PKCS #11 Other Mechanisms v2.30: Cryptoki

RSA Laboratories

16 April 2009

Table of Contents

List of Tables

1 Introduction

This document lists the PKCS#11 mechanisms in active use at the time of writing. Refer to PKCS#11 Obsolete Mechanisms for additional mechanisms defined for PKCS#11 but no longer in common use.

2 Scope

A number of cryptographic mechanisms (algorithms) are supported in this version. In addition, new mechanisms can be added later without changing the general interface. It is possible that additional mechanisms will be published from time to time in separate documents; it is also possible for token vendors to define their own mechanisms (although, for the sake of interoperability, registration through the PKCS process is preferable).

3 References

ANSI C — ANSI/ISO. *American National Standard for Programming Languages – C*. 1990.

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

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

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

ANSI X9.63 — Accredited Standards Committee X9. *Public Key Cryptography for the Financial Services Industry: Key Agreement and Key Transport Using Elliptic Curve Cryptography*. 2001.

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

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

FIPS PUB 46–3 — NIST. *FIPS 46-3: Data Encryption Standard (DES).* October 25, 1999. URL: http://csrc.nist.gov/publications/fips/index.html

FIPS PUB 74 NIST. *FIPS 74: Guidelines for Implementing and Using the NBS Data Encryption Standard.* April 1, 1981. URL: http://csrc.nist.gov/publications/fips/index.html

FIPS PUB 81 NIST. *FIPS 81: DES Modes of Operation.* December 1980. URL: http://csrc.nist.gov/publications/fips/index.html

FIPS PUB 113 NIST. *FIPS 113: Computer Data Authentication.* May 30, 1985. URL: http://csrc.nist.gov/publications/fips/index.html

FIPS PUB 180-2 NIST. *FIPS 180-2: Secure Hash Standard.* August 1, 2002. URL: http://csrc.nist.gov/publications/fips/index.html

FIPS PUB 186-2 NIST. *FIPS 186-2: Digital Signature Standard.* January 27, 2000. URL: http://csrc.nist.gov/publications/fips/index.html

FIPS PUB 197 NIST. *FIPS 197: Advanced Encryption Standard (AES).* November 26, 2001. URL: http://csrc.nist.gov/publications/fips/index.html

FORTEZZA CIPG NSA, Workstation Security Products. *FORTEZZA Cryptologic Interface Programmers Guide, Revision 1.52*. November 1995.

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

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

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

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

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

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

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

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

Java MIDP Java Community Process. *Mobile Information Device Profile for Java 2 Micro Edition.* November 2002. URL: http://jcp.org/jsr/detail/118.jsp

MeT-PTD MeT. *MeT PTD Definition – Personal Trusted Device Definition*, Version 1.0, February 2003. URL: http://www.mobiletransaction.org

PCMCIA	Personal Computer Memory Card International Association. *PC Card Standard*, Release 2.1,. July 1993.
PKCS #1	RSA Laboratories. *RSA Cryptography Standard.* v2.1, June 14, 2002.
PKCS #3	RSA Laboratories. *Diffie-Hellman Key-Agreement Standard.* v1.4, November 1993.
PKCS #5	RSA Laboratories. *Password-Based Encryption Standard.* v2.0, March 25, 1999.
PKCS #7	RSA Laboratories. *Cryptographic Message Syntax Standard.* v1.5, November 1993.
PKCS #8	RSA Laboratories. *Private-Key Information Syntax Standard.* v1.2, November 1993.
PKCS #11-C	RSA Laboratories. *PKCS #11: Conformance Profile Specification*, October 2000.
PKCS #11-P	RSA Laboratories. *PKCS #11 Profiles for mobile devices*, June 2003.
PKCS #11-B	RSA Laboratories. *PKCS #11 Base Functionality*, April 2009.
PKCS #12	RSA Laboratories. *Personal Information Exchange Syntax Standard.* v1.0, June 1999.
RFC 1319	B. Kaliski. *RFC 1319: The MD2 Message-Digest Algorithm.* RSA Laboratories, April 1992. URL: http://ietf.org/rfc/rfc1319.txt
RFC 1321	R. Rivest. *RFC 1321: The MD5 Message-Digest Algorithm.* MIT Laboratory for Computer Science and RSA Data Security, Inc., April 1992. URL: http://ietf.org/rfc/rfc1321.txt
RFC 1421	J. Linn. *RFC 1421: Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures.* IAB IRTF PSRG, IETF PEM WG, February 1993. URL: http://ietf.org/rfc/rfc1421.txt
RFC 2045	Freed, N., and N. Borenstein. *RFC 2045: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*. November 1996. URL: http://ietf.org/rfc/rfc2045.txt
RFC 2246	T. Dierks & C. Allen. *RFC 2246: The TLS Protocol Version 1.0*. Certicom, January 1999. URL: http://ietf.org/rfc/rfc2246.txt
RFC 2279	F. Yergeau. *RFC 2279:* UTF-8, a transformation format of ISO 10646 Alis Technologies, January 1998. URL: http://ietf.org/rfc/rfc2279.txt
RFC 2534	Masinter, L., Wing, D., Mutz, A., and K. Holtman. *RFC 2534: Media Features for Display, Print, and Fax.* March 1999. URL: http://ietf.org/rfc/rfc2534.txt
RFC 2630	R. Housley. *RFC 2630: Cryptographic Message Syntax*. June 1999. URL: http://ietf.org/rfc/rfc2630.txt

RFC 2743	J. Linn. *RFC 2743: Generic Security Service Application Program Interface Version 2, Update 1.* RSA Laboratories, January 2000. URL: http://ietf.org/rfc/rfc2743.txt
RFC 2744	J. Wray. *RFC 2744: Generic Security Services API Version 2: C-bindings.* Iris Associates, January 2000. URL: http://ietf.org/rfc/rfc2744.txt
SEC 1	Standards for Efficient Cryptography Group (SECG). *Standards for Efficient Cryptography (SEC) 1: Elliptic Curve Cryptography*. Version 1.0, September 20, 2000.
SEC 2	Standards for Efficient Cryptography Group (SECG). Standards for Efficient Cryptography (SEC) 2: Recommended Elliptic Curve Domain Parameters. Version 1.0, September 20, 2000.
TLS	IETF. *RFC 2246: The TLS Protocol Version 1.0* . January 1999. URL: http://ietf.org/rfc/rfc2246.txt
WIM	WAP. *Wireless Identity Module. — WAP-260-WIM-20010712-a.* July 2001. URL: http://www.wapforum.org/
WPKI	WAP. *Wireless PKI. — WAP-217-WPKI-20010424-a.* April 2001. URL: http://www.wapforum.org/
WTLS	WAP. *Wireless Transport Layer Security Version — WAP-261-WTLS-20010406-a.* April 2001. URL: http://www.wapforum.org/.
X.500	ITU-T. *Information Technology — Open Systems Interconnection — The Directory: Overview of Concepts, Models and Services.* February 2001. Identical to ISO/IEC 9594-1
X.509	ITU-T. *Information Technology — Open Systems Interconnection — The Directory: Public-key and Attribute Certificate Frameworks.* March 2000. Identical to ISO/IEC 9594-8
X.680	ITU-T. *Information Technology — Abstract Syntax Notation One (ASN.1): Specification of Basic Notation.* July 2002. Identical to ISO/IEC 8824-1
X.690	ITU-T. *Information Technology — ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER).* July 2002. Identical to ISO/IEC 8825-1

4 Definitions

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

BATON	MISSI's BATON block cipher.

CAST	Entrust Technologies' proprietary symmetric block cipher.
CAST3	Entrust Technologies' proprietary symmetric block cipher.
CAST5	Another name for Entrust Technologies' symmetric block cipher CAST128. CAST128 is the preferred name.
CAST128	Entrust Technologies' symmetric block cipher.
CDMF	Commercial Data Masking Facility, a block encipherment method specified by International Business Machines Corporation and based on DES.
CMS	Cryptographic Message Syntax (see RFC 2630)
DES	Data Encryption Standard, as defined in FIPS PUB 46-3.
ECB	Electronic Codebook mode, as defined in FIPS PUB 81.
FASTHASH	MISSI's FASTHASH message-digesting algorithm.
IDEA	Ascom Systec's symmetric block cipher.
IV	Initialization Vector.
JUNIPER	MISSI's JUNIPER block cipher.
KEA	MISSI's Key Exchange Algorithm.
LYNKS	A smart card manufactured by SPYRUS.
MAC	Message Authentication Code.
MD2	RSA Security's MD2 message-digest algorithm, as defined in RFC 1319.
MD5	RSA Security's MD5 message-digest algorithm, as defined in RFC 1321.
PRF	Pseudo random function.
RSA	The RSA public-key cryptosystem.
RC2	RSA Security's RC2 symmetric block cipher.
RC4	RSA Security's proprietary RC4 symmetric stream cipher.
RC5	RSA Security's RC5 symmetric block cipher.
SET	The Secure Electronic Transaction protocol.
SHA-1	The (revised) Secure Hash Algorithm with a 160-bit message digest, as defined in FIPS PUB 180-2.

SKIPJACK MISSI's SKIPJACK block cipher.

UTF-8 Universal Character Set (UCS) transformation format (UTF) that represents ISO 10646 and UNICODE strings with a variable number of octets.

5 General overview

5.1 Introduction

Refer to PKCS#11 Base Functionality for basic pkcs#11 API functions and behaviour.

6 Mechanisms

A mechanism specifies precisely how a certain cryptographic process is to be performed.

The following table shows which Cryptoki mechanisms are supported by different cryptographic operations. For any particular token, of course, a particular operation may well support only a subset of the mechanisms listed. There is also no guarantee that a token which supports one mechanism for some 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 can also perform RSA encryption with **CKM_RSA_PKCS**.

Table 1, Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_FORTEZZA_TIMESTAMP		✓[2]					
CKM_KEA_KEY_PAIR_GEN					✓		
CKM_KEA_KEY_DERIVE							✓
CKM_RC2_KEY_GEN					✓		
CKM_RC2_ECB	✓					✓	
CKM_RC2_CBC	✓					✓	
CKM_RC2_CBC_PAD	✓					✓	
CKM_RC2_MAC_GENERAL		✓					
CKM_RC2_MAC		✓					
CKM_RC4_KEY_GEN					✓		
CKM_RC4	✓						
CKM_RC5_KEY_GEN					✓		
CKM_RC5_ECB	✓					✓	
CKM_RC5_CBC	✓					✓	
CKM_RC5_CBC_PAD	✓					✓	
CKM_RC5_MAC_GENERAL		✓					

Mechanism	Functions						
	Encrypt & Decrypt	**Sign & Verify**	**SR & VR[1]**	**Digest**	**Gen. Key/ Key Pair**	**Wrap & Unwrap**	**Derive**
CKM_RC5_MAC		✓					
CKM_DES_KEY_GEN					✓		
CKM_DES_ECB	✓					✓	
CKM_DES_CBC	✓					✓	
CKM_DES_CBC_PAD	✓					✓	
CKM_DES_MAC_GENERAL		✓					
CKM_DES_MAC		✓					
CKM_CAST_KEY_GEN					✓		
CKM_CAST_ECB	✓					✓	
CKM_CAST_CBC	✓					✓	
CKM_CAST_CBC_PAD	✓					✓	
CKM_CAST_MAC_GENERAL		✓					
CKM_CAST_MAC		✓					
CKM_CAST3_KEY_GEN					✓		
CKM_CAST3_ECB	✓					✓	
CKM_CAST3_CBC	✓					✓	
CKM_CAST3_CBC_PAD	✓					✓	
CKM_CAST3_MAC_GENERAL		✓					
CKM_CAST3_MAC		✓					
CKM_CAST128_KEY_GEN (CKM_CAST5_KEY_GEN)					✓		
CKM_CAST128_ECB (CKM_CAST5_ECB)	✓					✓	
CKM_CAST128_CBC (CKM_CAST5_CBC)	✓					✓	
CKM_CAST128_CBC_PAD (CKM_CAST5_CBC_PAD)	✓					✓	
CKM_CAST128_MAC_GENERAL (CKM_CAST5_MAC_GENERAL)		✓					
CKM_CAST128_MAC (CKM_CAST5_MAC)		✓					
CKM_IDEA_KEY_GEN					✓		
CKM_IDEA_ECB	✓					✓	
CKM_IDEA_CBC	✓					✓	
CKM_IDEA_CBC_PAD	✓					✓	
CKM_IDEA_MAC_GENERAL		✓					
CKM_IDEA_MAC		✓					
CKM_CDMF_KEY_GEN					✓		
CKM_CDMF_ECB	✓					✓	
CKM_CDMF_CBC	✓					✓	
CKM_CDMF_CBC_PAD	✓					✓	
CKM_CDMF_MAC_GENERAL		✓					
CKM_CDMF_MAC		✓					
CKM_SKIPJACK_KEY_GEN					✓		
CKM_SKIPJACK_ECB64	✓						
CKM_SKIPJACK_CBC64	✓						
CKM_SKIPJACK_OFB64	✓						
CKM_SKIPJACK_CFB64	✓						
CKM_SKIPJACK_CFB32	✓						
CKM_SKIPJACK_CFB16	✓						
CKM_SKIPJACK_CFB8	✓						

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SKIPJACK_WRAP						✓	
CKM_SKIPJACK_PRIVATE_WRAP						✓	
CKM_SKIPJACK_RELAYX						✓[3]	
CKM_BATON_KEY_GEN					✓		
CKM_BATON_ECB128	✓						
CKM_BATON_ECB96	✓						
CKM_BATON_CBC128	✓						
CKM_BATON_COUNTER	✓						
CKM_BATON_SHUFFLE	✓						
CKM_BATON_WRAP						✓	
CKM_JUNIPER_KEY_GEN					✓		
CKM_JUNIPER_ECB128	✓						
CKM_JUNIPER_CBC128	✓						
CKM_JUNIPER_COUNTER	✓						
CKM_JUNIPER_SHUFFLE	✓						
CKM_JUNIPER_WRAP						✓	
CKM_MD2				✓			
CKM_MD2_HMAC_GENERAL		✓					
CKM_MD2_HMAC		✓					
CKM_MD2_KEY_DERIVATION							✓
CKM_MD5				✓			
CKM_MD5_HMAC_GENERAL		✓					
CKM_MD5_HMAC		✓					
CKM_MD5_KEY_DERIVATION							✓
CKM_RIPEMD128				✓			
CKM_RIPEMD128_HMAC_GENERAL		✓					
CKM_RIPEMD128_HMAC		✓					
CKM_RIPEMD160				✓			
CKM_RIPEMD160_HMAC_GENERAL		✓					
CKM_RIPEMD160_HMAC		✓					
CKM_FASTHASH				✓			
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_MD5_CAST5_CBC)					✓		
CKM_PBE_SHA1_CAST128_CBC (CKM_PBE_SHA1_CAST5_CBC)					✓		
CKM_PBE_SHA1_RC4_128					✓		
CKM_PBE_SHA1_RC4_40					✓		
CKM_PBE_SHA1_RC2_128_CBC					✓		
CKM_PBE_SHA1_RC2_40_CBC					✓		
CKM_PBA_SHA1_WITH_SHA1_HMAC					✓		
CKM_PKCS5_PBKD2					✓		
CKM_KEY_WRAP_SET_OAEP						✓	
CKM_KEY_WRAP_LYNKS						✓	

[1] SR = SignRecover, VR = VerifyRecover.

[2] Single-part operations only.
[3] Mechanism can only be used for wrapping, not unwrapping.

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

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

6.1.1 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

Function	Key type	Input length	Output length
C_Sign[1]	DSA private key	20	40
C_Verify[1]	DSA public key	20, 40[2]	N/A

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

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

6.2 KEA

6.2.1 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
```

6.2.2 KEA mechanism parameters

♦ **CK_KEA_DERIVE_PARAMS; CK_KEA_DERIVE_PARAMS_PTR**

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

```
typedef struct CK_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_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	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
pPublicData	pointer to other party's KEA public key value

CK_KEA_DERIVE_PARAMS_PTR is a pointer to a **CK_KEA_DERIVE_PARAMS**.

6.2.3 KEA public key objects

KEA public key objects (object class **CKO_PUBLIC_KEY,** key type **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 3, KEA Public Key Object Attributes

Attribute	Data type	Meaning
CKA_PRIME[1,3]	Big integer	Prime *p* (512 to 1024 bits, in steps of 64 bits)
CKA_SUBPRIME[1,3]	Big integer	Subprime *q* (160 bits)
CKA_BASE[1,3]	Big integer	Base *g* (512 to 1024 bits, in steps of 64 bits)
CKA_VALUE[1,4]	Big integer	Public value *y*

- Refer to [PKCS #11-B] table **Error! Reference source not found.** for footnotes

The **CKA_PRIME**, **CKA_SUBPRIME** and **CKA_BASE** attribute values are collectively the “KEA domain parameters”.

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

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

6.2.4 KEA private key objects

KEA private key objects (object class **CKO_PRIVATE_KEY,** key type **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 4, KEA Private Key Object Attributes

Attribute	Data type	Meaning
CKA_PRIME[1,4,6]	Big integer	Prime *p* (512 to 1024 bits, in steps of 64 bits)
CKA_SUBPRIME[1,4,6]	Big integer	Subprime *q* (160 bits)
CKA_BASE[1,4,6]	Big integer	Base *g* (512 to 1024 bits, in steps of 64 bits)
CKA_VALUE[1,4,6,7]	Big integer	Private value *x*

- Refer to [PKCS #11-B] table **Error! Reference source not found.** for footnotes

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:

```
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)},
  {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)}
};
```

6.2.5 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.

6.2.6 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 can 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 one (1) for the KEA parameter R_b (*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 5, KEA Parameter Values and Operations

Value of boolean *isSender*	**Value of big integer** *pRandomB*	**Token Action** (after checking parameter and template values)
CK_TRUE	0	Compute KEA R_a value, store it in *pRandomA*, return CKR_OK. No derived key object is created.
CK_TRUE	1	Compute KEA R_a value, store it in *pRandomA*, derive key value using e-mail mode, create key object, return CKR_OK.
CK_TRUE	>1	Compute KEA R_a value, store it in *pRandomA*, derive key value using full mode, create key object, return CKR_OK.
CK_FALSE	0	Compute KEA R_b value, store it in *pRandomB*, return CKR_OK. No derived key object is created.
CK_FALSE	1	Derive key value using e-mail mode, create key object, return CKR_OK.
CK_FALSE	>1	Derive key value using full mode, create key object, return CKR_OK.

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 (R_a or R_b, 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[†]:

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

[†] Note that the rules regarding the **CKA_SENSITIVE**, **CKA_EXTRACTABLE**, **CKA_ALWAYS_SENSITIVE**, and **CKA_NEVER_EXTRACTABLE** attributes have changed in version 2.11 to match the policy used by other key derivation mechanisms such as **CKM_SSL3_MASTER_KEY_DERIVE**.

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

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

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

6.3 RC2

RC2 is a block cipher which is trademarked by RSA Security. It has a variable keysize and an additional parameter, the “effective number of bits in the RC2 search space”, which can 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.

6.3.1 Definitions

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
```

6.3.2 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 6, RC2 Secret Key Object Attributes

Attribute	Data type	Meaning
CKA_VALUE[1,4,6,7]	Byte array	Key value (1 to 128 bytes)
CKA_VALUE_LEN[2,3]	CK_ULONG	Length in bytes of key value

- Refer to [PKCS #11-B] table **Error! Reference source not found.** for footnotes

The following is a sample template for creating an RC2 secret key object:

```
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)}
};
```

6.3.3 RC2 mechanism parameters

- **CK_RC2_PARAMS; CK_RC2_PARAMS_PTR**

CK_RC2_PARAMS provides the parameters to the **CKM_RC2_ECB** and **CKM_RC2_MAC** mechanisms. It holds the effective number of bits in the RC2 search space. It is defined as follows:

```
typedef CK_ULONG CK_RC2_PARAMS;
```

CK_RC2_PARAMS_PTR is a pointer to a **CK_RC2_PARAMS**.

- **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:

```
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**.

- ♦ **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:

```
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

CK_RC2_MAC_GENERAL_PARAMS_PTR is a pointer to a **CK_RC2_MAC_GENERAL_PARAMS**.

6.3.4 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.

6.3.5 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 can wrap and unwrap any secret key. Of course, a particular token may not be able to wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to 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, 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 7, RC2-ECB: Key And Data Length

Function	Key type	Input length	Output length	Comments
C_Encrypt	RC2	multiple of 8	same as input length	no final part
C_Decrypt	RC2	multiple of 8	same as input length	no final part
C_WrapKey	RC2	any	input length rounded up to multiple of 8	
C_UnwrapKey	RC2	multiple of 8	determined by type of key being unwrapped or CKA_VALUE_LEN	

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

6.3.6 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 can wrap and unwrap any secret key. Of course, a particular token may not be able to wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to 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, 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 8, RC2-CBC: Key And Data Length

Function	Key type	Input length	Output length	Comments
C_Encrypt	RC2	multiple of 8	same as input length	no final part
C_Decrypt	RC2	multiple of 8	same as input length	no final part
C_WrapKey	RC2	any	input length rounded up to multiple of 8	
C_UnwrapKey	RC2	multiple of 8	determined by type of key being unwrapped or CKA_VALUE_LEN	

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

6.3.7 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 can wrap and unwrap RSA, Diffie-Hellman, X9.42 Diffie-Hellman, EC (also related to ECDSA) and DSA private keys (see Section **Error! Reference source not found.** 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

Function	Key type	Input length	Output length
C_Encrypt	RC2	any	input length rounded up to multiple of 8
C_Decrypt	RC2	multiple of 8	between 1 and 8 bytes shorter than input length
C_WrapKey	RC2	any	input length rounded up to multiple of 8
C_UnwrapKey	RC2	multiple of 8	between 1 and 8 bytes shorter than input length

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

6.3.8 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 authentication 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 10, General-length RC2-MAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	RC2	any	0-8, as specified in parameters
C_Verify	RC2	any	0-8, as specified in parameters

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

6.3.9 RC2-MAC

RC2-MAC, denoted by **CKM_RC2_MAC**, is a special case of the general-length RC2-MAC mechanism (see Section 6.3.8). 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 always produces and verifies 4-byte MACs.

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

Table 11, RC2-MAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	RC2	any	4
C_Verify	RC2	any	4

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

6.4 RC4

6.4.1 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
```

6.4.2 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

Attribute	**Data type**	**Meaning**
CKA_VALUE[1,4,6,7]	Byte array	Key value (1 to 256 bytes)
CKA_VALUE_LEN[2,3,6]	CK_ULONG	Length in bytes of key value

- Refer to [PKCS #11-B] table **Error! Reference source not found.** for footnotes

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

```
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)}
};
```

6.4.3 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 CK**A_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.

6.4.4 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

Function	Key type	Input length	Output length	Comments
C_Encrypt	RC4	any	same as input length	no final part
C_Decrypt	RC4	any	same as input length	no final part

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of RC4 key sizes, in bits.

6.5 RC5

RC5 is a parametrizable 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 always equal to twice its wordsize.

6.5.1 Definitions

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
CKM_RC5_CBC_PAD
```

6.5.2 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

Attribute	Data type	Meaning
CKA_VALUE[1,4,6,7]	Byte array	Key value (0 to 255 bytes)
CKA_VALUE_LEN[2,3,6]	CK_ULONG	Length in bytes of key value

- Refer to [PKCS #11-B] table **Error! Reference source not found.** for footnotes

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

```
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;
CK_ATTRIBUTE template[] = {
  {CKA_CLASS, &class, sizeof(class)},
  {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
  {CKA_TOKEN, &true, sizeof(true)},
  {CKA_LABEL, label, sizeof(label)-1},
  {CKA_ENCRYPT, &true, sizeof(true)},
  {CKA_VALUE, value, sizeof(value)}
};
```

6.5.3 RC5 mechanism parameters

♦ **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:

```
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**.

♦ **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:

```
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**.

- **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_PARAMS {
  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**.

6.5.4 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.

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

This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with 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

Function	**Key type**	**Input length**	**Output length**	**Comments**
C_Encrypt	RC5	multiple of blocksize	same as input length	no final part
C_Decrypt	RC5	multiple of blocksize	same as input length	no final part
C_WrapKey	RC5	any	input length rounded up to multiple of blocksize	
C_UnwrapKey	RC5	multiple of blocksize	determined by type of key being unwrapped or CKA_VALUE_LEN	

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of RC5 key sizes, in bytes.

6.5.6 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 can wrap and unwrap any secret key. Of course, a particular token may not be able to wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to 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, 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 16, RC5-CBC: Key And Data Length

Function	Key type	Input length	Output length	Comments
C_Encrypt	RC5	multiple of blocksize	same as input length	no final part
C_Decrypt	RC5	multiple of blocksize	same as input length	no final part
C_WrapKey	RC5	any	input length rounded up to multiple of blocksize	
C_UnwrapKey	RC5	multiple of blocksize	determined by type of key being unwrapped or CKA_VALUE_LEN	

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of RC5 key sizes, in bytes.

6.5.7 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 can wrap and unwrap RSA, Diffie-Hellman, X9.42 Diffie-Hellman, EC (also related to ECDSA) and DSA private keys (see Section **Error! Reference source not found.** 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 17, RC5-CBC with PKCS Padding: Key And Data Length

Function	Key type	Input length	Output length
C_Encrypt	RC5	any	input length rounded up to multiple of blocksize
C_Decrypt	RC5	multiple of blocksize	between 1 and blocksize bytes shorter than input length
C_WrapKey	RC5	any	input length rounded up to multiple of blocksize
C_UnwrapKey	RC5	multiple of blocksize	between 1 and blocksize bytes shorter than input length

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of RC5 key sizes, in bytes.

6.5.8 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 18, General-length RC2-MAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	RC5	any	0-blocksize, as specified in parameters
C_Verify	RC5	any	0-blocksize, as specified in parameters

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of RC5 key sizes, in bytes.

6.5.9 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 always 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

Function	Key type	Data length	Signature length
C_Sign	RC5	any	RC5 wordsize = $\lfloor$blocksize/2$\rfloor$
C_Verify	RC5	any	RC5 wordsize = $\lfloor$blocksize/2$\rfloor$

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of RC5 key sizes, in bytes.

6.6 General block cipher

For brevity's sake, the mechanisms for the DES, CAST, CAST3, CAST128 (CAST5), IDEA, and CDMF block ciphers will be described together here. Each of these ciphers has the following mechanisms, which will be described in a templatized form.

6.6.1 Definitions

This section defines the key types "CKK_DES", "CKK_CAST", "CKK_CAST3", "CKK_CAST5" (deprecated in v2.11), "CKK_CAST128", "CKK_IDEA" and "CKK_CDMF" for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

```
CKM_DES_KEY_GEN
CKM_DES_ECB
CKM_DES_CBC
CKM_DES_MAC
CKM_DES_MAC_GENERAL
CKM_DES_CBC_PAD
CKM_CDMF_KEY_GEN
CKM_CDMF_ECB
CKM_CDMF_CBC
CKM_CDMF_MAC
CKM_CDMF_MAC_GENERAL
CKM_CDMF_CBC_PAD
CKM_DES_OFB64
CKM_DES_OFB8
CKM_DES_CFB64
CKM_DES_CFB8
CKM_CAST_KEY_GEN
CKM_CAST_ECB
CKM_CAST_CBC
CKM_CAST_MAC
CKM_CAST_MAC_GENERAL
CKM_CAST_CBC_PAD
CKM_CAST3_KEY_GEN
CKM_CAST3_ECB
CKM_CAST3_CBC
CKM_CAST3_MAC
CKM_CAST3_MAC_GENERAL
CKM_CAST3_CBC_PAD
CKM_CAST5_KEY_GEN
CKM_CAST128_KEY_GEN
CKM_CAST5_ECB
CKM_CAST128_ECB
CKM_CAST5_CBC
```

```
CKM_CAST128_CBC
CKM_CAST5_MAC
CKM_CAST128_MAC
CKM_CAST5_MAC_GENERAL
CKM_CAST128_MAC_GENERAL
CKM_CAST5_CBC_PAD
CKM_CAST128_CBC_PAD
CKM_IDEA_KEY_GEN
CKM_IDEA_ECB
CKM_IDEA_CBC
CKM_IDEA_MAC
CKM_IDEA_MAC_GENERAL
CKM_IDEA_CBC_PAD
```

6.6.2 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 20, DES Secret Key Object

Attribute	Data type	Meaning
CKA_VALUE[1,4,6,7]	Byte array	Key value (always 8 bytes long)

- Refer to [PKCS #11-B] table **Error! Reference source not found.** for footnotes

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

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

```
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.

6.6.3 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

Attribute	Data type	Meaning
CKA_VALUE[1,4,6,7]	Byte array	Key value (1 to 8 bytes)
CKA_VALUE_LEN[2,3,6]	CK_ULONG	Length in bytes of key value

- Refer to [PKCS #11-B] table **Error! Reference source not found.** for footnotes

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

```
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_ENCRYPT, &true, sizeof(true)},
  {CKA_VALUE, value, sizeof(value)}
};
```


6.6.4 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 defined for this object class:

Table 22, CAST3 Secret Key Object Attributes

Attribute	Data type	Meaning
CKA_VALUE[1,4,6,7]	Byte array	Key value (1 to 8 bytes)
CKA_VALUE_LEN[2,3,6]	CK_ULONG	Length in bytes of key value

- Refer to [PKCS #11-B] table **Error! Reference source not found.** for footnotes

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

```
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_ENCRYPT, &true, sizeof(true)},
  {CKA_VALUE, value, sizeof(value)}
};
```

6.6.5 CAST128 (CAST5) secret key objects

CAST128 (also known as CAST5) secret key objects (object class **CKO_SECRET_KEY,** key type CKK_CAST128 or **CKK_CAST5**) hold CAST128 keys. The following table defines the CAST128 secret key object attributes, in addition to the common attributes defined for this object class:

Table 23, CAST128 (CAST5) Secret Key Object Attributes

Attribute	Data type	Meaning
CKA_VALUE[1,4,6,7]	Byte array	Key value (1 to 16 bytes)
CKA_VALUE_LEN[2,3,6]	CK_ULONG	Length in bytes of key value

- Refer to [PKCS #11-B] table **Error! Reference source not found.** for footnotes

The following is a sample template for creating a CAST128 (CAST5) secret key object:

```
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_ENCRYPT, &true, sizeof(true)},
  {CKA_VALUE, value, sizeof(value)}
};
```

6.6.6 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 defined for this object class:

Table 24, IDEA Secret Key Object

Attribute	Data type	Meaning
CKA_VALUE[1,4,6,7]	Byte array	Key value (always 16 bytes long)

- Refer to [PKCS #11-B] table **Error! Reference source not found.** for footnotes

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

```
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_ENCRYPT, &true, sizeof(true)},
  {CKA_VALUE, value, sizeof(value)}
};
```

6.6.7 CDMF secret key objects

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

Table 25, CDMF Secret Key Object

Attribute	Data type	Meaning
CKA_VALUE[1,4,6,7]	Byte array	Key value (always 8 bytes long)

- Refer to [PKCS #11-B] table **Error! Reference source not found.** for footnotes

CDMF keys must always 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 will return an error.

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

```
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)}
};
```

6.6.8 General block cipher mechanism parameters

- **CK_MAC_GENERAL_PARAMS; CK_MAC_GENERAL_PARAMS_PTR**

CK_MAC_GENERAL_PARAMS provides the parameters to the general-length MACing mechanisms of the DES, DES3 (triple-DES), CAST, CAST3, CAST128 (CAST5), 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 will produce. It is defined as follows:

```
typedef CK_ULONG CK_MAC_GENERAL_PARAMS;
```

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

6.6.9 General block cipher key generation

Cipher <NAME> has a key generation mechanism, "<NAME> key generation", denoted **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 (CAST5) 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 or may not be used. The CAST, CAST3, and CAST128 (CAST5) 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 are not used.

6.6.10 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 can wrap and unwrap any secret key. Of course, a particular token may not be able to wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with 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 type 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

Function	**Key type**	**Input length**	**Output length**	**Comments**
C_Encrypt	<NAME>	multiple of blocksize	same as input length	no final part
C_Decrypt	<NAME>	multiple of blocksize	same as input length	no final part
C_WrapKey	<NAME>	any	input length rounded up to multiple of blocksize	
C_UnwrapKey	<NAME>	any	determined by type of key being unwrapped or CKA_VALUE_LEN	

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure may or may not be used. The CAST, CAST3, and CAST128 (CAST5) 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.

6.6.11 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

Function	Key type	Input length	Output length	Comments
C_Encrypt	<NAME>	multiple of blocksize	same as input length	no final part
C_Decrypt	<NAME>	multiple of blocksize	same as input length	no final part
C_WrapKey	<NAME>	any	input length rounded up to multiple of blocksize	
C_UnwrapKey	<NAME>	any	determined by type of key being unwrapped or CKA_VALUE_LEN	

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure may or may not be used. The CAST, CAST3, and CAST128 (CAST5) 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.

6.6.12 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 can wrap and unwrap RSA, Diffie-Hellman, X9.42 Diffie-Hellman, EC (also related to ECDSA) and DSA private keys (see Section **Error! Reference source not found.** 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 28, General Block Cipher CBC with PKCS Padding: Key And Data Length

Function	Key type	Input length	Output length
C_Encrypt	<NAME>	any	input length rounded up to multiple of blocksize
C_Decrypt	<NAME>	multiple of blocksize	between 1 and blocksize bytes shorter than input length
C_WrapKey	<NAME>	any	input length rounded up to multiple of blocksize
C_UnwrapKey	<NAME>	multiple of blocksize	between 1 and blocksize bytes shorter than input length

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure may or may not be used. The CAST, CAST3, and CAST128 (CAST5) 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.

6.6.13 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

Function	Key type	Data length	Signature length
C_Sign	<NAME>	any	0-blocksize, depending on parameters
C_Verify	<NAME>	any	0-blocksize, depending on parameters

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure may or may not be used. The CAST, CAST3, and CAST128 (CAST5) 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.

6.6.14 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 always 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

Function	Key type	Data length	Signature length
C_Sign	<NAME>	any	$\lfloor$blocksize/2$\rfloor$
C_Verify	<NAME>	any	$\lfloor$blocksize/2$\rfloor$

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure may or may not be used. The CAST, CAST3, and CAST128 (CAST5) 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.

6.7 SKIPJACK

6.7.1 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
CKM_SKIPJACK_CFB32
CKM_SKIPJACK_CFB16
CKM_SKIPJACK_CFB8
CKM_SKIPJACK_WRAP
CKM_SKIPJACK_PRIVATE_WRAP
CKM_SKIPJACK_RELAYX
```

6.7.2 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 key object attributes, in addition to the common attributes defined for this object class:

Table 31, SKIPJACK Secret Key Object

Attribute	Data type	Meaning
CKA_VALUE[1,4,6,7]	Byte array	Key value (always 12 bytes long)

- Refer to [PKCS #11-B] table **Error! Reference source not found.** 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 will return an error.

It is not clear that any tokens exist (or will ever 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:

```
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:

```
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)}
};
```


6.7.3 SKIPJACK Mechanism parameters

- **CK_SKIPJACK_PRIVATE_WRAP_PARAMS; CK_SKIPJACK_PRIVATE_WRAP_PARAMS_PTR**

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

```
typedef struct CK_SKIPJACK_PRIVATE_WRAP_PARAMS {
  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_PARAMS;
```

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
pRandomA	pointer to Ra data
pPrimeP	pointer to Prime, p, value
pBaseG	pointer to Base, g, value
pSubprimeQ	pointer to Subprime, q, value

CK_SKIPJACK_PRIVATE_WRAP_PARAMS_PTR is a pointer to a **CK_PRIVATE_WRAP_PARAMS**.

- **CK_SKIPJACK_RELAYX_PARAMS; CK_SKIPJACK_RELAYX_PARAMS_PTR**

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

```
typedef struct CK_SKIPJACK_RELAYX_PARAMS {
  CK_ULONG ulOldWrappedXLen;
  CK_BYTE_PTR pOldWrappedX;
  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:

ulOldWrappedXLen	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**.

6.7.4 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.

6.7.5 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 cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

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

Table 32, SKIPJACK-ECB64: Data and Length

Function	Key type	Input length	Output length	Comments
C_Encrypt	SKIPJACK	multiple of 8	same as input length	no final part
C_Decrypt	SKIPJACK	multiple of 8	same as input length	no final part

6.7.6 SKIPJACK-CBC64

SKIPJACK-CBC64, denoted **CKM_SKIPJACK_CBC64**, is a mechanism for single- and multiple-part encryption and decryption with SKIPJACK in 64-bit cipher-block chaining 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 cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

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

Table 33, SKIPJACK-CBC64: Data and Length

Function	Key type	Input length	Output length	Comments
C_Encrypt	SKIPJACK	multiple of 8	same as input length	no final part
C_Decrypt	SKIPJACK	multiple of 8	same as input length	no final part

6.7.7 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 cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

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

Table 34, SKIPJACK-OFB64: Data and Length

Function	Key type	Input length	Output length	Comments
C_Encrypt	SKIPJACK	multiple of 8	same as input length	no final part
C_Decrypt	SKIPJACK	multiple of 8	same as input length	no final part

6.7.8 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 cannot specify a particular IV when encrypting. It can, 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

Function	Key type	Input length	Output length	Comments
C_Encrypt	SKIPJACK	multiple of 8	same as input length	no final part
C_Decrypt	SKIPJACK	multiple of 8	same as input length	no final part

6.7.9 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 cannot specify a particular IV when encrypting. It can, 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

Function	Key type	Input length	Output length	Comments
C_Encrypt	SKIPJACK	multiple of 4	same as input length	no final part
C_Decrypt	SKIPJACK	multiple of 4	same as input length	no final part

6.7.10 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 cannot specify a particular IV when encrypting. It can, 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

Function	Key type	Input length	Output length	Comments
C_Encrypt	SKIPJACK	multiple of 4	same as input length	no final part
C_Decrypt	SKIPJACK	multiple of 4	same as input length	no final part

6.7.11 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 cannot specify a particular IV when encrypting. It can, 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

Function	Key type	Input length	Output length	Comments
C_Encrypt	SKIPJACK	multiple of 4	same as input length	no final part
C_Decrypt	SKIPJACK	multiple of 4	same as input length	no final part

6.7.12 SKIPJACK-WRAP

The SKIPJACK-WRAP mechanism, denoted **CKM_SKIPJACK_WRAP**, is used to wrap and unwrap a secret key (MEK). It can wrap or unwrap SKIPJACK, BATON, and JUNIPER keys.

It does not have a parameter.

6.7.13 SKIPJACK-PRIVATE-WRAP

The SKIPJACK-PRIVATE-WRAP mechanism, denoted **CKM_SKIPJACK_PRIVATE_WRAP**, is used to wrap and unwrap a private key. It can wrap KEA and DSA private keys.

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

6.7.14 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 6.7.13).

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.

6.8 BATON

6.8.1 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:

```
CKM_BATON_KEY_GEN
CKM_BATON_ECB128
CKM_BATON_ECB96
CKM_BATON_CBC128
CKM_BATON_COUNTER
CKM_BATON_SHUFFLE
CKM_BATON_WRAP
```

6.8.2 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

Attribute	**Data type**	**Meaning**
CKA_VALUE[1,4,6,7]	Byte array	Key value (always 40 bytes long)

- Refer to [PKCS #11-B] table **Error! Reference source not found.** 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 will 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:

```
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_ENCRYPT, &true, sizeof(true)},
  {CKA_VALUE, value, sizeof(value)}
};
```

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


```
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)}
};
```

6.8.3 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.

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

6.8.4 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 cannot specify a particular IV when encrypting. It can, 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

Function	Key type	Input length	Output length	Comments
C_Encrypt	BATON	multiple of 16	same as input length	no final part
C_Decrypt	BATON	multiple of 16	same as input length	no final part

6.8.5 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 cannot specify a particular IV when encrypting. It can, 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

Function	Key type	Input length	Output length	Comments
C_Encrypt	BATON	multiple of 12	same as input length	no final part
C_Decrypt	BATON	multiple of 12	same as input length	no final part

6.8.6 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 cannot specify a particular IV when encrypting. It can, 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: Data and Length

Function	Key type	Input length	Output length	Comments
C_Encrypt	BATON	multiple of 16	same as input length	no final part
C_Decrypt	BATON	multiple of 16	same as input length	no final part

6.8.7 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 cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

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

Table 43, BATON-COUNTER: Data and Length

Function	Key type	Input length	Output length	Comments
C_Encrypt	BATON	multiple of 16	same as input length	no final part
C_Decrypt	BATON	multiple of 16	same as input length	no final part

6.8.8 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 cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

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

Table 44, BATON-SHUFFLE: Data and Length

Function	Key type	Input length	Output length	Comments
C_Encrypt	BATON	multiple of 16	same as input length	no final part
C_Decrypt	BATON	multiple of 16	same as input length	no final part

6.8.9 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 can 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.

6.9 JUNIPER

6.9.1 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
```

6.9.2 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 JUNIPER secret key object attributes, in addition to the common attributes defined for this object class:

Table 45, JUNIPER Secret Key Object

Attribute	Data type	Meaning
CKA_VALUE[1,4,6,7]	Byte array	Key value (always 40 bytes long)

- Refer to [PKCS #11-B] table **Error! Reference source not found.** for footnotes

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

It is not clear that any tokens exist (or will ever exist) which permit an application to create a JUNIPER 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:

```
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_ENCRYPT, &true, sizeof(true)},
  {CKA_VALUE, value, sizeof(value)}
};
```

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

```
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)}
};
```

6.9.3 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.

6.9.4 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 cannot specify a particular IV when encrypting. It can, 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 46, JUNIPER-ECB128: Data and Length

Function	Key type	Input length	Output length	Comments
C_Encrypt	JUNIPER	multiple of 16	same as input length	no final part
C_Decrypt	JUNIPER	multiple of 16	same as input length	no final part

6.9.5 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 cannot specify a particular IV when encrypting. It can, 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 47, JUNIPER-CBC128: Data and Length

Function	Key type	Input length	Output length	Comments
C_Encrypt	JUNIPER	multiple of 16	same as input length	no final part
C_Decrypt	JUNIPER	multiple of 16	same as input length	no final part

6.9.6 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 cannot specify a particular IV when encrypting. It can, 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 48, JUNIPER-COUNTER: Data and Length

Function	Key type	Input length	Output length	Comments
C_Encrypt	JUNIPER	multiple of 16	same as input length	no final part
C_Decrypt	JUNIPER	multiple of 16	same as input length	no final part

6.9.7 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 cannot specify a particular IV when encrypting. It can, 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

Function	Key type	Input length	Output length	Comments
C_Encrypt	JUNIPER	multiple of 16	same as input length	no final part
C_Decrypt	JUNIPER	multiple of 16	same as input length	no final part

6.9.8 JUNIPER WRAP

The JUNIPER wrap and unwrap mechanism, denoted **CKM_JUNIPER_WRAP**, is a function used to wrap and unwrap an MEK. It can 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.

6.10 MD2

6.10.1 Definitions

Mechanisms:

```
CKM_MD2
CKM_MD2_HMAC
CKM_MD2_HMAC_GENERAL
CKM_MD2_KEY_DERIVATION
```

6.10.2 MD2 digest

The MD2 mechanism, denoted **CKM_MD2**, is a mechanism for message digesting, following the MD2 message-digest algorithm defined in RFC 1319.

It does not have a parameter.

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

Table 50, MD2: Data Length

Function	Data length	Digest length
C_Digest	any	16

6.10.3 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 will be taken from the start of the full 16-byte HMAC output.

Table 51, General-length MD2-HMAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	generic secret	any	0-16, depending on parameters
C_Verify	generic secret	any	0-16, depending on parameters

6.10.4 MD2-HMAC

The MD2-HMAC mechanism, denoted **CKM_MD2_HMAC**, is a special case of the general-length MD2-HMAC mechanism in Section 6.10.3.

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

6.10.5 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 derived secret key.

- If no length or key type is provided in the template, then the key produced by this mechanism will be a generic secret key. Its length will be 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 will be a generic secret key of the specified length.

- If no length was provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn’t, an error will be returned.

- If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism will be of the specified type and length.

If a DES, DES2, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more than 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 can both be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some default value.

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

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

6.11 MD5

6.11.1 Definitions

Mechanisms:

```
CKM_MD5
CKM_MD5_HMAC
CKM_MD5_HMAC_GENERAL
CKM_MD5_KEY_DERIVATION
```

6.11.2 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 52, MD5: Data Length

Function	Data length	Digest length
C_Digest	any	16

6.11.3 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 will be taken from the start of the full 16-byte HMAC output.

Table 53, General-length MD5-HMAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	generic secret	any	0-16, depending on parameters
C_Verify	generic secret	any	0-16, depending on parameters

6.11.4 MD5-HMAC

The MD5-HMAC mechanism, denoted **CKM_MD5_HMAC**, is a special case of the general-length MD5-HMAC mechanism in Section 6.11.3.

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

6.11.5 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 will be a generic secret key. Its length will 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 will be a generic secret key of the specified length.

- If no length was provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn't, an error will be returned.

- If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism will be of the specified type and length.

If a DES, DES2, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more than 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 can both be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some default value.

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

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

6.12 FASTHASH

6.12.1 Definitions

Mechanisms:

```
CKM_FASTHASH
```

6.12.2 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 54, FASTHASH: Data Length

Function	Input length	Digest length
C_Digest	any	40

6.13 PKCS #5 and PKCS #5-style password-based encryption (PBE)

The mechanisms in this section are for generating keys and IVs for performing password-based encryption. The method used to generate keys and IVs is specified in PKCS #5.

6.13.1 Definitions

Mechanisms:

```
CKM_PBE_MD2_DES_CBC
CKM_PBE_MD5_DES_CBC
CKM_PBE_MD5_CAST_CBC
CKM_PBE_MD5_CAST3_CBC
CKM_PBE_MD5_CAST5_CBC
CKM_PBE_MD5_CAST128_CBC
CKM_PBE_SHA1_CAST5_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
```

6.13.2 Password-based encryption/authentication mechanism parameters

- **CK_PBE_PARAMS; CK_PBE_PARAMS_PTR**

CK_PBE_PARAMS is a structure which provides all of the necessary information required by the CKM_PBE mechanisms (see PKCS #5 and PKCS #12 for information on the PBE generation mechanisms) and the CKM_PBA_SHA1_WITH_SHA1_HMAC mechanism. It is defined as follows:

```
typedef struct CK_PBE_PARAMS {
  CK_BYTE_PTR pInitVector;
  CK_UTF8CHAR_PTR pPassword;
  CK_ULONG ulPasswordLen;
  CK_BYTE_PTR pSalt;
  CK_ULONG ulSaltLen;
  CK_ULONG ulIteration;
} CK_PBE_PARAMS;
```

The fields of the structure have the following meanings:

pInitVector	pointer to the location that receives the 8-byte initialization vector (IV), if an IV is required;
pPassword	points to the password to be used in the PBE key generation;
ulPasswordLen	length in bytes of the password information;
pSalt	points to the salt to be used in the PBE key generation;
ulSaltLen	length in bytes of the salt information;
ulIteration	number of iterations required for the generation.

CK_PBE_PARAMS_PTR is a pointer to a **CK_PBE_PARAMS**.

6.13.3 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 will receive the 8-byte IV generated by the mechanism.

6.13.4 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 will receive the 8-byte IV generated by the mechanism.

6.13.5 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 will receive 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.

6.13.6 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 will receive 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.

6.13.7 MD5-PBE for CAST128-CBC (CAST5-CBC)

MD5-PBE for CAST128-CBC (CAST5-CBC), denoted **CKM_PBE_MD5_CAST128_CBC** or **CKM_PBE_MD5_CAST5_CBC**, is a mechanism used for generating a CAST128 (CAST5) 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 will receive the 8-byte IV generated by the mechanism.

The length of the CAST128 (CAST5) 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.

6.13.8 SHA-1-PBE for CAST128-CBC (CAST5-CBC)

SHA-1-PBE for CAST128-CBC (CAST5-CBC), denoted **CKM_PBE_SHA1_CAST128_CBC** or **CKM_PBE_SHA1_CAST5_CBC**, is a mechanism used for generating a CAST128 (CAST5) secret key and an IV from a password and a salt value by 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 will receive the 8-byte IV generated by the mechanism.

The length of the CAST128 (CAST5) 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.

6.14 PKCS #12 password-based encryption/authentication mechanisms

The mechanisms in this section are for generating keys and IVs for performing password-based encryption or authentication. The method used to generate keys and IVs is based on a method that was specified in PKCS #12.

We specify here a general method for producing various types of pseudo-random bits from a password, *p*; a string of salt bits, *s*; and an iteration count, *c*. The "type" of pseudo-random bits to be produced is identified by an identification byte, *ID*, the meaning of which will be discussed later.

Let H be a hash function built around a compression function $f\colon \boldsymbol{Z}_2^u \times \boldsymbol{Z}_2^v \rightarrow \boldsymbol{Z}_2^u$ (that is, H has a chaining variable and output of length u bits, and the message input to the compression function of H is v bits). For MD2 and MD5, u=128 and v=512; for SHA-1, u=160 and v=512.

We assume here that u and v are both multiples of 8, as are the lengths in bits of the password and salt strings and the number n of pseudo-random bits required. In addition, u and v are of course nonzero.

1. Construct a string, *D* (the "diversifier"), by concatenating $v/8$ copies of *ID*.

2. Concatenate copies of the salt together to create a string *S* of length $v \cdot \lceil s/v \rceil$ 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 $v \cdot \lceil p/v \rceil$ bits (the final copy of the password may be truncated to create *P*). Note that if the password is the empty string, then so is *P*.

4. Set $I=S||P$ to be the concatenation of *S* and *P*.

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

6. For i=1, 2, …, j, do the following:

a) Set A_i=$H^c(D||I)$, the c^{th} hash of $D||I$. That is, compute the hash of $D||I$; compute the hash of that hash; etc.; continue in this fashion until a total of c hashes have been computed, each on the result of the previous hash.

b) Concatenate copies of A_i to create a string B of length v bits (the final copy of A_i may be truncated to create B).

c) Treating I as a concatenation I_0, I_1, …, I_{k-1} of v-bit blocks, where $k=\lceil s/v \rceil+\lceil p/v \rceil$, modify I by setting $I_j=(I_j+B+1) \bmod 2^v$ for each j. To perform this addition, treat each v-bit block as a binary number represented most-significant bit first.

7. Concatenate A_1, A_2, …, A_j together to form a pseudo-random bit string, A.

8. Use the first n bits of A as the output of this entire process.

When the password-based encryption mechanisms presented in this section are used to generate a key and IV (if needed) from a password, salt, and an iteration count, the above algorithm is used. To generate a key, the identifier byte *ID* is set to the value 1; to generate an IV, the identifier byte *ID* is set to the value 2.

When the password based authentication mechanism presented in this section is used to generate a key from a password, salt, and an iteration count, the above algorithm is used. The identifier byte *ID* is set to the value 3.

6.14.1 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 will receive 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.

6.14.2 SHA-1-PBE for 40-bit RC4

SHA-1-PBE for 40-bit RC4, denoted **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 **CK_PBE_PARAMS** structure. The parameter specifies the input information for the key generation process. The parameter also has a field to hold the location of an application-supplied buffer which will receive an IV; for this mechanism, the contents of this field are ignored, since RC4 does not require an IV.

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

6.14.3 SHA-1-PBE for 128-bit RC2-CBC

SHA-1-PBE for 128-bit RC2-CBC, denoted **CKM_PBE_SHA1_RC2_128_CBC**, is a mechanism used for generating a 128-bit RC2 secret key and IV from a password and a salt value by using the SHA-1 digest algorithm and an iteration count. The method used to generate the key and IV is described above.

It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the key generation process and the location of the application-supplied buffer which will receive the 8-byte IV generated by the mechanism.

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 `pbeWithSHA1And128BitRC2-CBC`.

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

6.14.4 SHA-1-PBE for 40-bit RC2-CBC

SHA-1-PBE for 40-bit RC2-CBC, denoted **CKM_PBE_SHA1_RC2_40_CBC**, is a mechanism used for generating a 40-bit RC2 secret key and IV from a password and a salt value by using the SHA-1 digest algorithm and an iteration count. The method used to generate the key and IV is described above.

It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the key generation process and the location of the application-supplied buffer which will receive the 8-byte IV generated by the mechanism.

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 `pbeWithSHA1And40BitRC2-CBC`.

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

6.15 RIPE-MD

6.15.1 Definitions

Mechanisms:

```
CKM_RIPEMD128
CKM_RIPEMD128_HMAC
CKM_RIPEMD128_HMAC_GENERAL
CKM_RIPEMD160
CKM_RIPEMD160_HMAC
CKM_RIPEMD160_HMAC_GENERAL
```

6.15.2 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 55, RIPE-MD 128: Data Length

Function	Data length	Digest length
C_Digest	any	16

6.15.3 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 will be taken from the start of the full 16-byte HMAC output.

Table 56, General-length RIPE-MD 128-HMAC:

Function	Key type	Data length	Signature length
C_Sign	generic secret	any	0-16, depending on parameters
C_Verify	generic secret	any	0-16, depending on

			parameters

6.15.4 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 6.15.3.

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

6.15.5 RIPE-MD 160

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

It does not have a parameter.

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

Table 57, RIPE-MD 160: Data Length

Function	Data length	Digest length
C_Digest	any	20

6.15.6 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 will be taken from the start of the full 20-byte HMAC output.

Table 58, General-length RIPE-MD 160-HMAC:

Function	Key type	Data length	Signature length
C_Sign	generic secret	any	0-20, depending on parameters
C_Verify	generic secret	any	0-20, depending on parameters

6.15.7 RIPE-MD 160-HMAC

The RIPE-MD 160-HMAC mechanism, denoted **CKM_RIPEMD160_HMAC**, is a special case of the general-length RIPE-MD 160-HMAC mechanism in Section 6.15.6.

It has no parameter, and always produces an output of length 20.

6.16 SET

6.16.1 Definitions

Mechanisms:

```
CKM_KEY_WRAP_SET_OAEP
```

6.16.2 SET mechanism parameters

- **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:

```
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**.

6.16.3 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 optionally 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 in Section **Error! Reference source not found.** on producing output. 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 will 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 is never 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 can be avoided by this means.

6.17 LYNKS

6.17.1 Definitions

Mechanisms:

```
CKM_KEY_WRAP_LYNKS
```

6.17.2 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 can 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 a 8-byte secret key *K* with a DES key *W*, this mechanism performs the following steps:

1. Initialize two 16-bit integers, sum_1 and sum_2, to 0.

2. Loop through the bytes of *K* from first to last.

    3. Set sum_1= sum_1+the key byte (treat the key byte as a number in the range 0-255).

    4. Set sum_2= sum_2+ sum_1.

5. Encrypt *K* with *W* in ECB mode, obtaining an encrypted key, *E*.

6. Concatenate the last 6 bytes of *E* with sum_2, representing sum_2 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.

A Manifest constants

The following definitions can be found in the appropriate header file.

Also, refer [PKCS #11-B] for additional definitions.

```
#define CKK_KEA              0x00000005
#define CKK_RC2              0x00000011
#define CKK_RC4              0x00000012
#define CKK_DES              0x00000013
#define CKK_CAST             0x00000016
#define CKK_CAST3            0x00000017
#define CKK_CAST5            0x00000018
#define CKK_CAST128          0x00000018
#define CKK_RC5              0x00000019
#define CKK_IDEA             0x0000001A
#define CKK_SKIPJACK         0x0000001B
#define CKK_BATON            0x0000001C
#define CKK_JUNIPER          0x0000001D

#define CKM_MD2_RSA_PKCS                    0x00000004
#define CKM_MD5_RSA_PKCS                    0x00000005
#define CKM_RIPEMD128_RSA_PKCS              0x00000007
#define CKM_RIPEMD160_RSA_PKCS              0x00000008
#define CKM_RC2_KEY_GEN                     0x00000100
#define CKM_RC2_ECB                         0x00000101
#define CKM_RC2_CBC                         0x00000102
#define CKM_RC2_MAC                         0x00000103
#define CKM_RC2_MAC_GENERAL                 0x00000104
#define CKM_RC2_CBC_PAD                     0x00000105
#define CKM_RC4_KEY_GEN                     0x00000110
#define CKM_RC4                             0x00000111
#define CKM_DES_KEY_GEN                     0x00000120
#define CKM_DES_ECB                         0x00000121
#define CKM_DES_CBC                         0x00000122
#define CKM_DES_MAC                         0x00000123
#define CKM_DES_MAC_GENERAL                 0x00000124
#define CKM_DES_CBC_PAD                     0x00000125
#define CKM_MD2                             0x00000200
#define CKM_MD2_HMAC                        0x00000201
#define CKM_MD2_HMAC_GENERAL                0x00000202
#define CKM_MD5                             0x00000210
#define CKM_MD5_HMAC                        0x00000211
#define CKM_MD5_HMAC_GENERAL                0x00000212
#define CKM_RIPEMD128                       0x00000230
#define CKM_RIPEMD128_HMAC                  0x00000231
#define CKM_RIPEMD128_HMAC_GENERAL          0x00000232
#define CKM_RIPEMD160                       0x00000240
#define CKM_RIPEMD160_HMAC                  0x00000241
#define CKM_RIPEMD160_HMAC_GENERAL          0x00000242
#define CKM_CAST_KEY_GEN                    0x00000300
#define CKM_CAST_ECB                        0x00000301
#define CKM_CAST_CBC                        0x00000302
#define CKM_CAST_MAC                        0x00000303
#define CKM_CAST_MAC_GENERAL                0x00000304
#define CKM_CAST_CBC_PAD                    0x00000305
#define CKM_CAST3_KEY_GEN                   0x00000310
```

```
#define CKM_CAST3_ECB                         0x00000311
#define CKM_CAST3_CBC                         0x00000312
#define CKM_CAST3_MAC                         0x00000313
#define CKM_CAST3_MAC_GENERAL                 0x00000314
#define CKM_CAST3_CBC_PAD                     0x00000315
#define CKM_CAST5_KEY_GEN                     0x00000320
#define CKM_CAST128_KEY_GEN                   0x00000320
#define CKM_CAST5_ECB                         0x00000321
#define CKM_CAST128_ECB                       0x00000321
#define CKM_CAST5_CBC                         0x00000322
#define CKM_CAST128_CBC                       0x00000322
#define CKM_CAST5_MAC                         0x00000323
#define CKM_CAST128_MAC                       0x00000323
#define CKM_CAST5_MAC_GENERAL                 0x00000324
#define CKM_CAST128_MAC_GENERAL               0x00000324
#define CKM_CAST5_CBC_PAD                     0x00000325
#define CKM_CAST128_CBC_PAD                   0x00000325
#define CKM_RC5_KEY_GEN                       0x00000330
#define CKM_RC5_ECB                           0x00000331
#define CKM_RC5_CBC                           0x00000332
#define CKM_RC5_MAC                           0x00000333
#define CKM_RC5_MAC_GENERAL                   0x00000334
#define CKM_RC5_CBC_PAD                       0x00000335
#define CKM_IDEA_KEY_GEN                      0x00000340
#define CKM_IDEA_ECB                          0x00000341
#define CKM_IDEA_CBC                          0x00000342
#define CKM_IDEA_MAC                          0x00000343
#define CKM_IDEA_MAC_GENERAL                  0x00000344
#define CKM_IDEA_CBC_PAD                      0x00000345
#define CKM_MD5_KEY_DERIVATION                0x00000390
#define CKM_MD2_KEY_DERIVATION                0x00000391
#define CKM_PBE_MD2_DES_CBC                   0x000003A0
#define CKM_PBE_MD5_DES_CBC                   0x000003A1
#define CKM_PBE_MD5_CAST_CBC                  0x000003A2
#define CKM_PBE_MD5_CAST3_CBC                 0x000003A3
#define CKM_PBE_MD5_CAST5_CBC                 0x000003A4
#define CKM_PBE_MD5_CAST128_CBC               0x000003A4
#define CKM_PBE_SHA1_CAST5_CBC                0x000003A5
#define CKM_PBE_SHA1_CAST128_CBC              0x000003A5
#define CKM_PBE_SHA1_RC4_128                  0x000003A6
#define CKM_PBE_SHA1_RC4_40                   0x000003A7
#define CKM_PBE_SHA1_RC2_128_CBC              0x000003AA
#define CKM_PBE_SHA1_RC2_40_CBC               0x000003AB
#define CKM_KEY_WRAP_LYNKS                    0x00000400
#define CKM_KEY_WRAP_SET_OAEP                 0x00000401
#define CKM_SKIPJACK_KEY_GEN                  0x00001000
#define CKM_SKIPJACK_ECB64                    0x00001001
#define CKM_SKIPJACK_CBC64                    0x00001002
#define CKM_SKIPJACK_OFB64                    0x00001003
#define CKM_SKIPJACK_CFB64                    0x00001004
#define CKM_SKIPJACK_CFB32                    0x00001005
#define CKM_SKIPJACK_CFB16                    0x00001006
#define CKM_SKIPJACK_CFB8                     0x00001007
#define CKM_SKIPJACK_WRAP                     0x00001008
#define CKM_SKIPJACK_PRIVATE_WRAP             0x00001009
#define CKM_SKIPJACK_RELAYX                   0x0000100a
#define CKM_KEA_KEY_PAIR_GEN                  0x00001010
#define CKM_KEA_KEY_DERIVE                    0x00001011
#define CKM_FORTEZZA_TIMESTAMP                0x00001020
#define CKM_BATON_KEY_GEN                     0x00001030
```

```
#define CKM_BATON_ECB128                    0x00001031
#define CKM_BATON_ECB96                     0x00001032
#define CKM_BATON_CBC128                    0x00001033
#define CKM_BATON_COUNTER                   0x00001034
#define CKM_BATON_SHUFFLE                   0x00001035
#define CKM_BATON_WRAP                      0x00001036
#define CKM_JUNIPER_KEY_GEN                 0x00001060
#define CKM_JUNIPER_ECB128                  0x00001061
#define CKM_JUNIPER_CBC128                  0x00001062
#define CKM_JUNIPER_COUNTER                 0x00001063
#define CKM_JUNIPER_SHUFFLE                 0x00001064
#define CKM_JUNIPER_WRAP                    0x00001065
#define CKM_FASTHASH                        0x00001070
```

B Intellectual property considerations

The RSA public-key cryptosystem is described in U.S. Patent 4,405,829, which expired on September 20, 2000. The RC5 block cipher is protected by U.S. Patents 5,724,428 and 5,835,600. RSA Security Inc. makes no other patent claims on the constructions described in this document, although specific underlying techniques may be covered.

RSA, RC2 and RC4 are registered trademarks of RSA Security Inc. RC5 is a trademark of RSA Security Inc.

CAST, CAST3, CAST5, and CAST128 are registered trademarks of Entrust Technologies. OS/2 and CDMF (Commercial Data Masking Facility) are registered trademarks of International Business Machines Corporation. LYNKS is a registered trademark of SPYRUS Corporation. IDEA is a registered trademark of Ascom Systec. Windows, Windows 3.1, Windows 95, Windows NT, and Developer Studio are registered trademarks of Microsoft Corporation. UNIX is a registered trademark of UNIX System Laboratories. FORTEZZA is a registered trademark of the National Security Agency.

License to copy this document is granted provided that it is identified as “RSA Security Inc. Public-Key Cryptography Standards (PKCS)” in all material mentioning or referencing this document.

RSA Security Inc. makes no other representations regarding intellectual property claims by other parties. Such determination is the responsibility of the user.

C Revision History

This is the initial version of PKCS #11 Other Mechanisms v2.30.

