<td>Any</td>
<td>0-20, depending on parameters</td>
</tr>
</tbody>
</table>

### 2.16.7 RIPE-MD 160-HMAC

The RIPE-MD 160-HMAC mechanism, denoted **CKM_RIPEMD160_HMAC**, is a special case of the general-length RIPE-MD 160HMAC mechanism in Section 2.16.6. It has no parameter, and produces an output of length 20.

### 2.17 SET

#### 2.17.1 Definitions

Mechanisms:

- **CKM_KEY_WRAP_SET_OAEP**

#### 2.17.2 SET mechanism parameters

#### 2.17.2.1 CK_KEY_WRAP_SET_OAEP_PARAMS; CK_KEY_WRAP_SET_OAEP_PARAMS_PTR

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

```c
typedef struct CK_KEY_WRAP_SET_OAEP_PARAMS {
    CK_BYTE bBC;
    CK_BYTE_PTR pX;
    CK_ULONG ulXLen;
} CK_KEY_WRAP_SET_OAEP_PARAMS;
```

The fields of the structure have the following meanings:

- **bBC** block contents byte
- **pX** concatenation of hash of plaintext data (if present) and extra data (if present)
- **ulXLen** length in bytes of concatenation of hash of plaintext data (if present) and extra data (if present). 0 if neither is present.

**CK_KEY_WRAP_SET_OAEP_PARAMS_PTR** is a pointer to a **CK_KEY_WRAP_SET_OAEP_PARAMS**.

#### 2.17.3 OAEP key wrapping for SET

The OAEP key wrapping for SET mechanism, denoted **CKM_KEY_WRAP_SET_OAEP**, is a mechanism for wrapping and unwrapping a DES key with an RSA key. The hash of some plaintext data and/or some
extra data MAY be wrapped together with the DES key. This mechanism is defined in the SET protocol specifications.

It takes a parameter, a **CK_KEY_WRAP_SET_OAEP_PARAMS** structure. This structure holds the "Block Contents" byte of the data and the concatenation of the hash of plaintext data (if present) and the extra data to be wrapped (if present). If neither the hash nor the extra data is present, this is indicated by the *ulXLen* field having the value 0.

When this mechanism is used to unwrap a key, the concatenation of the hash of plaintext data (if present) and the extra data (if present) is returned following the convention described [PKCS #11-Curr], Miscellaneous simple key derivation mechanisms. Note that if the inputs to C_UnwrapKey are such that the extra data is not returned (e.g. the buffer supplied in the **CK_KEY_WRAP_SET_OAEP_PARAMS** structure is NULL_PTR), then the unwrapped key object MUST NOT be created, either.

Be aware that when this mechanism is used to unwrap a key, the *bBC* and *pX* fields of the parameter supplied to the mechanism MAY be modified.

If an application uses C_UnwrapKey with **CKM_KEY_WRAP_SET_OAEP**, it may be preferable for it simply to allocate a 128-byte buffer for the concatenation of the hash of plaintext data and the extra data (this concatenation MUST NOT be larger than 128 bytes), rather than calling C_UnwrapKey twice. Each call of C_UnwrapKey with **CKM_KEY_WRAP_SET_OAEP** requires an RSA decryption operation to be performed, and this computational overhead MAY be avoided by this means.

### 2.18 LYNKS

#### 2.18.1 Definitions

Mechanisms:

- **CKM_KEY_WRAP_LYNKS**

#### 2.18.2 LYNKS key wrapping

The LYNKS key wrapping mechanism, denoted **CKM_KEY_WRAP_LYNKS**, is a mechanism for wrapping and unwrapping secret keys with DES keys. It MAY wrap any 8-byte secret key, and it produces a 10-byte wrapped key, containing a cryptographic checksum.

It does not have a parameter.

To wrap an 8-byte secret key *K* with a DES key *W*, this mechanism performs the following steps:

1. Initialize two 16-bit integers, *sum1* and *sum2*, to 0
2. Loop through the bytes of *K* from first to last.
3. Set *sum1* = *sum1*+the key byte (treat the key byte as a number in the range 0-255).
4. Set *sum2* = *sum2*+ *sum1*.
5. Encrypt *K* with *W* in ECB mode, obtaining an encrypted key, *E*.
6. Concatenate the last 6 bytes of *E* with *sum2*, representing *sum2* most-significant bit first. The result is an 8-byte block, *T*.
7. Encrypt *T* with *W* in ECB mode, obtaining an encrypted checksum, *C*.
8. Concatenate *E* with the last 2 bytes of *C* to obtain the wrapped key.

When unwrapping a key with this mechanism, if the cryptographic checksum does not check out properly, an error is returned. In addition, if a DES key or CDMF key is unwrapped with this mechanism, the parity bits on the wrapped key must be set appropriately. If they are not set properly, an error is returned.
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 [PKCS #11-Prof].

A PKCS #11 implementation SHALL be a conforming PKCS #11 implementation.

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:
Gil Abel, Athena Smartcard Solutions, Inc.
Warren Armstrong, Quintessence Labs
Jeff Bartell, Semper Fortis 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, Quintessence Labs
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.
Appendix B. Manifest constants

The definitions for manifest constants specified in this document can be found in the following normative computer language definition files:

- include/pkcs11-v3.0/pkcs11.h
- include/pkcs11-v3.0/pkcs11t.h
- include/pkcs11-v3.0/pkcs11f.h

These files are linked from the "Additional artifacts" section at the top of this specification.
## Appendix C. Revision History

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Editor</th>
<th>Changes Made</th>
</tr>
</thead>
</table>
| wd01     | 20 April 2019  | Dieter Bong| - All CAST5 definitions removed  
- Replaced reference to [PKCS11-Base] table 10 by [PKCS11-Base] table 11 throughout whole document |
| wd02     | May 28, 2019   | Tony Cox   | Final cleanup of front introductory texts and links prior to CSPRD            |