PKCS #11 Mechanisms v2.30: Cryptoki – Draft 7

RSA Laboratories

29 July 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 Other 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

AES KEYWRAP AES Key Wrap Specification (Draft) http://csrc.nist.gov/groups/ST/toolkit/documents/kms/key-wrap.pdf.

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.

ARIA National Security Research Institute, Korea, "Block Cipher Algorithm ARIA", URL: http://www.nsri.re.kr/ARIA/index-e.html.

CT-KIP RSA Laboratories. Cryptographic Token Key Initialization Protocol. Version 1.0, December 2005. URL: ftp://ftp.rsasecurity.com/pub/otps/ct-kip/ct-kip-v1-0.pdf.

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

GCM McGrew, D. and J. Viega, "The Galois/Counter Mode of Operation (GCM)," J Submission to NIST, January 2004. URL: http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/gcm/gcm-spec.pdf.

GOST 28147-89 "Information Processing Systems. Cryptographic Protection. Cryptographic Algorithm", GOST 28147-89, Gosudarstvennyi Standard of USSR, Government Committee of the USSR for Standards, 1989. (In Russian).

GOST R 34.10-2001 "Information Technology. Cryptographic Data Security. Formation and Verification Processes of [Electronic] Digital Signature", GOST R 34.10-2001, Gosudarstvennyi Standard of the Russian Federation, Government Committee of the Russian Federation for Standards, 2001. (In Russian).

GOST R 34.11-94 "Information Technology. Cryptographic Data Security. Hashing function", GOST R 34.11-94, Gosudarstvennyi Standard of the Russian Federation, Government Committee of the Russian Federation for Standards, 1994. (In Russian).

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

NIST sp800-38a National Institute for Standards and Technology, *Recommendation for Block Cipher Modes of Operation, NIST SP 800-38A. URL: http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf*

NIST sp800-38b National Institute for Standards and Technology, *Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentications, Special Publication 800-38B. URL: http://csrc.nist.gov/publications/nistpubs/800-38B/SP_800-38B.pdf*

NIST AESCTS National Institute for Standards and Technology, *Proposal To Extend CBC Mode By "Ciphertext Stealing" . URL: http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/ciphertext%20stealing%20proposal.pdf*

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 2104	Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", February 1997.
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

RFC 2865 Rigney et al, "Remote Authentication Dial In User Service (RADIUS)", IETF RFC2865, June 2000. URL: http://ietf.org/rfc/rfc2865.txt.

RFC 3874 *Smit et al, "A 224-bit One-way Hash Function: SHA-224," IETF RFC 3874, June 2004. URL: http://ietf.org/rfc/rfc3874.txt.*

RFC 3686 Housley, "Using Advanced Encryption Standard (AES) Counter Mode With IPsec Encapsulating Security Payload (ESP)," IETF RFC 3686, January 2004. URL: http://ietf.org/rfc/rfc3686.txt.

RFC 3717 Matsui, et al, "A Description of the Camellia Encryption Algorithm," IETF RFC 3717, April 2004. URL: http://ietf.org/rfc/rfc3713.txt.

RFC 3610 Whiting, D., Housley, R., and N. Ferguson, "Counter with CBC-MAC (CCM)", IETF RFC 3610, September 2003. URL: http://www.ietf.org/rfc/rfc3610.txt

RFC 4309 Housley, R., "Using Advanced Encryption Standard (AES) CCM Mode with IPsec Encapsulating Security Payload (ESP)," IETF RFC 4309, December 2005. URL: http://ietf.org/rfc/rfc4309.txt

RFC 3748 Aboba et al, "Extensible Authentication Protocol (EAP)", IETF RFC 3748, June 2004. URL: http://ietf.org/rfc/rfc3748.txt.

RFC 3394 Advanced Encryption Standard (AES) Key Wrap Algorithm: http://www.ietf.org/rfc/rfc3394.txt.

RFC 4269 South Korean Information Security Agency (KISA) "The SEED Encryption Algorithm", December 2005. ftp://ftp.rfc-editor.org/in-notes/rfc4269.txt

RFC 4357 V. Popov, I. Kurepkin, S. Leontiev "Additional Cryptographic Algorithms for Use with GOST 28147-89, GOST R 34.10-94, GOST R 34.10-2001, and GOST R 34.11-94 Algorithms", January 2006.

RFC 4490 S. Leontiev, Ed. G. Chudov, Ed. "Using the GOST 28147-89, GOST R 34.11-94,GOST R 34.10-94, and GOST R 34.10-2001 Algorithms with Cryptographic Message Syntax (CMS)", May 2006.

RFC 4491 S. Leontiev, Ed., D. Shefanovski, Ed., "Using the GOST R 34.10-94, GOST R 34.10-2001, and GOST R 34.11-94 Algorithms with the Internet X.509 Public Key Infrastructure Certificate and CRL Profile", May 2006.

RFC 4493 J. Song et al. *RFC 4493: The AES-CMAC Algorithm.* June 2006. URL: http://www.ietf.org/rfc/rfc4493.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. Please refer to the PKCS#11 base document for further definitions:

AES	Advanced Encryption Standard, as defined in FIPS PUB 197.
CAMELLIA	The Camellia encryption algorithm, as defined in RFC 3713.
BLOWFISH	The Blowfish Encryption Algorithm of Bruce Schneier, www.schneier.com.
CBC	Cipher-Block Chaining mode, as defined in FIPS PUB 81.

CDMF	Commercial Data Masking Facility, a block encipherment method specified by International Business Machines Corporation and based on DES.
CMAC	Cipher-based Message Authenticate Code as defined in [NIST sp800-38b] and [RFC 4493].
CMS	Cryptographic Message Syntax (see RFC 2630)
CT-KIP	Cryptographic Token Key Initialization Protocol (as defined in [CT-KIP])
DES	Data Encryption Standard, as defined in FIPS PUB 46-3.
DSA	Digital Signature Algorithm, as defined in FIPS PUB 186-2.
EC	Elliptic Curve
ECB	Electronic Codebook mode, as defined in FIPS PUB 81.
ECDH	Elliptic Curve Diffie-Hellman.
ECDSA	Elliptic Curve DSA, as in ANSI X9.62.
ECMQV	Elliptic Curve Menezes-Qu-Vanstone
GOST 28147-89	The encryption algorithm, as defined in Part 2 [GOST 28147-89] and [RFC 4357] [RFC 4490], and RFC [4491].
GOST R 34.11-94	Hash algorithm, as defined in [GOST R 34.11-94] and [RFC 4357], [RFC 4490], and [RFC 4491].
GOST R 34.10-2001	The digital signature algorithm, as defined in [GOST R 34.10-2001] and [RFC 4357], [RFC 4490], and [RFC 4491].
IV	Initialization Vector.
MAC	Message Authentication Code.
MQV	Menezes-Qu-Vanstone
OAEP	Optimal Asymmetric Encryption Padding for RSA.
PKCS	Public-Key Cryptography Standards.
PRF	Pseudo random function.
PTD	Personal Trusted Device, as defined in MeT-PTD
RSA	The RSA public-key cryptosystem.
SHA-1	The (revised) Secure Hash Algorithm with a 160-bit message digest, as defined in FIPS PUB 180-2.

SHA-224	The Secure Hash Algorithm with a 224-bit message digest, as defined in RFC 3874. Also defined in FIPS PUB 180-2 with Change Notice 1.
SHA-256	The Secure Hash Algorithm with a 256-bit message digest, as defined in FIPS PUB 180-2.
SHA-384	The Secure Hash Algorithm with a 384-bit message digest, as defined in FIPS PUB 180-2.
SHA-512	The Secure Hash Algorithm with a 512-bit message digest, as defined in FIPS PUB 180-2.
SSL	The Secure Sockets Layer 3.0 protocol.
SO	A Security Officer user.
TLS	Transport Layer Security.
UTF-8	Universal Character Set (UCS) transformation format (UTF) that represents ISO 10646 and UNICODE strings with a variable number of octets.
WIM	Wireless Identification Module.
WTLS	Wireless Transport Layer Security.

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**.

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

Table 1, Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive

[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 RSA

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_RSA_PKCS_KEY_PAIR_GEN					✓		
CKM_RSA_X9_31_KEY_PAIR_GEN					✓		
CKM_RSA_PKCS	✓[2]	✓[2]	✓			✓	
CKM_RSA_PKCS_OAEP	✓[2]					✓	
CKM_RSA_PKCS_PSS		✓[2]					
CKM_RSA_9796		✓[2]	✓				
CKM_RSA_X_509	✓[2]	✓[2]	✓			✓	
CKM_RSA_X9_31		✓[2]					
CKM_SHA1_RSA_PKCS		✓					
CKM_SHA256_RSA_PKCS		✓					
CKM_SHA384_RSA_PKCS		✓					
CKM_SHA512_RSA_PKCS		✓					
CKM_SHA1_RSA_PKCS_PSS		✓					
CKM_SHA256_RSA_PKCS_PSS		✓					
CKM_SHA384_RSA_PKCS_PSS		✓					
CKM_SHA512_RSA_PKCS_PSS		✓					
CKM_SHA1_RSA_X9_31		✓					
CKM_RSA_PKCS_TPM_1_1	✓[2]					✓	
CKM_RSA_OAEP_TPM_1_1	✓[2]					✓	

6.1.1 Definitions

This section defines the RSA key type "CKK_RSA" for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of RSA key objects.

Mechanisms:

```
CKM_RSA_PKCS_KEY_PAIR_GEN
CKM_RSA_PKCS
CKM_RSA_9796
CKM_RSA_X_509
CKM_MD2_RSA_PKCS
CKM_MD5_RSA_PKCS
CKM_SHA1_RSA_PKCS
CKM_SHA224_RSA_PKCS
CKM_SHA256_RSA_PKCS
CKM_SHA384_RSA_PKCS
CKM_SHA512_RSA_PKCS
CKM_RIPEMD128_RSA_PKCS
CKM_RIPEMD160_RSA_PKCS
CKM_RSA_PKCS_OAEP
CKM_RSA_X9_31_KEY_PAIR_GEN
CKM_RSA_X9_31
CKM_SHA1_RSA_X9_31
CKM_RSA_PKCS_PSS
CKM_SHA1_RSA_PKCS_PSS
CKM_SHA224_RSA_PKCS_PSS
CKM_SHA256_RSA_PKCS_PSS
CKM_SHA512_RSA_PKCS_PSS
CKM_SHA384_RSA_PKCS_PSS
CKM_RSA_PKCS_TPM_1_1
CKM_RSA_OAEP_TPM_1_1
```

6.1.2 RSA public key objects

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

Table 2, RSA Public Key Object Attributes

Attribute	**Data type**	**Meaning**
CKA_MODULUS[1,4]	Big integer	Modulus *n*
CKA_MODULUS_BITS[2,3]	CK_ULONG	Length in bits of modulus *n*
CKA_PUBLIC_EXPONENT[1]	Big integer	Public exponent *e*

- Refer to [PKCS #11-B] table 15 for footnotes

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

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

```
CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_RSA;
CK_UTF8CHAR label[] = "An RSA public key object";
CK_BYTE modulus[] = {...};
CK_BYTE exponent[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
   {CKA_CLASS, &class, sizeof(class)},
   {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   {CKA_TOKEN, &true, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)-1},
   {CKA_WRAP, &true, sizeof(true)},
   {CKA_ENCRYPT, &true, sizeof(true)},
   {CKA_MODULUS, modulus, sizeof(modulus)},
   {CKA_PUBLIC_EXPONENT, exponent, sizeof(exponent)}
};
```

6.1.3 RSA private key objects

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

Table 3, RSA Private Key Object Attributes

Attribute	**Data type**	**Meaning**
CKA_MODULUS[1,4,6]	Big integer	Modulus *n*
CKA_PUBLIC_EXPONENT[4,6]	Big integer	Public exponent *e*
CKA_PRIVATE_EXPONENT[1,4,6,7]	Big integer	Private exponent *d*
CKA_PRIME_1[4,6,7]	Big integer	Prime *p*
CKA_PRIME_2[4,6,7]	Big integer	Prime *q*
CKA_EXPONENT_1[4,6,7]	Big integer	Private exponent *d* modulo *p*-1
CKA_EXPONENT_2[4,6,7]	Big integer	Private exponent *d* modulo *q*-1
CKA_COEFFICIENT[4,6,7]	Big integer	CRT coefficient q^{-1} mod *p*

- Refer to [PKCS #11-B] table 15 for footnotes

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

Tokens vary in what they actually store for RSA private keys. Some tokens store all of the above attributes, which can assist in performing rapid RSA computations. Other tokens might store only the **CKA_MODULUS** and **CKA_PRIVATE_EXPONENT** values.

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

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

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

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

```
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_RSA;
CK_UTF8CHAR label[] = “An RSA private key object”;
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE modulus[] = {...};
CK_BYTE publicExponent[] = {...};
CK_BYTE privateExponent[] = {...};
CK_BYTE prime1[] = {...};
CK_BYTE prime2[] = {...};
CK_BYTE exponent1[] = {...};
CK_BYTE exponent2[] = {...};
CK_BYTE coefficient[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
  {CKA_CLASS, &class, sizeof(class)},
  {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
  {CKA_TOKEN, &true, sizeof(true)},
  {CKA_LABEL, label, sizeof(label)-1},
  {CKA_SUBJECT, subject, sizeof(subject)},
  {CKA_ID, id, sizeof(id)},
  {CKA_SENSITIVE, &true, sizeof(true)},
```



```
    {CKA_DECRYPT, &true, sizeof(true)},
    {CKA_SIGN, &true, sizeof(true)},
    {CKA_MODULUS, modulus, sizeof(modulus)},
    {CKA_PUBLIC_EXPONENT, publicExponent,
          sizeof(publicExponent)},
    {CKA_PRIVATE_EXPONENT, privateExponent,
          sizeof(privateExponent)},
    {CKA_PRIME_1, prime1, sizeof(prime1)},
    {CKA_PRIME_2, prime2, sizeof(prime2)},
    {CKA_EXPONENT_1, exponent1, sizeof(exponent1)},
    {CKA_EXPONENT_2, exponent2, sizeof(exponent2)},
    {CKA_COEFFICIENT, coefficient, sizeof(coefficient)}
  };
```

6.1.4 PKCS #1 RSA key pair generation

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

It does not have a parameter.

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

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

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

6.1.5 X9.31 RSA key pair generation

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

It does not have a parameter.

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

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_MODULUS**, and **CKA_PUBLIC_EXPONENT** attributes to the new public key. It contributes the **CKA_CLASS** and **CKA_KEY_TYPE** attributes to the new private key; it may also contribute some of the following attributes to the new private key: **CKA_MODULUS**, **CKA_PUBLIC_EXPONENT**, **CKA_PRIVATE_EXPONENT**, **CKA_PRIME_1**, **CKA_PRIME_2**, **CKA_EXPONENT_1**, **CKA_EXPONENT_2**, **CKA_COEFFICIENT**. Other attributes supported by the RSA public and private key types (specifically, the flags indicating which functions the keys support) may also be specified in the templates for the keys, or else are assigned default initial values. Unlike the **CKM_RSA_PKCS_KEY_PAIR_GEN** mechanism, this mechanism is guaranteed to generate *p* and *q* values, **CKA_PRIME_1** and **CKA_PRIME_2** respectively, that meet the strong primes requirement of X9.31.

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

6.1.6 PKCS #1 v1.5 RSA

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

This mechanism does not have a parameter.

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

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

Table 4, PKCS #1 v1.5 RSA: Key And Data Length

Function	**Key type**	**Input length**	**Output length**	**Comments**
C_Encrypt[1]	RSA public key	$\leq k$-11	k	block type `02`
C_Decrypt[1]	RSA private key	k	$\leq k$-11	block type `02`
C_Sign[1]	RSA private key	$\leq k$-11	k	block type `01`
C_SignRecover	RSA private key	$\leq k$-11	k	block type `01`
C_Verify[1]	RSA public key	$\leq k$-11, k[2]	N/A	block type `01`
C_VerifyRecover	RSA public key	k	$\leq k$-11	block type `01`
C_WrapKey	RSA public key	$\leq k$-11	k	block type `02`
C_UnwrapKey	RSA private key	k	$\leq k$-11	block type `02`

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

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

6.1.7 PKCS #1 RSA OAEP mechanism parameters

♦ **CK_RSA_PKCS_MGF_TYPE; CK_RSA_PKCS_MGF_TYPE_PTR**

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

```
typedef CK_ULONG CK_RSA_PKCS_MGF_TYPE;
```


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

Table 5, PKCS #1 Mask Generation Functions

Source Identifier	Value
CKG_MGF1_SHA1	0x00000001
CKG_MGF1_SHA224	0x00000005
CKG_MGF1_SHA256	0x00000002
CKG_MGF1_SHA384	0x00000003
CKG_MGF1_SHA512	0x00000004

CK_RSA_PKCS_MGF_TYPE_PTR is a pointer to a **CK_RSA_PKCS_ MGF_TYPE**.

- ♦ **CK_RSA_PKCS_OAEP_SOURCE_TYPE; CK_RSA_PKCS_OAEP_SOURCE_TYPE_PTR**

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

```
typedef CK_ULONG CK_RSA_PKCS_OAEP_SOURCE_TYPE;
```

The following encoding parameter sources are defined in PKCS #1. The following table lists the defined sources along with the corresponding data type for the *pSourceData* field in the **CK_RSA_PKCS_OAEP_PARAMS** structure defined below.

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

Source Identifier	Value	Data Type
CKZ_DATA_SPECIFIED	0x00000001	Array of CK_BYTE containing the value of the encoding parameter. If the parameter is empty, *pSourceData* must be NULL and *ulSourceDataLen* must be zero.

CK_RSA_PKCS_OAEP_SOURCE_TYPE_PTR is a pointer to a **CK_RSA_PKCS_OAEP_SOURCE_TYPE**.

♦ **CK_RSA_PKCS_OAEP_PARAMS; CK_RSA_PKCS_OAEP_PARAMS_PTR**

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

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

The fields of the structure have the following meanings:

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

CK_RSA_PKCS_OAEP_PARAMS_PTR is a pointer to a **CK_RSA_PKCS_OAEP_PARAMS**.

6.1.8 PKCS #1 RSA OAEP

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

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

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

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

Table 7, PKCS #1 RSA OAEP: Key And Data Length

Function	Key type	Input length	Output length
C_Encrypt[1]	RSA public key	≤ *k*-2-2*hLen*	*k*
C_Decrypt[1]	RSA private key	*k*	≤ *k*-2-2*hLen*
C_WrapKey	RSA public key	≤ *k*-2-2*hLen*	*k*
C_UnwrapKey	RSA private key	*k*	≤ *k*-2-2*hLen*

[1] Single-part operations only.

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

6.1.9 PKCS #1 RSA PSS mechanism parameters

♦ **CK_RSA_PKCS_PSS_PARAMS; CK_RSA_PKCS_PSS_PARAMS_PTR**

CK_RSA_PKCS_PSS_PARAMS is a structure that provides the parameters to the **CKM_RSA_PKCS_PSS** mechanism. The structure is defined as follows:

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

The fields of the structure have the following meanings:

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

mgf mask generation function to use on the encoded block

sLen length, in bytes, of the salt value used in the PSS encoding; typical values are the length of the message hash and zero

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

6.1.10 PKCS #1 RSA PSS

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

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

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

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

Function	**Key type**	**Input length**	**Output length**
C_Sign[1]	RSA private key	*hLen*	*k*
C_Verify[1]	RSA public key	*hLen*, *k*	N/A

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

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

6.1.11 ISO/IEC 9796 RSA

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

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

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

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

This mechanism does not have a parameter.

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

Table 9, ISO/IEC 9796 RSA: Key And Data Length

Function	Key type	Input length	Output length
C_Sign[1]	RSA private key	$\leq \lfloor k/2 \rfloor$	k
C_SignRecover	RSA private key	$\leq \lfloor k/2 \rfloor$	k
C_Verify[1]	RSA public key	$\leq \lfloor k/2 \rfloor$, k[2]	N/A
C_VerifyRecover	RSA public key	k	$\leq \lfloor k/2 \rfloor$

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

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

6.1.12 X.509 (raw) RSA

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

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

This mechanism does not have a parameter.

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

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

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

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

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

Table 10, X.509 (Raw) RSA: Key And Data Length

Function	Key type	Input length	Output length
C_Encrypt[1]	RSA public key	$\le k$	k
C_Decrypt[1]	RSA private key	k	k
C_Sign[1]	RSA private key	$\le k$	k
C_SignRecover	RSA private key	$\le k$	k
C_Verify[1]	RSA public key	$\le k$, k[2]	N/A
C_VerifyRecover	RSA public key	k	k
C_WrapKey	RSA public key	$\le k$	k
C_UnwrapKey	RSA private key	k	$\le k$ (specified in template)

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

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

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

6.1.13 ANSI X9.31 RSA

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

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

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

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

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

This mechanism does not have a parameter.

Constraints on key types and the length of input and output data are summarized in the following table. In the table, *k* is the length in bytes of the RSA modulus. For all operations, the *k* value must be at least 128 and a multiple of 32 as specified in ANSI X9.31.

Table 11, ANSI X9.31 RSA: Key And Data Length

Function	Key type	Input length	Output length
C_Sign[1]	RSA private key	≤ *k*-2	*k*
C_Verify[1]	RSA public key	≤ *k*-2, *k*[2]	N/A

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

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

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

The PKCS #1 v1.5 RSA signature with MD2 mechanism, denoted **CKM_MD2_RSA_PKCS**, performs single- and multiple-part digital signatures and verification operations without message recovery. The operations performed are as described initially in PKCS #1 v1.5 with the object identifier md2WithRSAEncryption, and as in the scheme RSASSA-PKCS1-v1_5 in the current version of PKCS #1, where the underlying hash function is MD2.

Similarly, the PKCS #1 v1.5 RSA signature with MD5 mechanism, denoted **CKM_MD5_RSA_PKCS**, performs the same operations described in PKCS #1 with the object identifier md5WithRSAEncryption. The PKCS #1 v1.5 RSA signature with SHA-1 mechanism, denoted **CKM_SHA1_RSA_PKCS**, performs the same operations, except that it uses the hash function SHA-1 with object identifier sha1WithRSAEncryption.

Likewise, the PKCS #1 v1.5 RSA signature with SHA-256, SHA-384, and SHA-512 mechanisms, denoted **CKM_SHA256_RSA_PKCS**, **CKM_SHA384_RSA_PKCS**, and **CKM_SHA512_RSA_PKCS** respectively, perform the same operations using the SHA-256, SHA-384 and SHA-512 hash functions with the object identifiers sha256WithRSAEncryption, sha384WithRSAEncryption and sha384WithRSAEncryption respectively.

The PKCS #1 v1.5 RSA signature with RIPEMD-128 or RIPEMD-160, denoted **CKM_RIPEMD128_RSA_PKCS** and **CKM_RIPEMD160_RSA_PKCS** respectively, perform the same operations using the RIPE-MD 128 and RIPE-MD 160 hash functions.

None of these mechanisms has a parameter.

Constraints on key types and the length of the data for these mechanisms are summarized in the following table. In the table, *k* is the length in bytes of the RSA modulus. For the PKCS #1 v1.5 RSA signature with MD2 and PKCS #1 v1.5 RSA signature with MD5 mechanisms, *k* must be at least 27; for the PKCS #1 v1.5 RSA signature with SHA-1 mechanism, *k* must be at least 31, and so on for other underlying hash functions, where the minimum is always 11 bytes more than the length of the hash value.

Table 12, PKCS #1 v1.5 RSA Signatures with Various Hash Functions: Key And Data Length

Function	**Key type**	**Input length**	**Output length**	**Comments**
C_Sign	RSA private key	any	*k*	block type `01`
C_Verify	RSA public key	any, *k*[2]	N/A	block type `01`

[2] Data length, signature length.

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

6.1.15 PKCS #1 v1.5 RSA signature with SHA-224

The PKCS #1 v1.5 RSA signature with SHA-224 mechanism, denoted **CKM_SHA224_RSA_PKCS,** performs similarly as the other **CKM_SHA*X*_RSA_PKCS** mechanisms but uses the SHA-224 hash function.

6.1.16 PKCS #1 RSA PSS signature with SHA-224

The PKCS #1 RSA PSS signature with SHA-224 mechanism, denoted **CKM_SHA224_RSA_PKCS_PSS**, performs similarly as the other **CKM_SHA*X*_RSA_PSS** mechanisms but uses the SHA-224 hash function.

6.1.17 PKCS #1 RSA PSS signature with SHA-1, SHA-256, SHA-384 or SHA-512

The PKCS #1 RSA PSS signature with SHA-1 mechanism, denoted **CKM_SHA1_RSA_PKCS_PSS**, performs single- and multiple-part digital signatures and verification operations without message recovery. The operations performed are as described in PKCS #1 with the object identifier id-RSASSA-PSS, i.e., as in the scheme RSASSA-PSS in PKCS #1 where the underlying hash function is SHA-1.

The PKCS #1 RSA PSS signature with SHA-256, SHA-384, and SHA-512 mechanisms, denoted **CKM_SHA256_RSA_PKCS_PSS**, **CKM_SHA384_RSA_PKCS_PSS**, and **CKM_SHA512_RSA_PKCS_PSS** respectively, perform the same operations using the SHA-256, SHA-384 and SHA-512 hash functions.

The mechanisms have a parameter, a **CK_RSA_PKCS_PSS_PARAMS** structure. The *sLen* field must be less than or equal to *k**-2-*hLen* where *hLen* is the length in bytes of the hash value. *k** is the length in bytes of the RSA modulus, except if the length in bits of the RSA modulus is one more than a multiple of 8, in which case *k** is one less than the length in bytes of the RSA modulus.

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

Table 13, PKCS #1 RSA PSS Signatures with Various Hash Functions: Key And Data Length

Function	Key type	Input length	Output length
C_Sign	RSA private key	any	*k*
C_Verify	RSA public key	any, *k*[2]	N/A

[2] Data length, signature length.

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

6.1.18 ANSI X9.31 RSA signature with SHA-1

The ANSI X9.31 RSA signature with SHA-1 mechanism, denoted **CKM_SHA1_RSA_X9_31**, performs single- and multiple-part digital signatures and verification operations without message recovery. The operations performed are as described in ANSI X9.31.

This mechanism does not have a parameter.

Constraints on key types and the length of the data for these mechanisms are summarized in the following table. In the table, *k* is the length in bytes of the RSA modulus. For all operations, the *k* value must be at least 128 and a multiple of 32 as specified in ANSI X9.31.

Table 14, ANSI X9.31 RSA Signatures with SHA-1: Key And Data Length

Function	Key type	Input length	Output length
C_Sign	RSA private key	any	*k*
C_Verify	RSA public key	any, *k*[2]	N/A

[2] Data length, signature length.

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

6.1.19 TPM 1.1 PKCS #1 v1.5 RSA

The TPM 1.1 PKCS #1 v1.5 RSA mechanism, denoted **CKM_RSA_PKCS_TPM_1_1**, is a multi-use mechanism based on the RSA public-key cryptosystem and the block formats initially defined in PKCS #1 v1.5, with additional formatting rules defined in TCG TPM Specification Version 1.2. It supports single-part encryption and decryption; key wrapping; and key unwrapping.

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

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

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

Table 15, TPM 1.1 PKCS #1 v1.5 RSA: Key And Data Length

Function	Key type	Input length	Output length
C_Encrypt[1]	RSA public key	≤ *k*-11-5	*k*
C_Decrypt[1]	RSA private key	*k*	≤ *k*-11-5
C_WrapKey	RSA public key	≤ *k*-11-5	*k*
C_UnwrapKey	RSA private key	*k*	≤ *k*-11-5

[1] Single-part operations only.

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

6.1.20 TPM 1.1 PKCS #1 RSA OAEP

The TPM 1.1 PKCS #1 RSA OAEP mechanism, denoted **CKM_RSA_PKCS_OAEP_TPM_1_1**, is a multi-purpose mechanism based on the RSA public-key cryptosystem and the OAEP block format defined in PKCS #1, with additional formatting defined in TCG TPM Specification Version 1.2. It supports single-part encryption and decryption; key wrapping; and key unwrapping.

This mechanism does not have a parameter. It differs from the standard PKCS#1 OAEP RSA encryption mechanism in that the plaintext is wrapped in a TPM_BOUND_DATA structure before being submitted to the encryption process and that all of the values of the parameters that are passed to a standard CKM_RSA_PKCS_OAEP operation are fixed. On encryption, the version field of the TPM_BOUND_DATA structure must contain 0x01, 0x01, 0x00, 0x00. On decryption, any structure of the form 0x01, 0x01, 0xXX, 0xYY may be accepted.

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

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

Table 16, PKCS #1 RSA OAEP: Key And Data Length

Function	Key type	Input length	Output length
C_Encrypt[1]	RSA public key	≤ *k*-2-40-5	*k*
C_Decrypt[1]	RSA private key	*k*	≤ *k*-2-40-5
C_WrapKey	RSA public key	≤ *k*-2-40-5	*k*
C_UnwrapKey	RSA private key	*k*	≤ *k*-2-40-5

[1] Single-part operations only.

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

6.2 DSA

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_DSA_KEY_PAIR_GEN					✓		
CKM_DSA_PARAMETER_GEN					✓		
CKM_DSA		✓[2]					
CKM_DSA_SHA1		✓					

6.2.1 Definitions

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

Mechanisms:

```
CKM_DSA_KEY_PAIR_GEN
CKM_DSA
CKM_DSA_SHA1
CKM_DSA_PARAMETER_GEN
CKM_FORTEZZA_TIMESTAMP
```

6.2.2 DSA public key objects

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

Table 17, DSA 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*
CKA_VALUE[1,4]	Big integer	Public value *y*

- Refer to [PKCS #11-B] table 15 for footnotes

The **CKA_PRIME**, **CKA_SUBPRIME** and **CKA_BASE** attribute values are collectively the “DSA domain parameters”. See FIPS PUB 186-2 for more information on DSA keys.

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

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

6.2.3 DSA private key objects

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

Table 18, DSA Private Key Object Attributes

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

- Refer to [PKCS #11-B] table 15 for footnotes

The **CKA_PRIME**, **CKA_SUBPRIME** and **CKA_BASE** attribute values are collectively the "DSA domain parameters". See FIPS PUB 186-2 for more information on DSA keys.

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

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

```
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_DSA;
CK_UTF8CHAR label[] = "A DSA private key object";
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE prime[] = {...};
CK_BYTE subprime[] = {...};
CK_BYTE base[] = {...};
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
  {CKA_CLASS, &class, sizeof(class)},
  {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
  {CKA_TOKEN, &true, sizeof(true)},
  {CKA_LABEL, label, sizeof(label)-1},
  {CKA_SUBJECT, subject, sizeof(subject)},
  {CKA_ID, id, sizeof(id)},
  {CKA_SENSITIVE, &true, sizeof(true)},
  {CKA_SIGN, &true, sizeof(true)},
  {CKA_PRIME, prime, sizeof(prime)},
  {CKA_SUBPRIME, subprime, sizeof(subprime)},
  {CKA_BASE, base, sizeof(base)},
  {CKA_VALUE, value, sizeof(value)}
};
```


6.2.4 DSA domain parameter objects

DSA domain parameter objects (object class **CKO_DOMAIN_PARAMETERS,** key type **CKK_DSA**) hold DSA domain parameters. The following table defines the DSA domain parameter object attributes, in addition to the common attributes defined for this object class:

Table 19, DSA Domain Parameter Object Attributes

Attribute	Data type	Meaning
CKA_PRIME[1,4]	Big integer	Prime *p* (512 to 1024 bits, in steps of 64 bits)
CKA_SUBPRIME[1,4]	Big integer	Subprime *q* (160 bits)
CKA_BASE[1,4]	Big integer	Base *g*
CKA_PRIME_BITS[2,3]	CK_ULONG	Length of the prime value.

- Refer to [PKCS #11-B] table 15 for footnotes

The **CKA_PRIME**, **CKA_SUBPRIME** and **CKA_BASE** attribute values are collectively the "DSA domain parameters". See FIPS PUB 186-2 for more information on DSA domain parameters.

The following is a sample template for creating a DSA domain parameter object:

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

6.2.5 DSA key pair generation

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

This mechanism does not have a parameter.

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

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

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

6.2.6 DSA domain parameter generation

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

This mechanism does not have a parameter.

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

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

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

6.2.7 DSA without hashing

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

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

It does not have a parameter.

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

Table 20, DSA: 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.

6.2.8 DSA with SHA-1

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

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

This mechanism does not have a parameter.

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

Table 21, DSA with SHA-1: Key And Data Length

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

[2] Data length, signature length.

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

6.3 Elliptic Curve

The Elliptic Curve (EC) cryptosystem (also related to ECDSA) in this document is the one described in the ANSI X9.62 and X9.63 standards developed by the ANSI X9F1 working group.

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_EC_KEY_PAIR_GEN (CKM_ECDSA_KEY_PAIR_GEN)					✓		
CKM_ECDSA		✓[2]					
CKM_ECDSA_SHA1		✓					
CKM_ECDH1_DERIVE							✓
CKM_ECDH1_COFACTOR_DERIVE							✓
CKM_ECMQV_DERIVE							✓

Table 22, Mechanism Information Flags

CKF_EC_F_P	0x00100000	True if the mechanism can be used with EC domain parameters over F_p
CKF_EC_F_2M	0x00200000	True if the mechanism can be used with EC domain parameters over F_{2^m}
CKF_EC_ECPARAMETERS	0x00400000	True if the mechanism can be used with EC domain parameters of the choice **ecParameters**
CKF_EC_NAMEDCURVE	0x00800000	True if the mechanism can be used with EC domain parameters of the choice **namedCurve**
CKF_EC_UNCOMPRESS	0x01000000	True if the mechanism can be used with elliptic curve point uncompressed
CKF_EC_COMPRESS	0x02000000	True if the mechanism can be used with elliptic curve point compressed

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

1. EC using a field with an odd prime number of elements (i.e. the finite field F_p).

2. EC using a field of characteristic two (i.e. the finite field F_{2^m}).

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

In these specifications there are also three representation methods to define the domain parameters for an EC key. Only the **ecParameters** and the **namedCurve** choices are supported in Cryptoki. The **CK_MECHANISM_INFO** structure **CKF_EC_ECPARAMETERS** flag identifies a Cryptoki library supporting the **ecParameters** choice whereas the **CKF_EC_NAMEDCURVE** flag identifies a Cryptoki library supporting the **namedCurve** choice. A Cryptoki library that can perform EC mechanisms must set either or both of these flags for each EC mechanism.

In these specifications, an EC public key (i.e. EC point *Q*) or the base point *G* when the **ecParameters** choice is used can be represented as an octet string of the uncompressed form or the compressed form. The **CK_MECHANISM_INFO** structure **CKF_EC_UNCOMPRESS** flag identifies a Cryptoki library supporting the uncompressed form whereas the **CKF_EC_COMPRESS** flag identifies a Cryptoki library supporting the compressed form. A Cryptoki library that can perform EC mechanisms must set either or both of these flags for each EC mechanism.

Note that an implementation of a Cryptoki library supporting EC with only one variety, one representation of domain parameters or one form may encounter difficulties achieving interoperability with other implementations.

If an attempt to create, generate, derive, or unwrap an EC key of an unsupported variety (or of an unsupported size of a supported variety) is made, that attempt should fail with the error code CKR_TEMPLATE_INCONSISTENT. If an attempt to create, generate, derive, or unwrap an EC key with invalid or of an unsupported representation of domain parameters is made, that attempt should fail with the error code CKR_DOMAIN_PARAMS_INVALID. If an attempt to create, generate, derive, or unwrap an EC key of an unsupported form is made, that attempt should fail with the error code CKR_TEMPLATE_INCONSISTENT.

6.3.1 EC Signatures

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

If the length of the hash value is larger than the bit length of *n*, only the leftmost bits of the hash up to the length of *n* will be used. Any truncation is done by the token.

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

6.3.2 Definitions

This section defines the key type "CKK_ECDSA" and "CKK_EC" for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

```
Note: CKM_ECDSA_KEY_PAIR_GEN is deprecated in v2.11
CKM_ECDSA_KEY_PAIR_GEN
CKM_EC_KEY_PAIR_GEN
CKM_ECDSA
CKM_ECDSA_SHA1
CKM_ECDH1_DERIVE
CKM_ECDH1_COFACTOR_DERIVE
CKM_ECMQV_DERIVE

CKD_NULL
CKD SHA1_KDF
```

6.3.3 ECDSA public key objects

EC (also related to ECDSA) public key objects (object class **CKO_PUBLIC_KEY,** key type **CKK_EC** or **CKK_ECDSA**) hold EC public keys. The following table defines the EC public key object attributes, in addition to the common attributes defined for this object class:

Table 23, Elliptic Curve Public Key Object Attributes

Attribute	Data type	Meaning
CKA_EC_PARAMS[1,3] (CKA_ECDSA_PARAMS)	Byte array	DER-encoding of an ANSI X9.62 `Parameters` value
CKA_EC_POINT[1,4]	Byte array	DER-encoding of ANSI X9.62 `ECPoint` value *Q*

- Refer to [PKCS #11-B] table 15 for footnotes

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

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

This allows detailed specification of all required values using choice **ecParameters**, the use of a **namedCurve** as an object identifier substitute for a particular set of elliptic curve domain parameters, or **implicitlyCA** to indicate that the domain parameters are explicitly defined elsewhere. The use of a **namedCurve** is recommended over the choice **ecParameters**. The choice **implicitlyCA** must not be used in Cryptoki.

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

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

6.3.4 Elliptic curve private key objects

EC (also related to ECDSA) private key objects (object class **CKO_PRIVATE_KEY,** key type **CKK_EC** or **CKK_ECDSA**) hold EC private keys. See Section 6.3 for more information about EC. The following table defines the EC private key object attributes, in addition to the common attributes defined for this object class:

Table 24, Elliptic Curve Private Key Object Attributes

Attribute	Data type	Meaning
CKA_EC_PARAMS[1,4,6] (CKA_ECDSA_PARAMS)	Byte array	DER-encoding of an ANSI X9.62 `Parameters` value
CKA_VALUE[1,4,6,7]	Big integer	ANSI X9.62 private value *d*

- Refer to [PKCS #11-B] table 15 for footnotes

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

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

This allows detailed specification of all required values using choice **ecParameters**, the use of a **namedCurve** as an object identifier substitute for a particular set of elliptic curve domain parameters, or **implicitlyCA** to indicate that the domain parameters are explicitly defined elsewhere. The use of a **namedCurve** is recommended over the choice **ecParameters**. The choice **implicitlyCA** must not be used in Cryptoki.

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

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

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

```
    {CKA_SUBJECT, subject, sizeof(subject)},
    {CKA_ID, id, sizeof(id)},
    {CKA_SENSITIVE, &true, sizeof(true)},
    {CKA_DERIVE, &true, sizeof(true)},
    {CKA_EC_PARAMS, ecParams, sizeof(ecParams)},
    {CKA_VALUE, value, sizeof(value)}
  };
```

6.3.5 Elliptic curve key pair generation

The EC (also related to ECDSA) key pair generation mechanism, denoted **CKM_EC_KEY_PAIR_GEN** or **CKM_ECDSA_KEY_PAIR_GEN**, is a key pair generation mechanism for EC.

This mechanism does not have a parameter.

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

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

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

6.3.6 ECDSA without hashing

Refer section 6.3.1 for signature encoding.

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

This mechanism does not have a parameter.

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

Table 25, ECDSA: Key And Data Length

Function	Key type	Input length	Output length
C_Sign[1]	ECDSA private key	any[3]	2*nLen*
C_Verify[1]	ECDSA public key	any[3], ≤2*nLen* [2]	N/A

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

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

6.3.7 ECDSA with SHA-1

Refer section 6.3.1 for signature encoding.

The ECDSA with SHA-1 mechanism, denoted **CKM_ECDSA_SHA1**, is a mechanism for single- and multiple-part signatures and verification for ECDSA. This mechanism computes the entire ECDSA specification, including the hashing with SHA-1.

This mechanism does not have a parameter.

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

Table 26, ECDSA with SHA-1: Key And Data Length

Function	Key type	Input length	Output length
C_Sign	ECDSA private key	any	2*nLen*
C_Verify	ECDSA public key	any, ≤2*nLen* [2]	N/A

[2] Data length, signature length.

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

6.3.8 EC mechanism parameters

♦ **CK_EC_KDF_TYPE, CK_EC_KDF_TYPE_PTR**

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

```
typedef CK_ULONG CK_EC_KDF_TYPE;
```

The following table lists the defined functions.

Table 27, EC: Key Derivation Functions

Source Identifier
CKD_NULL
CKD_SHA1_KDF
CKD_SHA224_KDF
CKD_SHA256_KDF
CKD_SHA384_KDF
CKD_SHA512_KDF

The key derivation function **CKD_NULL** produces a raw shared secret value without applying any key derivation function whereas the key derivation function **CKD_SHA1_KDF**, which is based on SHA-1, derives keying data from the shared secret value as defined in ANSI X9.63.

CK_EC_KDF_TYPE_PTR is a pointer to a **CK_EC_KDF_TYPE**.

- **CK_ECDH1_DERIVE_PARAMS, CK_ECDH1_DERIVE_PARAMS_PTR**

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

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

The fields of the structure have the following meanings:

kdf	key derivation function used on the shared secret value
ulSharedDataLen	the length in bytes of the shared info
pSharedData	some data shared between the two parties
ulPublicDataLen	the length in bytes of the other party's EC public key
pPublicData†	pointer to other party's EC public key value. A token MUST be able to accept this value encoded as a raw octet string (as per section A.5.2 of [ANSI X9.62]). A token MAY, in addition, support accepting this value as a DER-encoded ECPoint (as per section E.6 of [ANSI X9.62]) i.e. the same as a CKA_EC_POINT encoding. The calling application is responsible for converting the offered public key to the compressed or uncompressed forms of these encodings if the token does not support the offered form.

With the key derivation function **CKD_NULL**, *pSharedData* must be NULL and *ulSharedDataLen* must be zero. With the key derivation function **CKD_SHA1_KDF**, an optional *pSharedData* may be supplied, which consists of some data shared by the two parties intending to share the shared secret. Otherwise, *pSharedData* must be NULL and *ulSharedDataLen* must be zero.

_

† *The encoding in V2.20 was not specified and resulted in different implementations choosing different encodings. Applications relying only on a V2.20 encoding (e.g. the DER variant) other than the one specified now (raw) may not work with all V2.30 compliant tokens.*

CK_ECDH1_DERIVE_PARAMS_PTR is a pointer to a **CK_ECDH1_DERIVE_PARAMS**.

♦ **CK_ ECMQV _DERIVE_PARAMS, CK_ ECMQV _DERIVE_PARAMS_PTR**

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

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

The fields of the structure have the following meanings:

kdf	key derivation function used on the shared secret value
ulSharedDataLen	the length in bytes of the shared info
pSharedData	some data shared between the two parties
ulPublicDataLen	the length in bytes of the other party's first EC public key
pPublicData	pointer to other party's first EC public key value. Encoding rules are as per *pPublicData* of CK_ECDH1_DERIVE_PARAMS
ulPrivateDataLen	the length in bytes of the second EC private key
hPrivateData	key handle for second EC private key value
ulPublicDataLen2	the length in bytes of the other party's second EC public key
pPublicData2	pointer to other party's second EC public key value. Encoding rules are as per *pPublicData* of CK_ECDH1_DERIVE_PARAMS

publicKey Handle to the first party's ephemeral public key

With the key derivation function **CKD_NULL**, *pSharedData* must be NULL and *ulSharedDataLen* must be zero. With the key derivation function **CKD_SHA1_KDF**, an optional *pSharedData* may be supplied, which consists of some data shared by the two parties intending to share the shared secret. Otherwise, *pSharedData* must be NULL and *ulSharedDataLen* must be zero.

CK_ECMQV_DERIVE_PARAMS_PTR is a pointer to a **CK_ECMQV_DERIVE_PARAMS**.

6.3.9 Elliptic curve Diffie-Hellman key derivation

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

It has a parameter, a **CK_ECDH1_DERIVE_PARAMS** structure.

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

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

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

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

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

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

6.3.10 Elliptic curve Diffie-Hellman with cofactor key derivation

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

It has a parameter, a **CK_ECDH1_DERIVE_PARAMS** structure.

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

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

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

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

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

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

6.3.11 Elliptic curve Menezes-Qu-Vanstone key derivation

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

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

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

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

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

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

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

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

6.4 Diffie-Hellman

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_DH_PKCS_KEY_PAIR_GEN					✓		
CKM_DH_PKCS_PARAMETER_GEN					✓		
CKM_DH_PKCS_DERIVE							✓
CKM_X9_42_DH_KEY_PAIR_GEN					✓		
CKM_X9_42_DH_PKCS_PARAMETER_GEN					✓		
CKM_X9_42_DH_DERIVE							✓
CKM_X9_42_DH_HYBRID_DERIVE							✓
CKM_X9_42_MQV_DERIVE							✓

6.4.1 Definitions

This section defines the key type "CKK_DH" for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of DH key objects.

Mechanisms:

```
CKM_DH_PKCS_KEY_PAIR_GEN
CKM_DH_PKCS_DERIVE
CKM_X9_42_DH_KEY_PAIR_GEN
CKM_X9_42_DH_DERIVE
CKM_X9_42_DH_HYBRID_DERIVE
CKM_X9_42_MQV_DERIVE
CKM_DH_PKCS_PARAMETER_GEN
CKM_X9_42_DH_PARAMETER_GEN
```

6.4.2 Diffie-Hellman public key objects

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

Table 28, Diffie-Hellman Public Key Object Attributes

Attribute	Data type	Meaning
CKA_PRIME[1,3]	Big integer	Prime *p*
CKA_BASE[1,3]	Big integer	Base *g*
CKA_VALUE[1,4]	Big integer	Public value *y*

- Refer to [PKCS #11-B] table 15 for footnotes

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

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

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

6.4.3 X9.42 Diffie-Hellman public key objects

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

Table 29, X9.42 Diffie-Hellman Public Key Object Attributes

Attribute	Data type	Meaning
CKA_PRIME[1,3]	Big integer	Prime *p* (≥ 1024 bits, in steps of 256 bits)
CKA_BASE[1,3]	Big integer	Base *g*
CKA_SUBPRIME[1,3]	Big integer	Subprime *q* (≥ 160 bits)
CKA_VALUE[1,4]	Big integer	Public value *y*

- Refer to [PKCS #11-B] table 15 for footnotes

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

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

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

6.4.4 Diffie-Hellman private key objects

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

Table 30, Diffie-Hellman Private Key Object Attributes

Attribute	Data type	Meaning
CKA_PRIME[1,4,6]	Big integer	Prime *p*
CKA_BASE[1,4,6]	Big integer	Base *g*
CKA_VALUE[1,4,6,7]	Big integer	Private value *x*
CKA_VALUE_BITS[2,6]	CK_ULONG	Length in bits of private value *x*

- Refer to [PKCS #11-B] table 15 for footnotes

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

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

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

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

6.4.5 X9.42 Diffie-Hellman private key objects

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

Table 31, X9.42 Diffie-Hellman Private Key Object Attributes

Attribute	**Data type**	**Meaning**
CKA_PRIME[1,4,6]	Big integer	Prime *p* (≥ 1024 bits, in steps of 256 bits)
CKA_BASE[1,4,6]	Big integer	Base *g*
CKA_SUBPRIME[1,4,6]	Big integer	Subprime *q* (≥ 160 bits)
CKA_VALUE[1,4,6,7]	Big integer	Private value *x*

- Refer to [PKCS #11-B] table 15 for footnotes

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

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

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

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



```
    {CKA_BASE, base, sizeof(base)},
    {CKA_SUBPRIME, subprime, sizeof(subprime)},
    {CKA_VALUE, value, sizeof(value)}
  };
```

6.4.6 Diffie-Hellman domain parameter objects

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

Table 32, Diffie-Hellman Domain Parameter Object Attributes

Attribute	Data type	Meaning
CKA_PRIME[1,4]	Big integer	Prime *p*
CKA_BASE[1,4]	Big integer	Base *g*
CKA_PRIME_BITS[2,3]	CK_ULONG	Length of the prime value.

- Refer to [PKCS #11-B] table 15 for footnotes

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

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

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

6.4.7 X9.42 Diffie-Hellman domain parameters objects

X9.42 Diffie-Hellman domain parameters objects (object class **CKO_DOMAIN_PARAMETERS,** key type **CKK_X9_42_DH**) hold X9.42 Diffie-

Hellman domain parameters. The following table defines the X9.42 Diffie-Hellman domain parameters object attributes, in addition to the common attributes defined for this object class:

Table 33, X9.42 Diffie-Hellman Domain Parameters Object Attributes

Attribute	Data type	Meaning
CKA_PRIME[1,4]	Big integer	Prime *p* (≥ 1024 bits, in steps of 256 bits)
CKA_BASE[1,4]	Big integer	Base *g*
CKA_SUBPRIME[1,4]	Big integer	Subprime *q* (≥ 160 bits)
CKA_PRIME_BITS[2,3]	CK_ULONG	Length of the prime value.
CKA_SUBPRIME_BITS[2,3]	CK_ULONG	Length of the subprime value.

- Refer to [PKCS #11-B] table 15 for footnotes

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

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

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

6.4.8 PKCS #3 Diffie-Hellman key pair generation

The PKCS #3 Diffie-Hellman key pair generation mechanism, denoted **CKM_DH_PKCS_KEY_PAIR_GEN**, is a key pair generation mechanism based on Diffie-Hellman key agreement, as defined in PKCS #3. This is what PKCS #3 calls "phase I".

It does not have a parameter.

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

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

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

6.4.9 PKCS #3 Diffie-Hellman domain parameter generation

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

It does not have a parameter.

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

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

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

6.4.10 PKCS #3 Diffie-Hellman key derivation

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

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

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

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

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

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

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

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

[‡] 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**.

6.4.11 X9.42 Diffie-Hellman mechanism parameters

♦ **CK_X9_42_DH_KDF_TYPE, CK_X9_42_DH_KDF_TYPE_PTR**

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

```
typedef CK_ULONG CK_X9_42_DH_KDF_TYPE;
```

The following table lists the defined functions.

Table 34, X9.42 Diffie-Hellman Key Derivation Functions

Source Identifier
CKD_NULL
CKD_SHA1_KDF_ASN1
CKD_SHA1_KDF_CONCATENATE

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

CK_X9_42_DH_KDF_TYPE_PTR is a pointer to a **CK_X9_42_DH_KDF_TYPE**.

♦ **CK_X9_42_DH1_DERIVE_PARAMS, CK_X9_42_DH1_DERIVE_PARAMS_PTR**

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

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

The fields of the structure have the following meanings:

kdf key derivation function used on the shared secret value

ulOtherInfoLen	the length in bytes of the other info
pOtherInfo	some data shared between the two parties
ulPublicDataLen	the length in bytes of the other party's X9.42 Diffie-Hellman public key
pPublicData	pointer to other party's X9.42 Diffie-Hellman public key value

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

CK_X9_42_DH1_DERIVE_PARAMS_PTR is a pointer to a **CK_X9_42_DH1_DERIVE_PARAMS**.

- **CK_X9_42_DH2_DERIVE_PARAMS, CK_X9_42_DH2_DERIVE_PARAMS_PTR**

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

```
typedef struct CK_X9_42_DH2_DERIVE_PARAMS {
   CK_X9_42_DH_KDF_TYPE kdf;
   CK_ULONG ulOtherInfoLen;
   CK_BYTE_PTR pOtherInfo;
   CK_ULONG ulPublicDataLen;
   CK_BYTE_PTR pPublicData;
   CK_ULONG ulPrivateDataLen;
   CK_OBJECT_HANDLE hPrivateData;
   CK_ULONG ulPublicDataLen2;
   CK_BYTE_PTR pPublicData2;
} CK_X9_42_DH2_DERIVE_PARAMS;
```

The fields of the structure have the following meanings:

kdf	key derivation function used on the shared secret value
ulOtherInfoLen	the length in bytes of the other info

pOtherInfo	some data shared between the two parties
ulPublicDataLen	the length in bytes of the other party's first X9.42 Diffie-Hellman public key
pPublicData	pointer to other party's first X9.42 Diffie-Hellman public key value
ulPrivateDataLen	the length in bytes of the second X9.42 Diffie-Hellman private key
hPrivateData	key handle for second X9.42 Diffie-Hellman private key value
ulPublicDataLen2	the length in bytes of the other party's second X9.42 Diffie-Hellman public key
pPublicData2	pointer to other party's second X9.42 Diffie-Hellman public key value

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

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

♦ CK_X9_42_MQV_DERIVE_PARAMS, CK_X9_42_MQV_DERIVE_PARAMS_PTR

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

```
typedef struct CK_X9_42_MQV_DERIVE_PARAMS {
   CK_X9_42_DH_KDF_TYPE kdf;
   CK_ULONG ulOtherInfoLen;
   CK_BYTE_PTR pOtherInfo;
   CK_ULONG ulPublicDataLen;
   CK_BYTE_PTR pPublicData;
   CK_ULONG ulPrivateDataLen;
   CK_OBJECT_HANDLE hPrivateData;
   CK_ULONG ulPublicDataLen2;
   CK_BYTE_PTR pPublicData2;
   CK_OBJECT_HANDLE publicKey;
} CK_X9_42_MQV_DERIVE_PARAMS;
```

The fields of the structure have the following meanings:

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

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

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

6.4.12 X9.42 Diffie-Hellman key pair generation

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

It does not have a parameter.

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

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

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

6.4.13 X9.42 Diffie-Hellman domain parameter generation

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

It does not have a parameter.

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

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_BASE, CKA_SUBPRIME**, **CKA_PRIME_BITS** and **CKA_SUBPRIME_BITS** attributes to the new object. Other attributes supported by the X9.42 Diffie-Hellman domain parameter types may also be specified in the template for the domain parameters, or else are assigned default initial values.

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

6.4.14 X9.42 Diffie-Hellman key derivation

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

It has a parameter, a **CK_X9_42_DH1_DERIVE_PARAMS** structure.

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

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

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

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

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

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

6.4.15 X9.42 Diffie-Hellman hybrid key derivation

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

It has a parameter, a **CK_X9_42_DH2_DERIVE_PARAMS** structure.

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

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

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

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

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

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

6.4.16 X9.42 Diffie-Hellman Menezes-Qu-Vanstone key derivation

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

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

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

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

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

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

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

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

6.5 Wrapping/unwrapping private keys

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

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

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

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

id-ecPublicKey OBJECT IDENTIFIER ::= { iso(1) member-
  body(2) us(840) ansi-x9-62(10045) publicKeyType(2) 1 }

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

where

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

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

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

```
NULL

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

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

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

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

Dss-Parms ::= SEQUENCE {
  p INTEGER,
  q INTEGER,
  g INTEGER
}
```

For the X9.42 Diffie-Hellman domain parameters, the **cofactor** and the **validationParms** optional fields should not be used when wrapping or unwrapping X9.42 Diffie-Hellman private keys since their values are not stored within the token.

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

Within the PrivateKeyInfo type:

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

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

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

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

```
ECPrivateKey ::= SEQUENCE {
   Version       INTEGER { ecPrivkeyVer1(1) }
        (ecPrivkeyVer1),
   privateKey    OCTET STRING,
```



```
    parameters    [0] Parameters OPTIONAL,
    publicKey     [1] BIT STRING OPTIONAL
}
```

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

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

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

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

Earlier drafts of PKCS #11 Version 2.0 and Version 2.01 used the object identifier

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

with associated parameters

```
DSAParameters ::= SEQUENCE {
  prime1 INTEGER,  -- modulus p
  prime2 INTEGER,  -- modulus q
  base INTEGER  -- base g
}
```

for wrapping DSA private keys. Note that although the two structures for holding DSA domain parameters appear identical when instances of them are encoded, the two corresponding object identifiers are different.

6.6 Generic secret key

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_GENERIC_SECRET_KEY_GEN					✓		

6.6.1 Definitions

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

Mechanisms:

```
CKM_GENERIC_SECRET_KEY_GEN
```

6.6.2 Generic secret key objects

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

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

Table 35, Generic Secret Key Object Attributes

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

- Refer to [PKCS #11-B] table 15 for footnotes

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

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_GENERIC_SECRET;
CK_UTF8CHAR label[] = "A generic secret key object";
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
  {CKA_CLASS, &class, sizeof(class)},
  {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
  {CKA_TOKEN, &true, sizeof(true)},
```



```
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_DERIVE, &true, sizeof(true)},
    {CKA_VALUE, value, sizeof(value)}
  };
```

CKA_CHECK_VALUE: The value of this attribute is derived from the key object by taking the first three bytes of the SHA-1 hash of the generic secret key object's CKA_VALUE attribute.

6.6.3 Generic secret key generation

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

It does not have a parameter.

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

CK_OBJECT_CLASS = **CKO_SECRET_KEY**;

CK_KEY_TYPE = **CKK_GENERIC_SECRET**;

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

6.7 HMAC mechanisms

Refer **RFC2104** and **FIPS 198** for HMAC algorithm description.. The HMAC secret key shall correspond to the PKCS11 generic secret key type or the mechanism specific key types (see mechanism definition). Such keys, for use with HMAC operations can be created using C_CreateObject or C_GenerateKey.

The RFC also specifies test vectors for the various hash function based HMAC mechanisms described in the respective hash mechanism descriptions. The RFC should be consulted to obtain these test vectors.

6.8 AES

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

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_AES_KEY_GEN					✓		
CKM_AES_ECB	✓					✓	
CKM_AES_CBC	✓					✓	
CKM_AES_CBC_PAD	✓					✓	
CKM_AES_MAC_GENERAL		✓					
CKM_AES_MAC		✓					
CKM_AES_OFB	✓					✓	
CKM_AES_CFB64	✓					✓	
CKM_AES_CFB8	✓					✓	
CKM_AES_CFB128	✓					✓	

6.8.1 Definitions

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

Mechanisms:

```
CKM_AES_KEY_GEN
CKM_AES_ECB
CKM_AES_CBC
CKM_AES_MAC
CKM_AES_MAC_GENERAL
CKM_AES_CBC_PAD
CKM_AES_OFB
CKM_AES_CFB64
CKM_AES_CFB8
CKM_AES_CFB128
```

6.8.2 AES secret key objects

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

Table 36, AES Secret Key Object Attributes

Attribute	Data type	Meaning
CKA_VALUE[1,4,6,7]	Byte array	Key value (16, 24, or 32 bytes)
CKA_VALUE_LEN[2,3,6]	CK_ULONG	Length in bytes of key value

- Refer to [PKCS #11-B] table 15 for footnotes

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

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

CKA_CHECK_VALUE: The value of this attribute is derived from the key object by taking the first three bytes of the ECB encryption of a single block of null (0x00) bytes, using the default cipher associated with the key type of the secret key object.

6.8.3 AES key generation

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

It does not have a parameter.

The mechanism generates AES keys with a particular length in bytes, as specified in the **CKA_VALUE_LEN** attribute of the template for the key.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the AES key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

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

6.8.4 AES-ECB

AES-ECB, denoted **CKM_AES_ECB**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on NIST Advanced Encryption Standard and electronic codebook mode.

It does not have a parameter.

This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus one null bytes so that the resulting length is a multiple of the block size. The output data is the same 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 37, AES-ECB: Key And Data Length

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

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

6.8.5 AES-CBC

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

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

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

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

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

Table 38, AES-CBC: Key And Data Length

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

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

6.8.6 AES-CBC with PKCS padding

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

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

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

In addition to being able to wrap and unwrap secret keys, this mechanism can wrap and unwrap RSA, Diffie-Hellman, X9.42 Diffie-Hellman, EC (also related to ECDSA) and DSA private keys (see Section 6.5 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 39, AES-CBC with PKCS Padding: Key And Data Length

Function	**Key type**	**Input length**	**Output length**
C_Encrypt	AES	any	input length rounded up to multiple of the block size
C_Decrypt	AES	multiple of block size	between 1 and block size bytes shorter than input length
C_WrapKey	AES	any	input length rounded up to multiple of the block size
C_UnwrapKey	AES	multiple of block size	between 1 and block length bytes shorter than input length

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

6.8.7 AES-OFB

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

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

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

Table 40, AES-OFB: Key And Data Length

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

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

6.8.8 AES-CFB

Cipher AES has a cipher feedback mode, AES-CFB, denoted CKM_AES_CFB8, CKM_AES_CFB64, and CKM_AES_CFB128. It is a mechanism for single and multiple-part encryption and decryption with AES. AES-OFB mode is described [NIST sp800-38a].

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

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

Table 41, AES-CFB: Key And Data Length

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

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

6.8.9 General-length AES-MAC

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

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

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

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

Table 42, General-length AES-MAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	AES	any	0-block size, as specified in parameters
C_Verify	AES	any	0-block size, as specified in parameters

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

6.8.10 AES-MAC

AES-MAC, denoted by **CKM_AES_MAC**, is a special case of the general-length AES-MAC mechanism. AES-MAC always produces and verifies MACs that are half the block size in length.

It does not have a parameter.

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

Table 43, AES-MAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	AES	any	½ block size (8 bytes)
C_Verify	AES	any	½ block size (8 bytes)

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

6.9 AES with Counter

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_AES_CTR	✓					✓	

6.9.1 Definitions

Mechanisms:

CKM_AES_CTR

6.9.2 AES with Counter mechanism parameters

♦ **CK_AES_CTR_PARAMS; CK_AES_CTR_PARAMS_PTR**

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

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

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

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

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

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            Nonce                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  Initialization Vector (IV)                   |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Block Counter                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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

CK_AES_CTR _PARAMS_PTR is a pointer to a **CK_AES_CTR _PARAMS**.

6.9.3 AES with Counter Encryption / Decryption

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

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

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

6.10 AES CBC with Cipher Text Stealing CTS

Ref [NIST AESCTS]

This mode allows unpadded data that has length that is not a multiple of the block size to be encrypted to the same length of cipher text.

Mechanism	Functions						
	Encrypt & Decrypt	**Sign & Verify**	**SR & VR[1]**	**Digest**	**Gen. Key/ Key Pair**	**Wrap & Unwrap**	**Derive**
CKM_AES_CTS	✓					✓	

6.10.1 Definitions

Mechanisms:

CKM_AES_CTS

6.10.2 AES CTS mechanism parameters

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

Table 44, AES-CTS: Key And Data Length

Function	Key type	Input length	Output length	Comments
C_Encrypt	AES	Any, ≥ block size (16 bytes)	same as input length	no final part
C_Decrypt	AES	any, ≥ block size (16 bytes)	same as input length	no final part

6.11 Additional AES Mechanisms

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_AES_GCM	✓						
CKM_AES_CCM	✓						

6.11.1 Definitions

Mechanisms:

```
CKM_AES_GCM
CKM_AES_CCM
```

6.11.2 AES GCM and CCM Mechanism parameters

♦ **CK_GCM _PARAMS; CK_GCM _PARAMS_PTR**

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

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

The fields of the structure have the following meanings:

pIv pointer to initialization vector

ulIvLen length of initialization vector in bytes. The length of the initialization vector can be any number between 1 and 2^{56}. 96-bit (12 byte) IV values can be processed

	more efficiently, so that length is recommended for situations in which efficiency is critical.
pAAD	pointer to additional authentication data. This data is authenticated but not encrypted.
ulAADLen	length of *pAAD* in bytes.
ulTagBits	length of authentication tag (output following cipher text) in bits. Can be any value between 0 and 128.

CK_GCM_PARAMS_PTR is a pointer to a **CK_GCM_PARAMS**.

- **CK_CCM _PARAMS; CK_CCM _PARAMS_PTR**

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

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

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

ulDataLen	length of the data where 0 <= *ulDataLen* < 2^{8L}.
pNonce	the nonce.
ulNonceLen	length of *pNonce* (<= 15-L) in bytes.
pAAD	Additional authentication data. This data is authenticated but not encrypted.
ulAADLen	length of *pAuthData* in bytes.
ulMACLen	length of the MAC (output following cipher text) in bytes. Valid values are 4, 6, 8, 10, 12, 14, and 16.

CK_CCM_PARAMS_PTR is a pointer to a **CK_CCM_PARAMS**.

6.11.3 AES-GCM authenticated Encryption / Decryption

Generic GCM mode is described in [GCM]. To set up for AES-GCM use the following process, where *K* (key) and *AAD* (additional authenticated data) are as described in [GCM].

Encrypt:

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

Decrypt:

- . Set the IV length *ulIvLen* in the parameter block.
- Set the IV data *pIv* in the parameter block. *pIV* may be NULL *if ulIvLen* is 0.
- Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD m*ay be NULL if ulAADLen is 0.
- Set the tag length *ulTagBits* in the parameter block.
- Call C_DecryptInit() for **CKM_AES_GCM** mechanism with parameters and key *K*.
- Call C_Decrypt(), or C_DecryptUpdate()*[1] C_DecryptFinal(), for the ciphertext, including the appended tag, obtaining plaintext output.

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

The tag is appended to the cipher text and the least significant bit of the tag is the rightmost bit and the tag bits are the rightmost *ulTagBits* bits.

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

[§] "*" indicates 0 or more calls may be made as required

6.11.4 AES-CCM authenticated Encryption / Decryption

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

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

Encrypt:

- Set the message/data length *ulDataLen* in the parameter block.
- Set the nonce length *ulNonceLen* and the nonce data *pNonce* in the parameter block. *pNonce* may be NULL *if ulNonceLen* is 0.
- Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD m*ay be NULL if *ulAADLen* is 0.
- Set the MAC length *ulMACLen* in the parameter block.
- Call C_EncryptInit() for **CKM_AES_CCM** mechanism with parameters and key *K*.
- Call C_Encrypt(), or C_DecryptUpdate()*[§] C_EncryptFinal(), for the plaintext obtaining ciphertext output obtaining the final ciphertext output and the MAC. The total length of data processed must be *ulDataLen. The output length will be ulDataLen + ulMACLen.*

Decrypt:

- Set the message/data length *ulDataLen* in the parameter block. This length should not include the length of the MAC that is appended to the cipher text.
- Set the nonce length *ulNonceLen* and the nonce data *pNonce* in the parameter block. *pNonce* may be NULL *if ulNonceLen* is 0.
- Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD m*ay be NULL if *ulAADLen* is 0.
- Set the MAC length *ulMACLen* in the parameter block.
- Call C_DecryptInit() for **CKM_AES_CCM** mechanism with parameters and key *K*.
- Call C_Decrypt(), or C_DecryptUpdate()*[§] C_DecryptFinal(), for the ciphertext, including the appended MAC, obtaining plaintext output. The total length of data processed must be *ulDataLen + ulMACLen.*

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

6.12 AES CMAC

Table 45, Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_AES_CMAC_GENERAL		✓					
CKM_AES_CMAC		✓					

[1] SR = SignRecover, VR = VerifyRecover.

6.12.1 Definitions

Mechanisms:

```
CKM_AES_CMAC_GENERAL
CKM_AES_CMAC
```

6.12.2 Mechanism parameters

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

6.12.3 General-length AES-CMAC

General-length AES-CMAC, denoted **CKM_AES_CMAC_GENERAL**, is a mechanism for single- and multiple-part signatures and verification, based on **[**NIST sp800-38b**]** and **[**RFC 4493**]**..

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

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

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

Table 46, General-length AES-CMAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	CKK_AES	any	0-block size, as specified in parameters
C_Verify	CKK_AES	any	0-block size, as specified in parameters

References [NIST sp800-38b] and [RFC 4493] recommend that the output MAC is not truncated to less than 64 bits. The MAC length must be specified before the communication starts, and must not be changed during the lifetime of the key. It is the caller's responsibility to follow these rules.

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

6.12.4 AES-CMAC

AES-CMAC, denoted **CKM_AES_CMAC**, is a special case of the general-length AES-CMAC mechanism. AES-MAC always produces and verifies MACs that are a full block size in length, the default output length specified by [RFC 4493].
Constraints on key types and the length of data are summarized in the following table:

Table 47, AES-CMAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	CKK_AES	any	Block size (16 bytes)
C_Verify	CKK_AES	any	Block size (16 bytes)

References [NIST sp800-38b] and [RFC 4493] recommend that the output MAC is not truncated to less than 64 bits. The MAC length must be specified before the communication starts, and must not be changed during the lifetime of the key. It is the caller's responsibility to follow these rules.

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

6.13 AES Key Wrap

	Functions						
Mechanism	**Encrypt & Decrypt**	**Sign & Verify**	**SR & VR[1]**	**Digest**	**Gen. Key/ Key Pair**	**Wrap & Unwrap**	**Derive**
CKM_AES_KEY_WRAP						✓	
CKM_AES_KEY_WRAP_PAD	✓						

[1]SR = SignRecover, VR = VerifyRecover

6.13.1 Definitions

Mechanisms:

```
CKM_AES_KEY_WRAP
CKM_AES_KEY_WRAP_PAD
```

6.13.2 AES Key Wrap Mechanism parameters

The mechanisms will accept an optional mechanism parameter as the Initialization vector which, if present, must be a fixed size array of 8 bytes, and, if NULL, will use the default initial value defined in Section 2.2.3.1 of [AES KEYWRAP].

The type of this parameter is CK_BYTE_PTR and the pointer points to the array of 8 bytes to be used as the initial value. The length shall be either 0 and the pointer NULL, or 8, and the pointer non-NULL.

6.13.3 AES Key Wrap

The mechanisms support only single-part operations, single part wrapping and unwrapping, and single-part encryption and decryption.

The CKM_AES_KEY_WRAP mechanism can wrap a key of any length. A key whose length is not a multiple of the AES Key Wrap block size (8 bytes) will be zero padded to fit. The CKM_AES_KEY_WRAP mechanism can only encrypt a block of data whose size is an exact multiple of the AES Key Wrap algorithm block size.

The CKM_AES_KEY_WRAP_PAD mechanism can wrap a key or block of data of any length. It does the usual padding of inputs (keys or data blocks) that are not multiples of the AES Key Wrap algorithm block size, always producing wrapped output that is larger than the input key/data to be wrapped. This padding is done by the token before being passed to the AES key wrap algorithm, which adds an 8 byte AES Key Wrap algorithm block of data.

6.14 Key derivation by data encryption – DES & AES

These mechanisms allow derivation of keys using the result of an encryption operation as the key value. They are for use with the C_DeriveKey function.

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_DES_ECB_ENCRYPT_DATA							✓
CKM_DES_CBC_ENCRYPT_DATA							✓
CKM_DES3_ECB_ENCRYPT_DATA							✓
CKM_DES3_CBC_ENCRYPT_DATA							✓
CKM_AES_ECB_ENCRYPT_DATA							✓
CKM_AES_CBC_ENCRYPT_DATA							✓

6.14.1 Definitions

Mechanisms:

```
CKM_DES_ECB_ENCRYPT_DATA
CKM_DES_CBC_ENCRYPT_DATA
CKM_DES3_ECB_ENCRYPT_DATA
CKM_DES3_CBC_ENCRYPT_DATA
CKM_AES_ECB_ENCRYPT_DATA
CKM_AES_CBC_ENCRYPT_DATA

typedef struct CK_DES_CBC_ENCRYPT_DATA_PARAMS {
  CK_BYTE      iv[8];
  CK_BYTE_PTR  pData;
  CK_ULONG     length;
} CK_DES_CBC_ENCRYPT_DATA_PARAMS;
typedef CK_DES_CBC_ENCRYPT_DATA_PARAMS CK_PTR
        CK_DES_CBC_ENCRYPT_DATA_PARAMS_PTR;

typedef struct CK_AES_CBC_ENCRYPT_DATA_PARAMS {
  CK_BYTE      iv[16];
  CK_BYTE_PTR  pData;
  CK_ULONG     length;
} CK_AES_CBC_ENCRYPT_DATA_PARAMS;
typedef CK_AES_CBC_ENCRYPT_DATA_PARAMS CK_PTR
CK_AES_CBC_ENCRYPT_DATA_PARAMS_PTR;
```

6.14.2 Mechanism Parameters

Uses CK_KEY_DERIVATION_STRING_DATA as defined in section 6.27.2

Table 48, Mechanism Parameters

CKM_DES_ECB_ENCRYPT_DATA CKM_DES3_ECB_ENCRYPT_DATA	Uses CK_KEY_DERIVATION_STRING_DATA structure. Parameter is the data to be encrypted and must be a multiple of 8 bytes long.
CKM_AES_ECB_ENCRYPT_DATA	Uses CK_KEY_DERIVATION_STRING_DATA structure. Parameter is the data to be encrypted and must be a multiple of 16 long.
CKM_DES_CBC_ENCRYPT_DATA CKM_DES3_CBC_ENCRYPT_DATA	Uses CK_DES_CBC_ENCRYPT_DATA_PARAMS. Parameter is an 8 byte IV value followed by the data. The data value part must be a multiple of 8 bytes long.
CKM_AES_CBC_ENCRYPT_DATA	Uses CK_AES_CBC_ENCRYPT_DATA_PARAMS. Parameter is an 16 byte IV value followed by the data. The data value part must be a multiple of 16 bytes long.

6.14.3 Mechanism Description

The mechanisms will function by performing the encryption over the data provided using the base key. The resulting cipher text shall be used to create the key value of the resulting key. If not all the cipher text is used then the part discarded will be from the trailing end (least significant bytes) of the cipher text data. The derived key shall be defined by the attribute template supplied but constrained by the length of cipher text available for the key value and other normal PKCS11 derivation constraints.

Attribute template handling, attribute defaulting and key value preparation will operate as per the SHA-1 Key Derivation mechanism in section 6.17.5.

If the data is too short to make the requested key then the mechanism returns CKR_DATA_LENGTH_INVALID.

6.15 Double and Triple-length DES

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_DES2_KEY_GEN					✓		
CKM_DES3_KEY_GEN					✓		
CKM_DES3_ECB	✓					✓	
CKM_DES3_CBC	✓					✓	

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_DES3_CBC_PAD	✓					✓	
CKM_DES3_MAC_GENERAL		✓					
CKM_DES3_MAC		✓					

6.15.1 Definitions

This section defines the key type "CKK_DES2" and "CKK_DES3" for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

```
CKM_DES2_KEY_GEN
CKM_DES3_KEY_GEN
CKM_DES3_ECB
CKM_DES3_CBC
CKM_DES3_MAC
CKM_DES3_MAC_GENERAL
CKM_DES3_CBC_PAD
```

6.15.2 DES2 secret key objects

DES2 secret key objects (object class **CKO_SECRET_KEY,** key type **CKK_DES2**) hold double-length DES keys. The following table defines the DES2 secret key object attributes, in addition to the common attributes defined for this object class:

Table 49, DES2 Secret Key Object Attributes

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

- Refer to [PKCS #11-B] table 15 for footnotes

DES2 keys must always have their parity bits properly set as described in FIPS PUB 46-3 (*i.e.*, each of the DES keys comprising a DES2 key must have its parity bits properly set). Attempting to create or unwrap a DES2 key with incorrect parity will return an error.

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

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_DES2;
CK_UTF8CHAR label[] = "A DES2 secret key object";
CK_BYTE value[16] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
```



```
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_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.

6.15.3 DES3 secret key objects

DES3 secret key objects (object class **CKO_SECRET_KEY,** key type **CKK_DES3**) hold triple-length DES keys. The following table defines the DES3 secret key object attributes, in addition to the common attributes defined for this object class:

Table 50, DES3 Secret Key Object Attributes

Attribute	Data type	Meaning
CKA_VALUE[1,4,6,7]	Byte array	Key value (always 24 bytes long)

- Refer to [PKCS #11-B] table 15 for footnotes

DES3 keys must always have their parity bits properly set as described in FIPS PUB 46-3 (*i.e.*, each of the DES keys comprising a DES3 key must have its parity bits properly set). Attempting to create or unwrap a DES3 key with incorrect parity will return an error.

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

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_DES3;
CK_UTF8CHAR label[] = "A DES3 secret key object";
CK_BYTE value[24] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
  {CKA_CLASS, &class, sizeof(class)},
  {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
  {CKA_TOKEN, &true, sizeof(true)},
  {CKA_LABEL, label, sizeof(label)-1},
  {CKA_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.

6.15.4 Double-length DES key generation

The double-length DES key generation mechanism, denoted **CKM_DES2_KEY_GEN**, is a key generation mechanism for double-length DES keys. The DES keys making up a double-length DES key both have their parity bits set properly, as specified in FIPS PUB 46-3.

It does not have a parameter.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the double-length DES key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

Double-length DES keys can be used with all the same mechanisms as triple-DES keys: **CKM_DES3_ECB**, **CKM_DES3_CBC**, **CKM_DES3_CBC_PAD**, **CKM_DES3_MAC_GENERAL**, and **CKM_DES3_MAC**. Triple-DES encryption with a double-length DES key is equivalent to encryption with a triple-length DES key with K1=K3 as specified in FIPS PUB 46-3.

When double-length DES keys are generated, it is token-dependent whether or not it is possible for either of the component DES keys to be "weak" or "semi-weak" keys.

6.15.5 Triple-length DES Order of Operations

Triple-length DES encryptions are carried out as specified in FIPS PUB 46-3: encrypt, decrypt, encrypt. Decryptions are carried out with the opposite three steps: decrypt, encrypt, decrypt. The mathematical representations of the encrypt and decrypt operations are as follows:

$$\text{DES3-E}(\{K1,K2,K3\}, P) = E(K3, D(K2, E(K1, P)))$$

$$\text{DES3-D}(\{K1,K2,K3\}, C) = D(K1, E(K2, D(K3, P)))$$

6.15.6 Triple-length DES in CBC Mode

Triple-length DES operations in CBC mode, with double or triple-length keys, are performed using outer CBC as defined in X9.52. X9.52 describes this mode as TCBC. The mathematical representations of the CBC encrypt and decrypt operations are as follows:

DES3-CBC-E({K1,K2,K3}, P) = E(K3, D(K2, E(K1, P + I)))

DES3-CBC-D({K1,K2,K3}, C) = D(K1, E(K2, D(K3, P))) + I

The value *I* is either an 8-byte initialization vector or the previous block of cipher text that is added to the current input block. The addition operation is used is addition modulo-2 (XOR).

6.15.7 DES and Triple length DES in OFB Mode

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_DES_OFB64	✓						
CKM_DES_ OFB8	✓						
CKM_DES_ CFB64	✓						
CKM_DES_ CFB8	✓						

Cipher DES has a output feedback mode, DES-OFB, denoted **CKM_DES_OFB8** and **CKM_DES_OFB64**. It is a mechanism for single and multiple-part encryption and decryption with DES.

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

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

Table 51, OFB: Key And Data Length

Function	Key type	Input length	Output length	Comments
C_Encrypt	CKK_DES, CKK_DES2, CKK_DES3	any	same as input length	no final part
C_Decrypt	CKK_DES, CKK_DES2, CKK_DES3	any	same as input length	no final part

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

6.15.8 DES and Triple length DES in CFB Mode

Cipher DES has a cipher feedback mode, DES-CFB, denoted **CKM_DES_CFB8** and **CKM_DES_CFB64**. It is a mechanism for single and multiple-part encryption and decryption with DES.

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

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

Table 52, CFB: Key And Data Length

Function	Key type	Input length	Output length	Comments
C_Encrypt	CKK_DES, CKK_DES2, CKK_DES3	any	same as input length	no final part
C_Decrypt	CKK_DES, CKK_DES2, CKK_DES3	any	same as input length	no final part

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

6.16 Double and Triple-length DES CMAC

Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_DES3_CMAC_GENERAL		✓					
CKM_DES3_CMAC		✓					

[1] SR = SignRecover, VR = VerifyRecover.

The following additional DES3 mechanisms have been added.

6.16.1 Definitions

Mechanisms:

```
CKM_DES3_CMAC_GENERAL
CKM_DES3_CMAC
```

6.16.2 Mechanism parameters

CKM_DES3_CMAC_GENERAL uses the existing **CK_MAC_GENERAL_PARAMS** structure. CKM_DES3_CMAC does not use a mechanism parameter.

6.16.3 General-length DES3-MAC

General-length DES3-CMAC, denoted **CKM_DES3_CMAC_GENERAL**, is a mechanism for single- and multiple-part signatures and verification with DES3 or DES2 keys, based on [NIST sp800-38b].

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

The output bytes from this mechanism are taken from the start of the final DES3 cipher block produced in the MACing process.

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

Table 53, General-length DES3-CMAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	CKK_DES3 CKK_DES2	any	0-block size, as specified in parameters
C_Verify	CKK_DES3 CKK_DES2	any	0-block size, as specified in parameters

Reference [NIST sp800-38b] recommends that the output MAC is not truncated to less than 64 bits (which means using the entire block for DES). The MAC length must be specified before the communication starts, and must not be changed during the lifetime of the key. It is the caller's responsibility to follow these rules.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure.are not used

6.16.4 DES3-CMAC

DES3-CMAC, denoted **CKM_DES3_CMAC**, is a special case of the general-length DES3-CMAC mechanism. DES3-MAC always produces and verifies MACs that are a full block size in length, since the DES3 block lenth is the minimum output length recommended by [NIST sp800-38b].
Constraints on key types and the length of data are summarized in the following table:

Table 54, DES3-CMAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	CKK_DES3 CKK_DES2	any	Block size (8 bytes)
C_Verify	CKK_DES3 CKK_DES2	any	Block size (8 bytes)

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure are not used.

6.17 SHA-1

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA_1				✓			
CKM_SHA_1_HMAC_GENERAL		✓					
CKM_SHA_1_HMAC		✓					
CKM_SHA1_KEY_DERIVATION							✓

6.17.1 Definitions

```
CKM_SHA_1
CKM_SHA_1_HMAC
CKM_SHA_1_HMAC_GENERAL
CKM_SHA1_KEY_DERIVATION

CKK_SHA_1_HMAC
```

6.17.2 SHA-1 digest

The SHA-1 mechanism, denoted **CKM_SHA_1**, is a mechanism for message digesting, following the Secure Hash Algorithm with a 160-bit message digest defined in FIPS PUB 180-2.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 55, SHA-1: Data Length

Function	Input length	Digest length
C_Digest	any	20

6.17.3 General-length SHA-1-HMAC

The general-length SHA-1-HMAC mechanism, denoted **CKM_SHA_1_HMAC_GENERAL**, is a mechanism for signatures and verification. It uses the HMAC construction, based on the SHA-1 hash function. The keys it uses are generic secret keys and CKK_SHA_1_HMAC.

It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired output. This length should be in the range 0-20 (the output size of SHA-1 is 20 bytes). Signatures (MACs) produced by this mechanism will be taken from the start of the full 20-byte HMAC output.

Table 56, General-length SHA-1-HMAC: Key And Data Length

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

6.17.4 SHA-1-HMAC

The SHA-1-HMAC mechanism, denoted **CKM_SHA_1_HMAC**, is a special case of the general-length SHA-1-HMAC mechanism in Section 6.17.3.

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

6.17.5 SHA-1 key derivation

SHA-1 key derivation, denoted **CKM_SHA1_KEY_DERIVATION**, is a mechanism which provides the capability of deriving a secret key by digesting the value of another secret key with SHA-1.

The value of the base key is digested once, and the result is used to make the value of derived secret key.

- If no length or key type is provided in the template, then the key produced by this mechanism will be a generic secret key. Its length will be 20 bytes (the output size of SHA-1).

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

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

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

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

If the requested type of key requires more than 20 bytes, such as DES3, an error is generated.

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

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

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

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

6.18 SHA-224

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA224				✓			

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA224_HMAC		✓					
CKM_SHA224_HMAC_GENERAL		✓					
CKM_SHA224_RSA_PKCS		✓					
CKM_SHA224_RSA_PKCS_PSS		✓					
CKM_SHA224_KEY_DERIVATION							✓

6.18.1 Definitions

```
CKM_SHA224
CKM_SHA224_HMAC
CKM_SHA224_HMAC_GENERAL
CKM_SHA224_KEY_DERIVATION

CKK_SHA224_HMAC
```

6.18.2 SHA-224 digest

The SHA-224 mechanism, denoted **CKM_SHA224**, is a mechanism for message digesting, following the Secure Hash Algorithm with a 224-bit message digest defined in 0.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 57, SHA-224: Data Length

Function	Input length	Digest length
C_Digest	any	28

6.18.3 General-length SHA-224-HMAC

The general-length SHA-224-HMAC mechanism, denoted **CKM_SHA224_HMAC_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism except that it uses the HMAC construction based on the SHA-224 hash function and length of the output should be in the range 0-28. The keys it uses are generic secret keys and CKK_SHA224_HMAC. FIPS-198 compliant tokens may require the key length to be at least 14 bytes; that is, half the size of the SHA-224 hash output.

It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired output. This length should be in the range 0-28 (the output size of SHA-

224 is 28 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 14 (half the maximum length). Signatures (MACs) produced by this mechanism will be taken from the start of the full 28-byte HMAC output.

Table 58, General-length SHA-224-HMAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	generic secret	Any	0-28, depending on parameters
C_Verify	generic secret	Any	0-28, depending on parameters

6.18.4 SHA-224-HMAC

The SHA-224-HMAC mechanism, denoted **CKM_SHA224_HMAC**, is a special case of the general-length SHA-224-HMAC mechanism.

It has no parameter, and always produces an output of length 28.

6.18.5 SHA-224 key derivation

SHA-224 key derivation, denoted **CKM_SHA224_KEY_DERIVATION**, is the same as the SHA-1 key derivation mechanism in Section 12.21.5 except that it uses the SHA-224 hash function and the relevant length is 28 bytes.

6.19 SHA-256

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA256				✓			
CKM_SHA256_HMAC_GENERAL		✓					
CKM_SHA256_HMAC		✓					
CKM_SHA256_KEY_DERIVATION							✓

6.19.1 Definitions

```
CKM_SHA256
CKM_SHA256_HMAC
CKM_SHA256_HMAC_GENERAL
CKM_SHA256_KEY_DERIVATION

CKK_SHA256_HMAC
```


6.19.2 SHA-256 digest

The SHA-256 mechanism, denoted **CKM_SHA256**, is a mechanism for message digesting, following the Secure Hash Algorithm with a 256-bit message digest defined in FIPS PUB 180-2.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 59, SHA-256: Data Length

Function	Input length	Digest length
C_Digest	any	32

6.19.3 General-length SHA-256-HMAC

The general-length SHA-256-HMAC mechanism, denoted **CKM_SHA256_HMAC_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in Section 6.17.3, except that it uses the HMAC construction based on the SHA-256 hash function and length of the output should be in the range 0-32. The keys it uses are generic secret keys and CKK_SHA256_HMAC. FIPS-198 compliant tokens may require the key length to be at least 16 bytes; that is, half the size of the SHA-256 hash output.

It has a parameter, a CK_MAC_GENERAL_PARAMS, which holds the length in bytes of the desired output. This length should be in the range 0-32 (the output size of SHA-256 is 32 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 16 (half the maximum length). Signatures (MACs) produced by this mechanism will be taken from the start of the full 32-byte HMAC output.

Table 60, General-length SHA-256-HMAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	generic secret	Any	0-32, depending on parameters
C_Verify	generic secret	Any	0-32, depending on parameters

6.19.4 SHA-256-HMAC

The SHA-256-HMAC mechanism, denoted **CKM_SHA256_HMAC**, is a special case of the general-length SHA-256-HMAC mechanism in Section 6.19.3.

It has no parameter, and always produces an output of length 32.

6.19.5 SHA-256 key derivation

SHA-256 key derivation, denoted **CKM_SHA256_KEY_DERIVATION**, is the same as the SHA-1 key derivation mechanism in Section 6.17.5, except that it uses the SHA-256 hash function and the relevant length is 32 bytes.

6.20 SHA-384

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA384				✓			
CKM_SHA384_HMAC_GENERAL		✓					
CKM_SHA384_HMAC		✓					
CKM_SHA384_KEY_DERIVATION							✓

6.20.1 Definitions

```
CKM_SHA384
CKM_SHA384_HMAC
CKM_SHA384_HMAC_GENERAL
CKM_SHA384_KEY_DERIVATION

CKK_SHA384_HMAC
```

6.20.2 SHA-384 digest

The SHA-384 mechanism, denoted **CKM_SHA384**, is a mechanism for message digesting, following the Secure Hash Algorithm with a 384-bit message digest defined in FIPS PUB 180-2.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 61, SHA-384: Data Length

Function	Input length	Digest length
C_Digest	any	48

6.20.3 General-length SHA-384-HMAC

The general-length SHA-384-HMAC mechanism, denoted **CKM_SHA384_HMAC_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in Section 6.17.3, except that it uses the HMAC construction based on the SHA-384 hash function and length of the output should be in the range 0-48.

6.20.4 SHA-384-HMAC

The SHA-384-HMAC mechanism, denoted **CKM_SHA384_HMAC**, is a special case of the general-length SHA-384-HMAC mechanism.

It has no parameter, and always produces an output of length 48.

6.20.5 SHA-384 key derivation

SHA-384 key derivation, denoted **CKM_SHA384_KEY_DERIVATION**, is the same as the SHA-1 key derivation mechanism in Section 6.17.5, except that it uses the SHA-384 hash function and the relevant length is 48 bytes.

6.21 SHA-512

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA512				✓			
CKM_SHA512_HMAC_GENERAL		✓					
CKM_SHA512_HMAC		✓					
CKM_SHA512_KEY_DERIVATION							✓

6.21.1 Definitions

```
CKM_SHA512
CKM_SHA512_HMAC
CKM_SHA512_HMAC_GENERAL
CKM_SHA512_KEY_DERIVATION

CKK_SHA512_HMAC
```

6.21.2 SHA-512 digest

The SHA-512 mechanism, denoted **CKM_SHA512**, is a mechanism for message digesting, following the Secure Hash Algorithm with a 512-bit message digest defined in FIPS PUB 180-2.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 62, SHA-512: Data Length

Function	Input length	Digest length
C_Digest	any	64

6.21.3 General-length SHA-512-HMAC

The general-length SHA-512-HMAC mechanism, denoted **CKM_SHA512_HMAC_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in Section 6.17.3, except that it uses the HMAC construction based on the SHA-512 hash function and length of the output should be in the range 0-64.

6.21.4 SHA-512-HMAC

The SHA-512-HMAC mechanism, denoted **CKM_SHA512_HMAC**, is a special case of the general-length SHA-512-HMAC mechanism.

It has no parameter, and always produces an output of length 64.

6.21.5 SHA-512 key derivation

SHA-512 key derivation, denoted **CKM_SHA512_KEY_DERIVATION**, is the same as the SHA-1 key derivation mechanism in Section 6.17.5, except that it uses the SHA-512 hash function and the relevant length is 64 bytes.

6.22 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.

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_PBE_SHA1_DES3_EDE_CBC					✓		
CKM_PBE_SHA1_DES2_EDE_CBC					✓		
CKM_PBA_SHA1_WITH_SHA1_HMAC					✓		
CKM_PKCS5_PBKD2					✓		

6.22.1 Definitions

Mechanisms:

```
CKM_PBE_SHA1_DES3_EDE_CBC
CKM_PBE_SHA1_DES2_EDE_CBC
CKM_PKCS5_PBKD2
CKM_PBA_SHA1_WITH_SHA1_HMAC
```

6.22.2 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**.

6.22.3 PKCS #5 PBKDF2 key generation mechanism parameters

♦ **CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE; CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE_PTR**

CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE is used to indicate the Pseudo-Random Function (PRF) used to generate key bits using PKCS #5 PBKDF2. It is defined as follows:

```
typedef CK_ULONG
        CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE;
```

The following PRFs are defined in PKCS #5 v2.0. The following table lists the defined functions.

Table 63, PKCS #5 PBKDF2 Key Generation: Pseudo-random functions

PRF Identifier	Value	Parameter Type
CKP_PKCS5_PBKD2_HMAC_SHA1	0x00000001	No Parameter. *pPrfData* must be NULL and *ulPrfDataLen* must be zero.
CKP_PKCS5_PBKD2_HMAC_GOSTR3411	0x00000002	This PRF uses GOST R34.11-94 hash to produce secret key value. *pPrfData* should point to DER-encoded OID, indicating GOSTR34.11-94 parameters. *ulPrfDataLen* holds encoded OID length in bytes. If *pPrfData* is set to NULL_PTR, then *id-GostR3411-94-CryptoProParamSet* parameters will be used (RFC 4357, 11.2), and *ulPrfDataLen* must be 0.

CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE_PTR is a pointer to a **CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE**.

♦ **CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE; CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE_PTR**

CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE is used to indicate the source of the salt value when deriving a key using PKCS #5 PBKDF2. It is defined as follows:

```
typedef CK_ULONG CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE;
```


The following salt value sources are defined in PKCS #5 v2.0. The following table lists the defined sources along with the corresponding data type for the *pSaltSourceData* field in the **CK_PKCS5_PBKD2_PARAM** structure defined below.

Table 64, PKCS #5 PBKDF2 Key Generation: Salt sources

Source Identifier	Value	Data Type
CKZ_SALT_SPECIFIED	0x00000001	Array of CK_BYTE containing the value of the salt value.

CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE_PTR is a pointer to a **CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE**.

- **CK_ PKCS5_PBKD2_PARAMS; CK_PKCS5_PBKD2_PARAMS_PTR**

CK_PKCS5_PBKD2_PARAMS is a structure that provides the parameters to the **CKM_PKCS5_PBKD2** mechanism. The structure is defined as follows:

```
typedef struct CK_PKCS5_PBKD2_PARAMS {
   CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE saltSource;
   CK_VOID_PTR pSaltSourceData;
   CK_ULONG ulSaltSourceDataLen;
   CK_ULONG iterations;
   CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE prf;
   CK_VOID_PTR pPrfData;
   CK_ULONG ulPrfDataLen;  CK_UTF8CHAR_PTR pPassword;
   CK_ULONG_PTR ulPasswordLen;
} CK_PKCS5_PBKD2_PARAMS;
```

The fields of the structure have the following meanings:

saltSource source of the salt value

pSaltSourceData data used as the input for the salt source

ulSaltSourceDataLen length of the salt source input

iterations number of iterations to perform when generating each block of random data

prf pseudo-random function to used to generate the key

pPrfData data used as the input for PRF in addition to the salt value

ulPrfDataLen length of the input data for the PRF

pPassword	points to the password to be used in the PBE key generation
ulPasswordLen	length in bytes of the password information

CK_PKCS5_PBKD2_PARAMS_PTR is a pointer to a **CK_PKCS5_PBKD2_PARAMS**.

6.22.4 PKCS #5 PBKD2 key generation

PKCS #5 PBKDF2 key generation, denoted **CKM_PKCS5_PBKD2**, is a mechanism used for generating a secret key from a password and a salt value. This functionality is defined in PKCS#5 as PBKDF2.

It has a parameter, a **CK_PKCS5_PBKD2_PARAMS** structure. The parameter specifies the salt value source, pseudo-random function, and iteration count used to generate the new key.

Since this mechanism can be used to generate any type of secret key, new key templates must contain the **CKA_KEY_TYPE** and **CKA_VALUE_LEN** attributes. If the key type has a fixed length the **CKA_VALUE_LEN** attribute may be omitted.

6.23 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 $f\colon \mathbf{Z}_2^u \times \mathbf{Z}_2^v \rightarrow \mathbf{Z}_2^u$ (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 $I_0, I_1, \ldots, I_{k-1}$ 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 $A_1, A_2, \ldots, A_j$ together to form a pseudo-random bit string, A.

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

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

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

6.23.1 SHA-1-PBE for 3-key triple-DES-CBC

SHA-1-PBE for 3-key triple-DES-CBC, denoted **CKM_PBE_SHA1_DES3_EDE_CBC**, is a mechanism used for generating a 3-key triple-DES secret key and IV from a password and a salt value by using the SHA-1 digest algorithm and an iteration count. The method used to generate the key and IV is described above. Each byte of the key produced will have its low-order bit adjusted, if necessary, so that a valid 3-key triple-DES key with proper parity bits is obtained.

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

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

6.23.2 SHA-1-PBE for 2-key triple-DES-CBC

SHA-1-PBE for 2-key triple-DES-CBC, denoted **CKM_PBE_SHA1_DES2_EDE_CBC**, is a mechanism used for generating a 2-key triple-DES secret key and IV from a password and a salt value by using the SHA-1 digest algorithm and an iteration count. The method used to generate the key and IV is described above. Each byte of the key produced will have its low-order bit adjusted, if necessary, so that a valid 2-key triple-DES key with proper parity bits is obtained.

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

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

6.23.3 SHA-1-PBA for SHA-1-HMAC

SHA-1-PBA for SHA-1-HMAC, denoted **CKM_PBA_SHA1_WITH_SHA1_HMAC**, is a mechanism used for generating a 160-bit generic secret key from a password and a salt value by using the SHA-1 digest algorithm and an iteration count. The method used to generate the key is described above.

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

The key generated by this mechanism will typically be used for computing a SHA-1 HMAC to perform password-based authentication (not *password-based encryption*). At the time of this writing, this is primarily done to ensure the integrity of a PKCS #12 PDU.

6.24 SSL

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SSL3_PRE_MASTER_KEY_GEN					✓		
CKM_SSL3_MASTER_KEY_DERIVE							✓
CKM_SSL3_MASTER_KEY_DERIVE_DH							✓
CKM_SSL3_KEY_AND_MAC_DERIVE							✓
CKM_SSL3_MD5_MAC		✓					
CKM_SSL3_SHA1_MAC		✓					

6.24.1 Definitions

Mechanisms:

```
CKM_SSL3_PRE_MASTER_KEY_GEN
CKM_SSL3_MASTER_KEY_DERIVE
CKM_SSL3_KEY_AND_MAC_DERIVE
CKM_SSL3_MASTER_KEY_DERIVE_DH
CKM_SSL3_MD5_MAC
CKM_SSL3_SHA1_MAC
```

6.24.2 SSL mechanism parameters

♦ **CK_SSL3_RANDOM_DATA**

CK_SSL3_RANDOM_DATA is a structure which provides information about the random data of a client and a server in an SSL context. This structure is used by both the **CKM_SSL3_MASTER_KEY_DERIVE** and the **CKM_SSL3_KEY_AND_MAC_DERIVE** mechanisms. It is defined as follows:

```
typedef struct CK_SSL3_RANDOM_DATA {
  CK_BYTE_PTR pClientRandom;
  CK_ULONG ulClientRandomLen;
  CK_BYTE_PTR pServerRandom;
  CK_ULONG ulServerRandomLen;
} CK_SSL3_RANDOM_DATA;
```

The fields of the structure have the following meanings:

pClientRandom pointer to the client's random data

ulClientRandomLen length in bytes of the client's random data

pServerRandom pointer to the server's random data *ulServerRandomLen* length in bytes of the server's random data

♦ **CK_SSL3_MASTER_KEY_DERIVE_PARAMS; CK_SSL3_MASTER_KEY_DERIVE_PARAMS_PTR**

CK_SSL3_MASTER_KEY_DERIVE_PARAMS is a structure that provides the parameters to the **CKM_SSL3_MASTER_KEY_DERIVE** mechanism. It is defined as follows:

```
typedef struct CK_SSL3_MASTER_KEY_DERIVE_PARAMS {
  CK_SSL3_RANDOM_DATA RandomInfo;
  CK_VERSION_PTR pVersion;
} CK_SSL3_MASTER_KEY_DERIVE_PARAMS;
```

The fields of the structure have the following meanings:

RandomInfo client's and server's random data information.

pVersion pointer to a **CK_VERSION** structure which receives the SSL protocol version information

CK_SSL3_MASTER_KEY_DERIVE_PARAMS_PTR is a pointer to a **CK_SSL3_MASTER_KEY_DERIVE_PARAMS**.

♦ **CK_SSL3_KEY_MAT_OUT; CK_SSL3_KEY_MAT_OUT_PTR**

CK_SSL3_KEY_MAT_OUT is a structure that contains the resulting key handles and initialization vectors after performing a C_DeriveKey function with the **CKM_SSL3_KEY_AND_MAC_DERIVE** mechanism. It is defined as follows:

```
typedef struct CK_SSL3_KEY_MAT_OUT {
  CK_OBJECT_HANDLE hClientMacSecret;
  CK_OBJECT_HANDLE hServerMacSecret;
  CK_OBJECT_HANDLE hClientKey;
  CK_OBJECT_HANDLE hServerKey;
  CK_BYTE_PTR pIVClient;
  CK_BYTE_PTR pIVServer;
} CK_SSL3_KEY_MAT_OUT;
```

The fields of the structure have the following meanings:

hClientMacSecret key handle for the resulting Client MAC Secret key

hServerMacSecret key handle for the resulting Server MAC Secret key

hClientKey key handle for the resulting Client Secret key

hServerKey	key handle for the resulting Server Secret key
pIVClient	pointer to a location which receives the initialization vector (IV) created for the client (if any)
pIVServer	pointer to a location which receives the initialization vector (IV) created for the server (if any)

CK_SSL3_KEY_MAT_OUT_PTR is a pointer to a **CK_SSL3_KEY_MAT_OUT**.

♦ **CK_SSL3_KEY_MAT_PARAMS; CK_SSL3_KEY_MAT_PARAMS_PTR**

CK_SSL3_KEY_MAT_PARAMS is a structure that provides the parameters to the **CKM_SSL3_KEY_AND_MAC_DERIVE** mechanism. It is defined as follows:

```
typedef struct CK_SSL3_KEY_MAT_PARAMS {
  CK_ULONG ulMacSizeInBits;
  CK_ULONG ulKeySizeInBits;
  CK_ULONG ulIVSizeInBits;
  CK_BBOOL bIsExport;
  CK_SSL3_RANDOM_DATA RandomInfo;
  CK_SSL3_KEY_MAT_OUT_PTR pReturnedKeyMaterial;
} CK_SSL3_KEY_MAT_PARAMS;
```

The fields of the structure have the following meanings:

ulMacSizeInBits	the length (in bits) of the MACing keys agreed upon during the protocol handshake phase
ulKeySizeInBits	the length (in bits) of the secret keys agreed upon during the protocol handshake phase
ulIVSizeInBits	the length (in bits) of the IV agreed upon during the protocol handshake phase. If no IV is required, the length should be set to 0
bIsExport	a Boolean value which indicates whether the keys have to be derived for an export version of the protocol
RandomInfo	client's and server's random data information.
pReturnedKeyMaterial	points to a **CK_SSL3_KEY_MAT_OUT** structures which receives the handles for the keys generated and the IVs

CK_SSL3_KEY_MAT_PARAMS_PTR is a pointer to a **CK_SSL3_KEY_MAT_PARAMS**.

6.24.3 Pre_master key generation

Pre_master key generation in SSL 3.0, denoted **CKM_SSL3_PRE_MASTER_KEY_GEN**, is a mechanism which generates a 48-byte generic secret key. It is used to produce the "pre_master" key used in SSL version 3.0 for RSA-like cipher suites.

It has one parameter, a **CK_VERSION** structure, which provides the client's SSL version number.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.

The template sent along with this mechanism during a **C_GenerateKey** call may indicate that the object class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN** attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK_MECHANISM_INFO** structure both indicate 48 bytes.

6.24.4 Master key derivation

Master key derivation in SSL 3.0, denoted **CKM_SSL3_MASTER_KEY_DERIVE**, is a mechanism used to derive one 48-byte generic secret key from another 48-byte generic secret key. It is used to produce the "master_secret" key used in the SSL protocol from the "pre_master" key. This mechanism returns the value of the client version, which is built into the "pre_master" key as well as a handle to the derived "master_secret" key.

It has a parameter, a **CK_SSL3_MASTER_KEY_DERIVE_PARAMS** structure, which allows for the passing of random data to the token as well as the returning of the protocol version number which is part of the pre-master key. This structure is defined in Section 6.24.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may be specified in the template; otherwise they are assigned default values.

The template sent along with this mechanism during a **C_DeriveKey** call may indicate that the object class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN** attribute has value 48.

However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

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

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

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK_MECHANISM_INFO** structure both indicate 48 bytes.

Note that the **CK_VERSION** structure pointed to by the **CK_SSL3_MASTER_KEY_DERIVE_PARAMS** structure's *pVersion* field will be modified by the **C_DeriveKey** call. In particular, when the call returns, this structure will hold the SSL version associated with the supplied pre_master key.

Note that this mechanism is only useable for cipher suites that use a 48-byte "pre_master" secret with an embedded version number. This includes the RSA cipher suites, but excludes the Diffie-Hellman cipher suites.

6.24.5 Master key derivation for Diffie-Hellman

Master key derivation for Diffie-Hellman in SSL 3.0, denoted **CKM_SSL3_MASTER_KEY_DERIVE_DH**, is a mechanism used to derive one 48-byte generic secret key from another arbitrary length generic secret key. It is used to produce the "master_secret" key used in the SSL protocol from the "pre_master" key.

It has a parameter, a **CK_SSL3_MASTER_KEY_DERIVE_PARAMS** structure, which allows for the passing of random data to the token. This structure is defined in Section 6.24. The *pVersion* field of the structure must be set to NULL_PTR since the version number is not embedded in the "pre_master" key as it is for RSA-like cipher suites.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.

The template sent along with this mechanism during a **C_DeriveKey** call may indicate that the object class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN** attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

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

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

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

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

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK_MECHANISM_INFO** structure both indicate 48 bytes.

Note that this mechanism is only useable for cipher suites that do not use a fixed length 48-byte “pre_master” secret with an embedded version number. This includes the Diffie-Hellman cipher suites, but excludes the RSA cipher suites.

6.24.6 Key and MAC derivation

Key, MAC and IV derivation in SSL 3.0, denoted **CKM_SSL3_KEY_AND_MAC_DERIVE**, is a mechanism used to derive the appropriate cryptographic keying material used by a "CipherSuite" from the "master_secret" key and random data. This mechanism returns the key handles for the keys generated in the process, as well as the IVs created.

It has a parameter, a **CK_SSL3_KEY_MAT_PARAMS** structure, which allows for the passing of random data as well as the characteristic of the cryptographic material for the given CipherSuite and a pointer to a structure which receives the handles and IVs which were generated. This structure is defined in Section 6.24.

This mechanism contributes to the creation of four distinct keys on the token and returns two IVs (if IVs are requested by the caller) back to the caller. The keys are all given an object class of **CKO_SECRET_KEY**.

The two MACing keys ("client_write_MAC_secret" and "server_write_MAC_secret") are always given a type of **CKK_GENERIC_SECRET**. They are flagged as valid for signing, verification, and derivation operations.

The other two keys ("client_write_key" and "server_write_key") are typed according to information found in the template sent along with this mechanism during a **C_DeriveKey** function call. By default, they are flagged as valid for encryption, decryption, and derivation operations.

IVs will be generated and returned if the *ulIVSizeInBits* field of the **CK_SSL_KEY_MAT_PARAMS** field has a nonzero value. If they are generated, their length in bits will agree with the value in the *ulIVSizeInBits* field.

All four keys inherit the values of the **CKA_SENSITIVE**, **CKA_ALWAYS_SENSITIVE**, **CKA_EXTRACTABLE**, and **CKA_NEVER_EXTRACTABLE** attributes from the base key. The template provided to **C_DeriveKey** may not specify values for any of these attributes which differ from those held by the base key.

Note that the **CK_SSL3_KEY_MAT_OUT** structure pointed to by the **CK_SSL3_KEY_MAT_PARAMS** structure's *pReturnedKeyMaterial* field will be modified by the **C_DeriveKey** call. In particular, the four key handle fields in the **CK_SSL3_KEY_MAT_OUT** structure will be modified to hold handles to the newly-created keys; in addition, the buffers pointed to by the **CK_SSL3_KEY_MAT_OUT** structure's *pIVClient* and *pIVServer* fields will have IVs returned in them (if IVs are requested by the caller). Therefore, these two fields must point to buffers with sufficient space to hold any IVs that will be returned.

This mechanism departs from the other key derivation mechanisms in Cryptoki in its returned information. For most key-derivation mechanisms, **C_DeriveKey** returns a single key handle as a result of a successful completion. However, since the **CKM_SSL3_KEY_AND_MAC_DERIVE** mechanism returns all of its key handles in the **CK_SSL3_KEY_MAT_OUT** structure pointed to by the **CK_SSL3_KEY_MAT_PARAMS** structure specified as the mechanism parameter, the parameter *phKey* passed to **C_DeriveKey** is unnecessary, and should be a NULL_PTR.

If a call to **C_DeriveKey** with this mechanism fails, then *none* of the four keys will be created on the token.

6.24.7 MD5 MACing in SSL 3.0

MD5 MACing in SSL3.0, denoted **CKM_SSL3_MD5_MAC**, is a mechanism for single- and multiple-part signatures (data authentication) and verification using MD5, based on the SSL 3.0 protocol. This technique is very similar to the HMAC technique.

It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which specifies the length in bytes of the signatures produced by this mechanism.

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

Table 65, MD5 MACing in SSL 3.0: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	generic secret	any	4-8, depending on parameters
C_Verify	generic secret	any	4-8, depending on parameters

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

6.24.8 SHA-1 MACing in SSL 3.0

SHA-1 MACing in SSL3.0, denoted **CKM_SSL3_SHA1_MAC**, is a mechanism for single- and multiple-part signatures (data authentication) and verification using SHA-1, based on the SSL 3.0 protocol. This technique is very similar to the HMAC technique.

It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which specifies the length in bytes of the signatures produced by this mechanism.

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

Table 66, SHA-1 MACing in SSL 3.0: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	generic secret	any	4-8, depending on parameters
C_Verify	generic secret	any	4-8, depending on parameters

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

6.25 TLS

Details can be found in [TLS].

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_TLS_PRE_MASTER_KEY_GEN					✓		
CKM_TLS_MASTER_KEY_DERIVE							✓
CKM_TLS_MASTER_KEY_DERIVE_DH							✓
CKM_TLS_KEY_AND_MAC_DERIVE							✓
CKM_TLS_PRF							✓

6.25.1 Definitions

Mechanisms:

```
CKM_TLS_PRE_MASTER_KEY_GEN
CKM_TLS_MASTER_KEY_DERIVE
CKM_TLS_KEY_AND_MAC_DERIVE
CKM_TLS_MASTER_KEY_DERIVE_DH
CKM_TLS_PRF
```

6.25.2 TLS mechanism parameters

- **CK_TLS_PRF_PARAMS; CK_TLS_PRF_PARAMS_PTR**

CK_TLS_PRF_PARAMS is a structure, which provides the parameters to the **CKM_TLS_PRF** mechanism. It is defined as follows:

```
typedef struct CK_TLS_PRF_PARAMS {
  CK_BYTE_PTR  pSeed;
  CK_ULONG     ulSeedLen;
  CK_BYTE_PTR  pLabel;
  CK_ULONG     ulLabelLen;
  CK_BYTE_PTR  pOutput;
  CK_ULONG_PTR pulOutputLen;
} CK_TLS_PRF_PARAMS;
```

The fields of the structure have the following meanings:

pSeed	pointer to the input seed
ulSeedLen	length in bytes of the input seed
pLabel	pointer to the identifying label
ulLabelLen	length in bytes of the identifying label
pOutput	pointer receiving the output of the operation
pulOutputLen	pointer to the length in bytes that the output to be created shall have, has to hold the desired length as input and will receive the calculated length as output

CK_TLS_PRF_PARAMS_PTR is a pointer to a **CK_TLS_PRF_PARAMS**.

6.25.3 TLS PRF (pseudorandom function)

PRF (pseudo random function) in TLS, denoted **CKM_TLS_PRF**, is a mechanism used to produce a securely generated pseudo-random output of arbitrary length. The keys it uses are generic secret keys.

It has a parameter, a **CK_TLS_PRF_PARAMS** structure, which allows for the passing of the input seed and its length, the passing of an identifying label and its length and the passing of the length of the output to the token and for receiving the output.

This mechanism produces securely generated pseudo-random output of the length specified in the parameter.

This mechanism departs from the other key derivation mechanisms in Cryptoki in not using the template sent along with this mechanism during a **C_DeriveKey** function call, which means the template shall be a NULL_PTR. For most key-derivation mechanisms, **C_DeriveKey** returns a single key handle as a result of a successful completion.

However, since the **CKM_TLS_PRF** mechanism returns the requested number of output bytes in the **CK_TLS_PRF_PARAMS** structure specified as the mechanism parameter, the parameter *phKey* passed to **C_DeriveKey** is unnecessary, and should be a NULL_PTR.

If a call to **C_DeriveKey** with this mechanism fails, then no output will be generated.

6.25.4 Pre_master key generation

Pre_master key generation in TLS 1.0, denoted **CKM_TLS_PRE_MASTER_KEY_GEN**, is a mechanism which generates a 48-byte generic secret key. It is used to produce the "pre_master" key used in TLS version 1.0 for RSA-like cipher suites.

It has one parameter, a CK_VERSION structure, which provides the client's TLS version number. The CK_VERSION structure should have the version value {3, 1} for TLS version 1.0.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.

The template sent along with this mechanism during a **C_GenerateKey** call may indicate that the object class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN** attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK_MECHANISM_INFO** structure both indicate 48 bytes.

6.25.5 Master key derivation

Master key derivation in TLS 1.0, denoted **CKM_TLS_MASTER_KEY_DERIVE**, is a mechanism used to derive one 48-byte generic secret key from another 48-byte generic secret key. It is used to produce the "master_secret" key used in the TLS protocol from the "pre_master" key. This mechanism returns the value of the client version, which is built into the "pre_master" key as well as a handle to the derived "master_secret" key.

It has a parameter, a **CK_SSL3_MASTER_KEY_DERIVE_PARAMS** structure, which allows for the passing of random data to the token as well as the returning of the protocol version number which is part of the pre-master key. This structure is defined in Section 6.24.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.

The template sent along with this mechanism during a **C_DeriveKey** call may indicate that the object class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN** attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

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

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

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

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

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK_MECHANISM_INFO** structure both indicate 48 bytes.

Note that the **CK_VERSION** structure pointed to by the **CK_SSL3_MASTER_KEY_DERIVE_PARAMS** structure's *pVersion* field will be modified by the **C_DeriveKey** call. In particular, when the call returns, this structure will hold the SSL version associated with the supplied pre_master key.

Note that this mechanism is only useable for cipher suites that use a 48-byte "pre_master" secret with an embedded version number. This includes the RSA cipher suites, but excludes the Diffie-Hellman cipher suites.

6.25.6 Master key derivation for Diffie-Hellman

Master key derivation for Diffie-Hellman in TLS 1.0, denoted **CKM_TLS_MASTER_KEY_DERIVE_DH**, is a mechanism used to derive one 48-

byte generic secret key from another arbitrary length generic secret key. It is used to produce the "master_secret" key used in the TLS protocol from the "pre_master" key.

It has a parameter, a **CK_SSL3_MASTER_KEY_DERIVE_PARAMS** structure, which allows for the passing of random data to the token. This structure is defined in Section 6.24. The *pVersion* field of the structure must be set to NULL_PTR since the version number is not embedded in the "pre_master" key as it is for RSA-like cipher suites.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.

The template sent along with this mechanism during a **C_DeriveKey** call may indicate that the object class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN** attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

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

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

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

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

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK_MECHANISM_INFO** structure both indicate 48 bytes.

Note that this mechanism is only useable for cipher suites that do not use a fixed length 48-byte “pre_master” secret with an embedded version number. This includes the Diffie-Hellman cipher suites, but excludes the RSA cipher suites.

6.25.7 Key and MAC derivation

Key, MAC and IV derivation in TLS 1.0, denoted **CKM_TLS_KEY_AND_MAC_DERIVE**, is a mechanism used to derive the appropriate cryptographic keying material used by a "CipherSuite" from the "master_secret" key and random data. This mechanism returns the key handles for the keys generated in the process, as well as the IVs created.

It has a parameter, a **CK_SSL3_KEY_MAT_PARAMS** structure, which allows for the passing of random data as well as the characteristic of the cryptographic material for the given CipherSuite and a pointer to a structure which receives the handles and IVs which were generated. This structure is defined in Section 6.24.

This mechanism contributes to the creation of four distinct keys on the token and returns two IVs (if IVs are requested by the caller) back to the caller. The keys are all given an object class of **CKO_SECRET_KEY**.

The two MACing keys ("client_write_MAC_secret" and "server_write_MAC_secret") are always given a type of **CKK_GENERIC_SECRET**. They are flagged as valid for signing, verification, and derivation operations.

The other two keys ("client_write_key" and "server_write_key") are typed according to information found in the template sent along with this mechanism during a **C_DeriveKey** function call. By default, they are flagged as valid for encryption, decryption, and derivation operations.

IVs will be generated and returned if the *ulIVSizeInBits* field of the **CK_SSL_KEY_MAT_PARAMS** field has a nonzero value. If they are generated, their length in bits will agree with the value in the *ulIVSizeInBits* field.

All four keys inherit the values of the **CKA_SENSITIVE**, **CKA_ALWAYS_SENSITIVE**, **CKA_EXTRACTABLE**, and **CKA_NEVER_EXTRACTABLE** attributes from the base key. The template provided to **C_DeriveKey** may not specify values for any of these attributes which differ from those held by the base key.

Note that the **CK_SSL3_KEY_MAT_OUT** structure pointed to by the **CK_SSL3_KEY_MAT_PARAMS** structure's *pReturnedKeyMaterial* field will be modified by the **C_DeriveKey** call. In particular, the four key handle fields in the **CK_SSL3_KEY_MAT_OUT** structure will be modified to hold handles to the newly-created keys; in addition, the buffers pointed to by the **CK_SSL3_KEY_MAT_OUT** structure's *pIVClient* and *pIVServer* fields will have IVs returned in them (if IVs are requested by the caller). Therefore, these two fields must point to buffers with sufficient space to hold any IVs that will be returned.

This mechanism departs from the other key derivation mechanisms in Cryptoki in its returned information. For most key-derivation mechanisms, **C_DeriveKey** returns a single key handle as a result of a successful completion. However, since the **CKM_SSL3_KEY_AND_MAC_DERIVE** mechanism returns all of its key handles in the **CK_SSL3_KEY_MAT_OUT** structure pointed to by the **CK_SSL3_KEY_MAT_PARAMS** structure specified as the mechanism parameter, the parameter *phKey* passed to **C_DeriveKey** is unnecessary, and should be a NULL_PTR.

If a call to **C_DeriveKey** with this mechanism fails, then *none* of the four keys will be created on the token.

6.26 WTLS

Details can be found in [WTLS].

When comparing the existing TLS mechanisms with these extensions to support WTLS one could argue that there would be no need to have distinct handling of the client and server side of the handshake. However, since in WTLS the server and client use different sequence numbers, there could be instances (e.g. when WTLS is used to protect asynchronous protocols) where sequence numbers on the client and server side differ, and hence this motivates the introduced split.

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_WTLS_PRE_MASTER_KEY_GEN					✓		
CKM_WTLS_MASTER_KEY_DERIVE							✓
CKM_WTLS_MASTER_KEY_DERIVE_DH_ECC							✓
CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE							✓
CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE							✓
CKM_WTLS_PRF							✓

6.26.1 Definitions

Mechanisms:

```
CKM_WTLS_PRE_MASTER_KEY_GEN
CKM_WTLS_MASTER_KEY_DERIVE
CKM_WTLS_MASTER_KEY_DERIVE_DH_ECC
CKM_WTLS_PRF
CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE
CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE
```

6.26.2 WTLS mechanism parameters

♦ **CK_WTLS_RANDOM_DATA; CK_WTLS_RANDOM_DATA_PTR**

CK_WTLS_RANDOM_DATA is a structure, which provides information about the random data of a client and a server in a WTLS context. This structure is used by the **CKM_WTLS_MASTER_KEY_DERIVE** mechanism. It is defined as follows:

```
typedef struct CK_WTLS_RANDOM_DATA {
  CK_BYTE_PTR pClientRandom;
  CK_ULONG    ulClientRandomLen;
  CK_BYTE_PTR pServerRandom;
  CK_ULONG    ulServerRandomLen;
} CK_WTLS_RANDOM_DATA;
```

The fields of the structure have the following meanings:

pClientRandom	pointer to the client's random data
ulClientRandomLen	length in bytes of the client's random data
pServerRandom	pointer to the server's random data
ulServerRandomLen	length in bytes of the server's random data

CK_WTLS_RANDOM_DATA_PTR is a pointer to a **CK_WTLS_RANDOM_DATA.**

♦ **CK_WTLS_MASTER_KEY_DERIVE_PARAMS; CK_WTLS_MASTER_KEY_DERIVE_PARAMS _PTR**

CK_WTLS_MASTER_KEY_DERIVE_PARAMS is a structure, which provides the parameters to the **CKM_WTLS_MASTER_KEY_DERIVE** mechanism. It is defined as follows:

```
typedef struct CK_WTLS_MASTER_KEY_DERIVE_PARAMS {
  CK_MECHANISM_TYPE   DigestMechanism;
  CK_WTLS_RANDOM_DATA RandomInfo;
  CK_BYTE_PTR         pVersion;
} CK_WTLS_MASTER_KEY_DERIVE_PARAMS;
```

The fields of the structure have the following meanings:

DigestMechanism	the mechanism type of the digest mechanism to be used (possible types can be found in [WTLS])
RandomInfo	Client's and server's random data

information

pVersion pointer to a **CK_BYTE** which receives the WTLS protocol version information

CK_WTLS_MASTER_KEY_DERIVE_PARAMS_PTR is a pointer to a **CK_WTLS_MASTER_KEY_DERIVE_PARAMS**.

- **CK_WTLS_PRF_PARAMS; CK_WTLS_PRF_PARAMS_PTR**

CK_WTLS_PRF_PARAMS is a structure, which provides the parameters to the **CKM_WTLS_PRF** mechanism. It is defined as follows:

```
typedef struct CK_WTLS_PRF_PARAMS {
  CK_MECHANISM_TYPE DigestMechanism;
  CK_BYTE_PTR       pSeed;
  CK_ULONG          ulSeedLen;
  CK_BYTE_PTR       pLabel;
  CK_ULONG          ulLabelLen;
  CK_BYTE_PTR       pOutput;
  CK_ULONG_PTR      pulOutputLen;
} CK_WTLS_PRF_PARAMS;
```

The fields of the structure have the following meanings:

DigestMechanism the mechanism type of the digest mechanism to be used (possible types can be found in [WTLS])

pSeed pointer to the input seed

ulSeedLen length in bytes of the input seed

pLabel pointer to the identifying label

ulLabelLen length in bytes of the identifying label

pOutput pointer receiving the output of the operation

pulOutputLen pointer to the length in bytes that the output to be created shall have, has to hold the desired length as input and will receive the calculated length as output

CK_WTLS_PRF_PARAMS_PTR is a pointer to a **CK_WTLS_PRF_PARAMS**.

♦ **CK_WTLS_KEY_MAT_OUT; CK_WTLS_KEY_MAT_OUT_PTR**

CK_WTLS_KEY_MAT_OUT is a structure that contains the resulting key handles and initialization vectors after performing a C_DeriveKey function with the **CKM_WTLS_SEVER_KEY_AND_MAC_DERIVE** or with the **CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE** mechanism. It is defined as follows:

```
typedef struct CK_WTLS_KEY_MAT_OUT {
  CK_OBJECT_HANDLE hMacSecret;
  CK_OBJECT_HANDLE hKey;
  CK_BYTE_PTR      pIV;
} CK_WTLS_KEY_MAT_OUT;
```

The fields of the structure have the following meanings:

hMacSecret	Key handle for the resulting MAC secret key
hKey	Key handle for the resulting secret key
pIV	Pointer to a location which receives the initialization vector (IV) created (if any)

CK_WTLS_KEY_MAT_OUT _PTR is a pointer to a **CK_WTLS_KEY_MAT_OUT**.

♦ **CK_WTLS_KEY_MAT_PARAMS; CK_WTLS_KEY_MAT_PARAMS_PTR**

CK_WTLS_KEY_MAT_PARAMS is a structure that provides the parameters to the **CKM_WTLS_SEVER_KEY_AND_MAC_DERIVE** and the **CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE** mechanisms. It is defined as follows:

```
typedef struct CK_WTLS_KEY_MAT_PARAMS {
  CK_MECHANISM_TYPE       DigestMechanism;
  CK_ULONG                ulMacSizeInBits;
  CK_ULONG                ulKeySizeInBits;
  CK_ULONG                ulIVSizeInBits;
  CK_ULONG                ulSequenceNumber;
  CK_BBOOL                bIsExport;
  CK_WTLS_RANDOM_DATA     RandomInfo;
  CK_WTLS_KEY_MAT_OUT_PTR pReturnedKeyMaterial;
} CK_WTLS_KEY_MAT_PARAMS;
```

The fields of the structure have the following meanings:

DigestMechanism	the mechanism type of the digest mechanism to be used (possible types can be found in [WTLS])

ulMacSizeInBits	the length (in bits) of the MACing key agreed upon during the protocol handshake phase
ulKeySizeInBits	the length (in bits) of the secret key agreed upon during the handshake phase
ulIVSizeInBits	the length (in bits) of the IV agreed upon during the handshake phase. If no IV is required, the length should be set to 0.
ulSequenceNumber	The current sequence number used for records sent by the client and server respectively
bIsExport	a boolean value which indicates whether the keys have to be derived for an export version of the protocol. If this value is true (i.e. the keys are exportable) then *ulKeySizeInBits* is the length of the key in bits before expansion. The length of the key after expansion is determined by the information found in the template sent along with this mechanism during a C_DeriveKey function call (either the **CKA_KEY_TYPE** or the **CKA_VALUE_LEN** attribute).
RandomInfo	client's and server's random data information
pReturnedKeyMaterial	points to a **CK_WTLS_KEY_MAT_OUT** structure which receives the handles for the keys generated and the IV

CK_WTLS_KEY_MAT_PARAMS_PTR is a pointer to a **CK_WTLS_KEY_MAT_PARAMS**.

6.26.3 Pre master secret key generation for RSA key exchange suite

Pre master secret key generation for the RSA key exchange suite in WTLS denoted **CKM_WTLS_PRE_MASTER_KEY_GEN**, is a mechanism, which generates a variable length secret key. It is used to produce the pre master secret key for RSA key exchange suite used in WTLS. This mechanism returns a handle to the pre master secret key.

It has one parameter, a **CK_BYTE**, which provides the client's WTLS version.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE** and **CKA_VALUE** attributes to the new key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.

The template sent along with this mechanism during a **C_GenerateKey** call may indicate that the object class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN** attribute indicates the length of the pre master secret key.

For this mechanism, the ulMinKeySize field of the **CK_MECHANISM_INFO** structure shall indicate 20 bytes.

6.26.4 Master secret key derivation

Master secret derivation in WTLS, denoted **CKM_WTLS_MASTER_KEY_DERIVE**, is a mechanism used to derive a 20 byte generic secret key from variable length secret key. It is used to produce the master secret key used in WTLS from the pre master secret key. This mechanism returns the value of the client version, which is built into the pre master secret key as well as a handle to the derived master secret key.

It has a parameter, a **CK_WTLS_MASTER_KEY_DERIVE_PARAMS** structure, which allows for passing the mechanism type of the digest mechanism to be used as well as the passing of random data to the token as well as the returning of the protocol version number which is part of the pre master secret key.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.

The template sent along with this mechanism during a **C_DeriveKey** call may indicate that the object class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN** attribute has value 20. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

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

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

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

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

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK_MECHANISM_INFO** structure both indicate 20 bytes.

Note that the **CK_BYTE** pointed to by the **CK_WTLS_MASTER_KEY_DERIVE_PARAMS** structure's *pVersion* field will be modified by the **C_DeriveKey** call. In particular, when the call returns, this byte will hold the WTLS version associated with the supplied pre master secret key.

Note that this mechanism is only useable for key exchange suites that use a 20-byte pre master secret key with an embedded version number. This includes the RSA key exchange suites, but excludes the Diffie-Hellman and Elliptic Curve Cryptography key exchange suites.

6.26.5 Master secret key derivation for Diffie-Hellman and Elliptic Curve Cryptography

Master secret derivation for Diffie-Hellman and Elliptic Curve Cryptography in WTLS, denoted **CKM_WTLS_MASTER_KEY_DERIVE_DH_ECC**, is a mechanism used to derive a 20 byte generic secret key from variable length secret key. It is used to produce the master secret key used in WTLS from the pre master secret key. This mechanism returns a handle to the derived master secret key.

It has a parameter, a **CK_WTLS_MASTER_KEY_DERIVE_PARAMS** structure, which allows for the passing of the mechanism type of the digest mechanism to be used as well as random data to the token. The *pVersion* field of the structure must be set to NULL_PTR since the version number is not embedded in the pre master secret key as it is for RSA-like key exchange suites.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.

The template sent along with this mechanism during a **C_DeriveKey** call may indicate that the object class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN** attribute has value 20. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

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

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

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

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

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK_MECHANISM_INFO** structure both indicate 20 bytes.

Note that this mechanism is only useable for key exchange suites that do not use a fixed length 20-byte pre master secret key with an embedded version number. This includes the Diffie-Hellman and Elliptic Curve Cryptography key exchange suites, but excludes the RSA key exchange suites.

6.26.6 WTLS PRF (pseudorandom function)

PRF (pseudo random function) in WTLS, denoted **CKM_WTLS_PRF**, is a mechanism used to produce a securely generated pseudo-random output of arbitrary length. The keys it uses are generic secret keys.

It has a parameter, a **CK_WTLS_PRF_PARAMS** structure, which allows for passing the mechanism type of the digest mechanism to be used, the passing of the input seed and its length, the passing of an identifying label and its length and the passing of the length of the output to the token and for receiving the output.

This mechanism produces securely generated pseudo-random output of the length specified in the parameter.

This mechanism departs from the other key derivation mechanisms in Cryptoki in not using the template sent along with this mechanism during a **C_DeriveKey** function call, which means the template shall be a NULL_PTR. For most key-derivation mechanisms, **C_DeriveKey** returns a single key handle as a result of a successful completion. However, since the **CKM_WTLS_PRF** mechanism returns the requested number of output bytes in the **CK_WTLS_PRF_PARAMS** structure specified as the mechanism parameter, the parameter *phKey* passed to **C_DeriveKey** is unnecessary, and should be a NULL_PTR.

If a call to **C_DeriveKey** with this mechanism fails, then no output will be generated.

6.26.7 Server Key and MAC derivation

Server key, MAC and IV derivation in WTLS, denoted **CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE**, is a mechanism used to derive the appropriate cryptographic keying material used by a cipher suite from the master secret key and random data. This mechanism returns the key handles for the keys generated in the process, as well as the IV created.

It has a parameter, a **CK_WTLS_KEY_MAT_PARAMS** structure, which allows for the passing of the mechanism type of the digest mechanism to be used, random data, the characteristic of the cryptographic material for the given cipher suite, and a pointer to a structure which receives the handles and IV which were generated.

This mechanism contributes to the creation of two distinct keys and returns one IV (if an IV is requested by the caller) back to the caller. The keys are all given an object class of **CKO_SECRET_KEY**.

The MACing key (server write MAC secret) is always given a type of **CKK_GENERIC_SECRET**. It is flagged as valid for signing, verification and derivation operations.

The other key (server write key) is typed according to information found in the template sent along with this mechanism during a **C_DeriveKey** function call. By default, it is flagged as valid for encryption, decryption, and derivation operations.

An IV (server write IV) will be generated and returned if the *ulIVSizeInBits* field of the **CK_WTLS_KEY_MAT_PARAMS** field has a nonzero value. If it is generated, its length in bits will agree with the value in the *ulIVSizeInBits* field

Both keys inherit the values of the **CKA_SENSITIVE**, **CKA_ALWAYS_SENSITIVE**, **CKA_EXTRACTABLE**, and **CKA_NEVER_EXTRACTABLE** attributes from the base key. The template provided to **C_DeriveKey** may not specify values for any of these attributes that differ from those held by the base key.

Note that the **CK_WTLS_KEY_MAT_OUT** structure pointed to by the **CK_WTLS_KEY_MAT_PARAMS** structure's *pReturnedKeyMaterial* field will be modified by the **C_DeriveKey** call. In particular, the two key handle fields in the **CK_WTLS_KEY_MAT_OUT** structure will be modified to hold handles to the newly-created keys; in addition, the buffer pointed to by the **CK_WTLS_KEY_MAT_OUT** structure's *pIV* field will have the IV returned in them (if an IV is requested by the caller). Therefore, this field must point to a buffer with sufficient space to hold any IV that will be returned.

This mechanism departs from the other key derivation mechanisms in Cryptoki in its returned information. For most key-derivation mechanisms, **C_DeriveKey** returns a single key handle as a result of a successful completion. However, since the **CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE** mechanism returns all of its key handles in the **CK_WTLS_KEY_MAT_OUT** structure pointed to by the **CK_WTLS_KEY_MAT_PARAMS** structure specified as the mechanism parameter, the parameter *phKey* passed to **C_DeriveKey** is unnecessary, and should be a NULL_PTR.

If a call to **C_DeriveKey** with this mechanism fails, then *none* of the two keys will be created.

6.26.8 Client key and MAC derivation

Client key, MAC and IV derivation in WTLS, denoted **CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE**, is a mechanism used to derive the appropriate cryptographic keying material used by a cipher suite from the master secret key and random data. This mechanism returns the key handles for the keys generated in the process, as well as the IV created.

It has a parameter, a **CK_WTLS_KEY_MAT_PARAMS** structure, which allows for the passing of the mechanism type of the digest mechanism to be used, random data, the characteristic of the cryptographic material for the given cipher suite, and a pointer to a structure which receives the handles and IV which were generated.

This mechanism contributes to the creation of two distinct keys and returns one IV (if an IV is requested by the caller) back to the caller. The keys are all given an object class of **CKO_SECRET_KEY**.

The MACing key (client write MAC secret) is always given a type of **CKK_GENERIC_SECRET**. It is flagged as valid for signing, verification and derivation operations.

The other key (client write key) is typed according to information found in the template sent along with this mechanism during a **C_DeriveKey** function call. By default, it is flagged as valid for encryption, decryption, and derivation operations.

An IV (client write IV) will be generated and returned if the *ulIVSizeInBits* field of the **CK_WTLS_KEY_MAT_PARAMS** field has a nonzero value. If it is generated, its length in bits will agree with the value in the *ulIVSizeInBits* field

Both keys inherit the values of the **CKA_SENSITIVE**, **CKA_ALWAYS_SENSITIVE**, **CKA_EXTRACTABLE**, and **CKA_NEVER_EXTRACTABLE** attributes from the base key. The template provided to **C_DeriveKey** may not specify values for any of these attributes that differ from those held by the base key.

Note that the **CK_WTLS_KEY_MAT_OUT** structure pointed to by the **CK_WTLS_KEY_MAT_PARAMS** structure's *pReturnedKeyMaterial* field will be modified by the **C_DeriveKey** call. In particular, the two key handle fields in the **CK_WTLS_KEY_MAT_OUT** structure will be modified to hold handles to the newly-created keys; in addition, the buffer pointed to by the **CK_WTLS_KEY_MAT_OUT** structure's *pIV* field will have the IV returned in them (if an IV is requested by the caller). Therefore, this field must point to a buffer with sufficient space to hold any IV that will be returned.

This mechanism departs from the other key derivation mechanisms in Cryptoki in its returned information. For most key-derivation mechanisms, **C_DeriveKey** returns a single key handle as a result of a successful completion. However, since the **CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE** mechanism returns all of its key handles in the **CK_WTLS_KEY_MAT_OUT** structure pointed to by the **CK_WTLS_KEY_MAT_PARAMS** structure specified as the mechanism parameter, the parameter *phKey* passed to **C_DeriveKey** is unnecessary, and should be a NULL_PTR.

If a call to **C_DeriveKey** with this mechanism fails, then *none* of the two keys will be created.

6.27 Miscellaneous simple key derivation mechanisms

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_CONCATENATE_BASE_AND_KEY							✓
CKM_CONCATENATE_BASE_AND_DATA							✓
CKM_CONCATENATE_DATA_AND_BASE							✓
CKM_XOR_BASE_AND_DATA							✓
CKM_EXTRACT_KEY_FROM_KEY							✓

6.27.1 Definitions

Mechanisms:

```
CKM_CONCATENATE_BASE_AND_DATA
CKM_CONCATENATE_DATA_AND_BASE
CKM_XOR_BASE_AND_DATA
CKM_EXTRACT_KEY_FROM_KEY
CKM_CONCATENATE_BASE_AND_KEY
```

6.27.2 Parameters for miscellaneous simple key derivation mechanisms

♦ **CK_KEY_DERIVATION_STRING_DATA; CK_KEY_DERIVATION_STRING_DATA_PTR**

CK_KEY_DERIVATION_STRING_DATA provides the parameters for the **CKM_CONCATENATE_BASE_AND_DATA**, **CKM_CONCATENATE_DATA_AND_BASE**, and **CKM_XOR_BASE_AND_DATA** mechanisms. It is defined as follows:

```
typedef struct CK_KEY_DERIVATION_STRING_DATA {
  CK_BYTE_PTR pData;
  CK_ULONG ulLen;
} CK_KEY_DERIVATION_STRING_DATA;
```

The fields of the structure have the following meanings:

pData pointer to the byte string

ulLen length of the byte string

CK_KEY_DERIVATION_STRING_DATA_PTR is a pointer to a **CK_KEY_DERIVATION_STRING_DATA**.

♦ CK_EXTRACT_PARAMS; CK_EXTRACT_PARAMS_PTR

CK_KEY_EXTRACT_PARAMS provides the parameter to the **CKM_EXTRACT_KEY_FROM_KEY** mechanism. It specifies which bit of the base key should be used as the first bit of the derived key. It is defined as follows:

```
typedef CK_ULONG CK_EXTRACT_PARAMS;
```

CK_EXTRACT_PARAMS_PTR is a pointer to a **CK_EXTRACT_PARAMS**.

6.27.3 Concatenation of a base key and another key

This mechanism, denoted **CKM_CONCATENATE_BASE_AND_KEY**, derives a secret key from the concatenation of two existing secret keys. The two keys are specified by handles; the values of the keys specified are concatenated together in a buffer.

This mechanism takes a parameter, a **CK_OBJECT_HANDLE**. This handle produces the key value information which is appended to the end of the base key's value information (the base key is the key whose handle is supplied as an argument to **C_DeriveKey**).

For example, if the value of the base key is `0x01234567,` and the value of the other key is `0x89ABCDEF`, then the value of the derived key will be taken from a buffer containing the string `0x0123456789ABCDEF`.

- If no length or key type is provided in the template, then the key produced by this mechanism will be a generic secret key. Its length will be equal to the sum of the lengths of the values of the two original keys.

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

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

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

If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more bytes than are available by concatenating the two original keys' values, an error is generated.

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

- If either of the two original keys has its **CKA_SENSITIVE** attribute set to CK_TRUE, so does the derived key. If not, then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied template or from a default value.
- Similarly, if either of the two original keys has its **CKA_EXTRACTABLE** attribute set to CK_FALSE, so does the derived key. If not, then the derived key's **CKA_EXTRACTABLE** attribute is set either from the supplied template or from a default value.
- The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to CK_TRUE if and only if both of the original keys have their **CKA_ALWAYS_SENSITIVE** attributes set to CK_TRUE.
- Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to CK_TRUE if and only if both of the original keys have their **CKA_NEVER_EXTRACTABLE** attributes set to CK_TRUE.

6.27.4 Concatenation of a base key and data

This mechanism, denoted **CKM_CONCATENATE_BASE_AND_DATA**, derives a secret key by concatenating data onto the end of a specified secret key.

This mechanism takes a parameter, a **CK_KEY_DERIVATION_STRING_DATA** structure, which specifies the length and value of the data which will be appended to the base key to derive another key.

For example, if the value of the base key is `0x01234567,` and the value of the data is `0x89ABCDEF`, then the value of the derived key will be taken from a buffer containing the string `0x0123456789ABCDEF`.

- If no length or key type is provided in the template, then the key produced by this mechanism will be a generic secret key. Its length will be equal to the sum of the lengths of the value of the original key and the data.
- If no key type is provided in the template, but a length is, then the key produced by this mechanism will be a generic secret key of the specified length.
- If no length is provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn't, an error will be returned.

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

If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more bytes than are available by concatenating the original key's value and the data, an error is generated.

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

- If the base key has its **CKA_SENSITIVE** attribute set to CK_TRUE, so does the derived key. If not, then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied template or from a default value.
- Similarly, if the base key has its **CKA_EXTRACTABLE** attribute set to CK_FALSE, so does the derived key. If not, then the derived key's **CKA_EXTRACTABLE** attribute is set either from the supplied template or from a default value.
- The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to CK_TRUE if and only if the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE.
- Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to CK_TRUE if and only if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_TRUE.

6.27.5 Concatenation of data and a base key

This mechanism, denoted **CKM_CONCATENATE_DATA_AND_BASE**, derives a secret key by prepending data to the start of a specified secret key.

This mechanism takes a parameter, a **CK_KEY_DERIVATION_STRING_DATA** structure, which specifies the length and value of the data which will be prepended to the base key to derive another key.

For example, if the value of the base key is `0x01234567,` and the value of the data is `0x89ABCDEF`, then the value of the derived key will be taken from a buffer containing the string `0x89ABCDEF01234567`.

- If no length or key type is provided in the template, then the key produced by this mechanism will be a generic secret key. Its length will be equal to the sum of the lengths of the data and the value of the original key.

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

If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more bytes than are available by concatenating the data and the original key's value, an error is generated.

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

- If the base key has its **CKA_SENSITIVE** attribute set to CK_TRUE, so does the derived key. If not, then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied template or from a default value.
- Similarly, if the base key has its **CKA_EXTRACTABLE** attribute set to CK_FALSE, so does the derived key. If not, then the derived key's **CKA_EXTRACTABLE** attribute is set either from the supplied template or from a default value.
- The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to CK_TRUE if and only if the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE.
- Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to CK_TRUE if and only if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_TRUE.

6.27.6 XORing of a key and data

XORing key derivation, denoted **CKM_XOR_BASE_AND_DATA**, is a mechanism which provides the capability of deriving a secret key by performing a bit XORing of a key pointed to by a base key handle and some data.

This mechanism takes a parameter, a **CK_KEY_DERIVATION_STRING_DATA** structure, which specifies the data with which to XOR the original key's value.

For example, if the value of the base key is `0x01234567,` and the value of the data is `0x89ABCDEF`, then the value of the derived key will be taken from a buffer containing the string `0x88888888`.

- If no length or key type is provided in the template, then the key produced by this mechanism will be a generic secret key. Its length will be equal to the minimum of the lengths of the data and the value of the original key.
- If no key type is provided in the template, but a length is, then the key produced by this mechanism will be a generic secret key of the specified length.
- If no length is provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn't, an error will be returned.
- If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism will be of the specified type and length.

If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more bytes than are available by taking the shorter of the data and the original key's value, an error is generated.

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

- If the base key has its **CKA_SENSITIVE** attribute set to CK_TRUE, so does the derived key. If not, then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied template or from a default value.
- Similarly, if the base key has its **CKA_EXTRACTABLE** attribute set to CK_FALSE, so does the derived key. If not, then the derived key's **CKA_EXTRACTABLE** attribute is set either from the supplied template or from a default value.
- The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to CK_TRUE if and only if the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE.
- Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to CK_TRUE if and only if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_TRUE.

6.27.7 Extraction of one key from another key

Extraction of one key from another key, denoted **CKM_EXTRACT_KEY_FROM_KEY**, is a mechanism which provides the capability of creating one secret key from the bits of another secret key.

This mechanism has a parameter, a CK_EXTRACT_PARAMS, which specifies which bit of the original key should be used as the first bit of the newly-derived key.

We give an example of how this mechanism works. Suppose a token has a secret key with the 4-byte value 0x329F84A9. We will derive a 2-byte secret key from this key, starting at bit position 21 (i.e., the value of the parameter to the CKM_EXTRACT_KEY_FROM_KEY mechanism is 21).

1. We write the key's value in binary: 0011 0010 1001 1111 1000 0100 1010 1001. We regard this binary string as holding the 32 bits of the key, labeled as b0, b1, …, b31.

2. We then extract 16 consecutive bits (i.e., 2 bytes) from this binary string, starting at bit b21. We obtain the binary string 1001 0101 0010 0110.

3. The value of the new key is thus 0x9526.

Note that when constructing the value of the derived key, it is permissible to wrap around the end of the binary string representing the original key's value.

If the original key used in this process is sensitive, then the derived key must also be sensitive for the derivation to succeed.

- If no length or key type is provided in the template, then an error will be returned.

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

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

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

If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more bytes than the original key has, an error is generated.

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

- If the base key has its **CKA_SENSITIVE** attribute set to CK_TRUE, so does the derived key. If not, then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied template or from a default value.
- Similarly, if the base key has its **CKA_EXTRACTABLE** attribute set to CK_FALSE, so does the derived key. If not, then the derived key's **CKA_EXTRACTABLE** attribute is set either from the supplied template or from a default value.
- The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to CK_TRUE if and only if the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE.
- Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to CK_TRUE if and only if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_TRUE.

6.28 CMS

	Functions						
Mechanism	**Encrypt & Decrypt**	**Sign & Verify**	**SR & VR**[1]	**Digest**	**Gen. Key/ Key Pair**	**Wrap & Unwrap**	**Derive**
CKM_CMS_SIG		✓	✓				

6.28.1 Definitions

Mechanisms:

```
CKM_CMS_SIG
```

6.28.2 CMS Signature Mechanism Objects

These objects provide information relating to the CKM_CMS_SIG mechanism. CKM_CMS_SIG mechanism object attributes represent information about supported CMS signature attributes in the token. They are only present on tokens supporting the **CKM_CMS_SIG** mechanism, but must be present on those tokens.

Table 67, CMS Signature Mechanism Object Attributes

Attribute	Data type	Meaning
CKA_REQUIRED_CMS_ATTRIBUTES	Byte array	Attributes the token always will include in the set of CMS signed attributes
CKA_DEFAULT_CMS_ATTRIBUTES	Byte array	Attributes the token will include in the set of CMS signed attributes in the absence of any attributes specified by the application
CKA_SUPPORTED_CMS_ATTRIBUTES	Byte array	Attributes the token may include in the set of CMS signed attributes upon request by the application

The contents of each byte array will be a DER-encoded list of CMS **Attributes** with optional accompanying values. Any attributes in the list shall be identified with its object identifier, and any values shall be DER-encoded. The list of attributes is defined in ASN.1 as:

```
Attributes ::= SET SIZE (1..MAX) OF Attribute

Attribute ::= SEQUENCE {
   attrType   OBJECT IDENTIFIER,
   attrValues SET OF ANY DEFINED BY OBJECT IDENTIFIER OPTIONAL
}
```

The client may not set any of the attributes.

6.28.3 CMS mechanism parameters

- **CK_CMS_SIG_PARAMS, CK_CMS_SIG_PARAMS_PTR**

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

```
typedef struct CK_CMS_SIG_PARAMS {
CK_OBJECT_HANDLE      certificateHandle;
CK_MECHANISM_PTR      pSigningMechanism;
CK_MECHANISM_PTR      pDigestMechanism;
CK_UTF8CHAR_PTR       pContentType;
CK_BYTE_PTR           pRequestedAttributes;
CK_ULONG              ulRequestedAttributesLen;
CK_BYTE_PTR           pRequiredAttributes;
CK_ULONG              ulRequiredAttributesLen;
} CK_CMS_SIG_PARAMS;
```

The fields of the structure have the following meanings:

certificateHandle	Object handle for a certificate associated with the signing key. The token may use information from this certificate to identify the signer in the **SignerInfo** result value. *CertificateHandle* may be NULL_PTR if the certificate is not available as a PKCS #11 object or if the calling application leaves the choice of certificate completely to the token.
pSigningMechanism	Mechanism to use when signing a constructed CMS **SignedAttributes** value. E.g. **CKM_SHA1_RSA_PKCS**.
pDigestMechanism	Mechanism to use when digesting the data. Value shall be NULL_PTR when the digest mechanism to use follows from the *pSigningMechanism* parameter.
pContentType	NULL-terminated string indicating complete MIME Content-type of message to be signed; or the value NULL_PTR if the message is a MIME object (which the token can parse to determine its MIME Content-type if required). Use the value "`application/octet-stream`" if the MIME type for the message is unknown or undefined. Note that the *pContentType* string shall conform to the syntax specified in RFC 2045, i.e. any parameters needed for correct presentation of the content by the token (such as, for example, a non-default "`charset`") must be present. The token must follow rules and procedures defined in RFC 2045 when presenting the content.
pRequestedAttributes	Pointer to DER-encoded list of CMS **Attributes** the caller requests to be included in the signed attributes. Token may freely ignore this list or modify any supplied values.
ulRequestedAttributesLen	Length in bytes of the value pointed to by *pRequestedAttributes*
pRequiredAttributes	Pointer to DER-encoded list of CMS **Attributes** (with accompanying values) required to be included in the resulting signed attributes. Token must not modify any supplied values. If the token does not support one or more of the attributes, or does not accept provided values, the signature operation will fail. The token will use its own default attributes when signing if both the *pRequestedAttributes* and *pRequiredAttributes* field are set to NULL_PTR.

ulRequiredAttributesLen	Length in bytes, of the value pointed to by *pRequiredAttributes*.

6.28.4 CMS signatures

The CMS mechanism, denoted **CKM_CMS_SIG**, is a multi-purpose mechanism based on the structures defined in PKCS #7 and RFC 2630. It supports single- or multiple-part signatures with and without message recovery. The mechanism is intended for use with, e.g., PTDs (see MeT-PTD) or other capable tokens. The token will construct a CMS **SignedAttributes** value and compute a signature on this value. The content of the **SignedAttributes** value is decided by the token, however the caller can suggest some attributes in the parameter *pRequestedAttributes*. The caller can also require some attributes to be present through the parameters *pRequiredAttributes*. The signature is computed in accordance with the parameter *pSigningMechanism*.

When this mechanism is used in successful calls to **C_Sign** or **C_SignFinal**, the *pSignature* return value will point to a DER-encoded value of type **SignerInfo**. **SignerInfo** is defined in ASN.1 as follows (for a complete definition of all fields and types, see RFC 2630):

```
SignerInfo ::= SEQUENCE {
    version CMSVersion,
    sid SignerIdentifier,
    digestAlgorithm DigestAlgorithmIdentifier,
    signedAttrs [0] IMPLICIT SignedAttributes OPTIONAL,
    signatureAlgorithm SignatureAlgorithmIdentifier,
    signature SignatureValue,
    unsignedAttrs [1] IMPLICIT UnsignedAttributes OPTIONAL }
```

The *certificateHandle* parameter, when set, helps the token populate the **sid** field of the **SignerInfo** value. If *certificateHandle* is NULL_PTR the choice of a suitable certificate reference in the **SignerInfo** result value is left to the token (the token could, e.g., interact with the user).

This mechanism shall not be used in calls to **C_Verify** or **C_VerifyFinal** (use the *pSigningMechanism* mechanism instead).

In order for an application to find out what attributes are supported by a token, what attributes that will be added by default, and what attributes that always will be added, it shall analyze the contents of the **CKH_CMS_ATTRIBUTES** hardware feature object.

For the *pRequiredAttributes* field, the token may have to interact with the user to find out whether to accept a proposed value or not. The token should never accept any proposed attribute values without some kind of confirmation from its owner (but this could be through, e.g., configuration or policy settings and not direct interaction). If a user rejects proposed values, or the signature request as such, the value CKR_FUNCTION_REJECTED shall be returned.

When possible, applications should use the **CKM_CMS_SIG** mechanism when generating CMS-compatible signatures rather than lower-level mechanisms such as **CKM_SHA1_RSA_PKCS**. This is especially true when the signatures are to be made on content that the token is able to present to a user. Exceptions may include those cases where the token does not support a particular signing attribute. Note however that the token may refuse usage of a particular signature key unless the content to be signed is known (i.e. the **CKM_CMS_SIG** mechanism is used).

When a token does not have presentation capabilities, the PKCS #11-aware application may avoid sending the whole message to the token by electing to use a suitable signature mechanism (e.g. **CKM_RSA_PKCS**) as the *pSigningMechanism* value in the **CKM_CMS_SIG_PARAMS** structure, and digesting the message itself before passing it to the token.

PKCS #11-aware applications making use of tokens with presentation capabilities, should attempt to provide messages to be signed by the token in a format possible for the token to present to the user. Tokens that receive multipart MIME-messages for which only certain parts are possible to present may fail the signature operation with a return value of **CKR_DATA_INVALID**, but may also choose to add a signing attribute indicating which parts of the message that were possible to present.

6.29 Blowfish

Blowfish, a secret-key block cipher. It is a Feistel network, iterating a simple encryption function 16 times. The block size is 64 bits, and the key can be any length up to 448 bits. Although there is a complex initialization phase required before any encryption can take place, the actual encryption of data is very efficient on large microprocessors. Ref. http://www.counterpane.com/bfsverlag.html

	Functions						
Mechanism	**Encrypt & Decrypt**	**Sign & Verify**	**SR & VR**[1]	**Digest**	**Gen. Key/ Key Pair**	**Wrap & Unwrap**	**Derive**
CKM_BLOWFISH_CBC	✓					✓	
CKM_BLOWFISH_CBC_PAD	✓					✓	

6.29.1 Definitions

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

Mechanisms:

```
CKM_BLOWFISH_KEY_GEN
CKM_BLOWFISH_CBC
```



```
CKM_BLOWFISH_CBC_PAD
```

6.29.2 BLOWFISH secret key objects

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

Table 68, BLOWFISH Secret Key Object

Attribute	Data type	Meaning
CKA_VALUE[1,4,6,7]	Byte array	Key value the key can be any length up to 448 bits. Bit length restricted to an byte array.
CKA_VALUE_LEN[2,3]	CK_ULONG	Length in bytes of key value

- Refer to [PKCS #11-B] table 15 for footnotes

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

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_BLOWFISH;
CK_UTF8CHAR label[] = “A blowfish secret key object”;
CK_BYTE value[16] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
  {CKA_CLASS, &class, sizeof(class)},
  {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
  {CKA_TOKEN, &true, sizeof(true)},
  {CKA_LABEL, label, sizeof(label)-1},
  {CKA_ENCRYPT, &true, sizeof(true)},
  {CKA_VALUE, value, sizeof(value)}
};
```

6.29.3 Blowfish key generation

The Blowfish key generation mechanism, denoted **CKM_BLOWFISH_KEY_GEN**, is a key generation mechanism Blowfish.

It does not have a parameter.

The mechanism generates Blowfish keys with a particular length, as specified in the **CKA_VALUE_LEN** attribute of the template for the key.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

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

6.29.4 Blowfish -CBC

Blowfish-CBC, denoted **CKM_BLOWFISH_CBC**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping.

It has a parameter, a 8-byte initialization vector.

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

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

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

Table 2, BLOWFISH-CBC: Key And Data Length

Function	Key type	Input lenght	Output lenght
C_Encrypt	BLOWFISH	multiple of block size	same as input length
C_Decrypt	BLOWFISH	multiple of block size	same as input length
C_WrapKey	BLOWFISH	any	input length rounded up to multiple of the block size
C_UnwrapKey	BLOWFISH	multiple of block size	determined by type of key being unwrapped or

			CKA_VALUE_LEN

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

6.29.5 Blowfish -CBC with PKCS padding

Blowfish-CBC-PAD, denoted CKM_BLOWFISH_CBC_PAD, is a mechanism for single- and multiple-part encryption and decryption, key wrapping and key unwrapping, cipher-block chaining mode and the block cipher padding method detailed in PKCS #7.

It has a parameter, a 8-byte initialization vector.

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

The entries in the table below for data length constraints when wrapping and unwrapping keys do not apply to wrapping and unwrapping private keys.

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

Table 3, BLOWFISH-CBC with PKCS Padding: Key And Data Length

Function	**Key type**	**Input lenght**	**Output lenght**
C_Encrypt	BLOWFISH	any	input length rounded up to multiple of the block size
C_Decrypt	BLOWFISH	multiple of block size	between 1 and block length block size bytes shorter than input length
C_WrapKey	BLOWFISH	any	input length rounded up to multiple of the block size

C_UnwrapKey	BLOWFISH	multiple of block size	between 1 and block length block size bytes shorter than input length

6.30 Twofish

Ref. http://www.counterpane.com/twofish-brief.html

6.30.1 Definitions

This section defines the key type "CKK_TWOFISH" for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

```
CKM_TWOFISH_KEY_GEN
CKM_TWOFISH_CBC
CKM_TWOFISH_CBC_PAD
```

6.30.2 Twofish secret key objects

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

Table 69, Twofish Secret Key Object

Attribute	Data type	Meaning
CKA_VALUE[1,4,6,7]	Byte array	Key value 128-, 192-, or 256-bit key
CKA_VALUE_LEN[2,3]	CK_ULONG	Length in bytes of key value

- Refer to [PKCS #11-B] table 15 for footnotes

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

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_TWOFISH;
CK_UTF8CHAR label[] = "A twofish secret key object";
CK_BYTE value[16] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
```



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

6.30.3 Twofish key generation

The Twofish key generation mechanism, denoted **CKM_TWOFISH_KEY_GEN**, is a key generation mechanism Twofish.

It does not have a parameter.

The mechanism generates Blowfish keys with a particular length, as specified in the **CKA_VALUE_LEN** attribute of the template for the key.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

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

6.30.4 Twofish -CBC

Twofish-CBC, denoted **CKM_TWOFISH_CBC**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping.

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

6.30.5 Towfish -CBC with PKCS padding

Towfish-CBC-PAD, denoted CKM_TOWFISH_CBC_PAD, is a mechanism for single- and multiple-part encryption and decryption, key wrapping and key unwrapping, cipher-block chaining mode and the block cipher padding method detailed in PKCS #7.

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

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

6.31 CAMELLIA

Camellia is a block cipher with 128-bit block size and 128-, 192-, and 256-bit keys, similar to AES. Camellia is described e.g. in IETF RFC 3713.

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_CAMELLIA_KEY_GEN					✓		
CKM_CAMELLIA_ECB	✓					✓	
CKM_CAMELLIA_CBC	✓					✓	
CKM_CAMELLIA_CBC_PAD	✓					✓	
CKM_CAMELLIA_MAC_GENERAL		✓					
CKM_CAMELLIA_MAC		✓					
CKM_CAMELLIA_ECB_ENCRYPT_DATA							✓
CKM_CAMELLIA_CBC_ENCRYPT_DATA							✓

6.31.1 Definitions

This section defines the key type "CKK_CAMELLIA" for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

```
CKM_CAMELLIA_KEY_GEN
CKM_CAMELLIA_ECB
CKM_CAMELLIA_CBC
CKM_CAMELLIA_MAC
CKM_CAMELLIA_MAC_GENERAL
CKM_CAMELLIA_CBC_PAD
```

6.31.2 Camellia secret key objects

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

Table 70, Camellia Secret Key Object Attributes

Attribute	Data type	Meaning
CKA_VALUE[1,4,6,7]	Byte array	Key value (16, 24, or 32 bytes)
CKA_VALUE_LEN[2,3,6]	CK_ULONG	Length in bytes of key value

- Refer to [PKCS #11-B] table 15 for footnotes.

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

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_CAMELLIA;
CK_UTF8CHAR label[] = “A Camellia secret key object”;
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
  {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)}
};
```

6.31.3 Camellia key generation

The Camellia key generation mechanism, denoted CKM_CAMELLIA_KEY_GEN, is a key generation mechanism for Camellia.

It does not have a parameter.

The mechanism generates Camellia keys with a particular length in bytes, as specified in the **CKA_VALUE_LEN** attribute of the template for the key.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the Camellia key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

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

6.31.4 Camellia-ECB

Camellia-ECB, denoted **CKM_CAMELLIA_ECB**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on Camellia and electronic codebook mode.

It does not have a parameter.

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

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

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

Table 71, Camellia-ECB: Key And Data Length

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

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

6.31.5 Camellia-CBC

Camellia-CBC, denoted **CKM_CAMELLIA_CBC**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on Camellia and cipher-block chaining mode.

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

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

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

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

Table 72, Camellia-CBC: Key And Data Length

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

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

6.31.6 Camellia-CBC with PKCS padding

Camellia-CBC with PKCS padding, denoted **CKM_CAMELLIA_CBC_PAD**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on Camellia; cipher-block chaining mode; and the block cipher padding method detailed in PKCS #7.

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

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

In addition to being able to wrap and unwrap secret keys, this mechanism can wrap and unwrap RSA, Diffie-Hellman, X9.42 Diffie-Hellman, EC (also related to ECDSA) and DSA private keys (see Section TBA for details). The entries in the table below for data length constraints when wrapping and unwrapping keys do not apply to wrapping and unwrapping private keys.

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

Table 73, Camellia-CBC with PKCS Padding: Key And Data Length

Function	Key type	Input length	Output length
C_Encrypt	CKK_CAMELLIA	any	input length rounded up to multiple of the block size
C_Decrypt	CKK_CAMELLIA	multiple of block size	between 1 and block size bytes shorter than input length
C_WrapKey	CKK_CAMELLIA	any	input length rounded up to multiple of the block size
C_UnwrapKey	CKK_CAMELLIA	multiple of block size	between 1 and block length bytes shorter than input length

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

6.31.7 General-length Camellia-MAC

General-length Camellia -MAC, denoted CKM_CAMELLIA_MAC_GENERAL, is a mechanism for single- and multiple-part signatures and verification, based on Camellia and data authentication as defined in.[CAMELLIA]

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

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

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

Table 74, General-length Camellia-MAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	CKK_CAMELLIA	any	0-block size, as specified in parameters
C_Verify	CKK_CAMELLIA	any	0-block size, as specified in parameters

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

6.31.8 Camellia-MAC

Camellia-MAC, denoted by **CKM_CAMELLIA_MAC**, is a special case of the general-length Camellia-MAC mechanism. Camellia-MAC always produces and verifies MACs that are half the block size in length.

It does not have a parameter.

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

Table 75, Camellia-MAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	CKK_CAMELLIA	any	½ block size (8 bytes)
C_Verify	CKK_CAMELLIA	any	½ block size (8 bytes)

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

6.32 Key derivation by data encryption - Camellia

These mechanisms allow derivation of keys using the result of an encryption operation as the key value. They are for use with the C_DeriveKey function.

6.32.1 Definitions

Mechanisms:

```
CKM_CAMELLIA_ECB_ENCRYPT_DATA
CKM_CAMELLIA_CBC_ENCRYPT_DATA

typedef struct CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS {
  CK_BYTE      iv[16];
  CK_BYTE_PTR  pData;
  CK_ULONG     length;
} CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS;
typedef CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS CK_PTR
CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS_PTR;
```

6.32.2 Mechanism Parameters

Uses CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS, and CK_KEY_DERIVATION_STRING_DATA.

Table 76, Mechanism Parameters for Camellia-based key derivation

CKM_CAMELLIA_ECB_ENCRYPT_DATA	Uses CK_KEY_DERIVATION_STRING_DATA structure. Parameter is the data to be encrypted and must be a multiple of 16 long.
CKM_CAMELLIA_CBC_ENCRYPT_DATA	Uses CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS. Parameter is an 16 byte IV value followed by the data. The data value part must be a multiple of 16 bytes long.

6.33 ARIA

ARIA is a block cipher with 128-bit block size and 128-, 192-, and 256-bit keys, similar to AES. ARIA is described in NSRI "Specification of ARIA".

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_ARIA_KEY_GEN					✓		
CKM_ARIA_ECB	✓					✓	
CKM_ARIA_CBC	✓					✓	
CKM_ARIA_CBC_PAD	✓					✓	
CKM_ARIA_MAC_GENERAL		✓					
CKM_ARIA_MAC		✓					

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_ARIA_ECB_ENCRYPT_DATA							✓
CKM_ARIA_CBC_ENCRYPT_DATA							✓

6.33.1 Definitions

This section defines the key type "CKK_ARIA" for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

```
CKM_ARIA_KEY_GEN
CKM_ARIA_ECB
CKM_ARIA_CBC
CKM_ARIA_MAC
CKM_ARIA_MAC_GENERAL
CKM_ARIA_CBC_PAD
```

6.33.2 Aria secret key objects

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

Table 77, ARIA Secret Key Object Attributes

Attribute	Data type	Meaning
CKA_VALUE[1,4,6,7]	Byte array	Key value (16, 24, or 32 bytes)
CKA_VALUE_LEN[2,3,6]	CK_ULONG	Length in bytes of key value

- Refer to [PKCS #11-B] table 15 for footnotes.

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

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_ARIA;
CK_UTF8CHAR label[] = “An ARIA secret key object”;
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
  {CKA_CLASS, &class, sizeof(class)},
  {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
  {CKA_TOKEN, &true, sizeof(true)},
  {CKA_LABEL, label, sizeof(label)-1},
```

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

6.33.3 ARIA key generation

The ARIA key generation mechanism, denoted CKM_ARIA_KEY_GEN, is a key generation mechanism for Aria.

It does not have a parameter.

The mechanism generates ARIA keys with a particular length in bytes, as specified in the **CKA_VALUE_LEN** attribute of the template for the key.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the ARIA key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

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

6.33.4 ARIA-ECB

ARIA-ECB, denoted **CKM_ARIA_ECB**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on Aria and electronic codebook mode.

It does not have a parameter.

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

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

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

Table 78, ARIA-ECB: Key And Data Length

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

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

6.33.5 ARIA-CBC

ARIA-CBC, denoted **CKM_ARIA_CBC**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on ARIA and cipher-block chaining mode.

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

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

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

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

Table 79, ARIA-CBC: Key And Data Length

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

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

6.33.6 ARIA-CBC with PKCS padding

ARIA-CBC with PKCS padding, denoted **CKM_ARIA_CBC_PAD**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on ARIA; cipher-block chaining mode; and the block cipher padding method detailed in PKCS #7.

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

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

In addition to being able to wrap and unwrap secret keys, this mechanism can wrap and unwrap RSA, Diffie-Hellman, X9.42 Diffie-Hellman, EC (also related to ECDSA) and DSA private keys (see Section TBA for details). The entries in the table below for data length constraints when wrapping and unwrapping keys do not apply to wrapping and unwrapping private keys.

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

Table 80, ARIA-CBC with PKCS Padding: Key And Data Length

Function	Key type	Input length	Output length
C_Encrypt	CKK_ARIA	any	input length rounded up to multiple of the block size
C_Decrypt	CKK_ARIA	multiple of block size	between 1 and block size bytes shorter than input length
C_WrapKey	CKK_ARIA	any	input length rounded up to multiple of the block size
C_UnwrapKey	CKK_ARIA	multiple of block size	between 1 and block length bytes shorter than input length

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

6.33.7 General-length ARIA-MAC

General-length ARIA -MAC, denoted **CKM_ARIA_MAC_GENERAL**, is a mechanism for single- and multiple-part signatures and verification, based on ARIA and data authentication as defined in [FIPS 113].

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

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

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

Table 81, General-length ARIA-MAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	CKK_ARIA	any	0-block size, as specified in parameters
C_Verify	CKK_ARIA	any	0-block size, as specified in parameters

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

6.33.8 ARIA-MAC

ARIA-MAC, denoted by **CKM_ARIA_MAC**, is a special case of the general-length ARIA-MAC mechanism. ARIA-MAC always produces and verifies MACs that are half the block size in length.

It does not have a parameter.

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

Table 82, ARIA-MAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	CKK_ARIA	any	½ block size (8 bytes)
C_Verify	CKK_ARIA	any	½ block size (8 bytes)

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

6.34 Key derivation by data encryption - ARIA

These mechanisms allow derivation of keys using the result of an encryption operation as the key value. They are for use with the C_DeriveKey function.

6.34.1 Definitions

Mechanisms:

```
CKM_ARIA_ECB_ENCRYPT_DATA
CKM_ARIA_CBC_ENCRYPT_DATA

typedef struct CK_ARIA_CBC_ENCRYPT_DATA_PARAMS {
  CK_BYTE      iv[16];
  CK_BYTE_PTR  pData;
  CK_ULONG     length;
} CK_ARIA_CBC_ENCRYPT_DATA_PARAMS;
typedef CK_ARIA_CBC_ENCRYPT_DATA_PARAMS CK_PTR
CK_ARIA_CBC_ENCRYPT_DATA_PARAMS_PTR;
```

6.34.2 Mechanism Parameters

Uses CK_ARIA_CBC_ENCRYPT_DATA_PARAMS, and CK_KEY_DERIVATION_STRING_DATA.

Table 83, Mechanism Parameters for Aria-based key derivation

CKM_ARIA_ECB_ENCRYPT_DATA	Uses CK_KEY_DERIVATION_STRING_DATA structure. Parameter is the data to be encrypted and must be a multiple of 16 long.
CKM_ARIA_CBC_ENCRYPT_DATA	Uses CK_ARIA_CBC_ENCRYPT_DATA_PARAMS. Parameter is an 16 byte IV value followed by the data. The data value part must be a multiple of 16 bytes long.

6.35 SEED

SEED is a symmetric block cipher developed by the South Korean Information Security Agency (KISA). It has a 128-bit key size and a 128-bit block size.

Its specification has been published as Internet [RFC 4269].

```
RFCs have been published defining the use of SEED in
TLS    ftp://ftp.rfc-editor.org/in-notes/rfc4162.txt
IPsec  ftp://ftp.rfc-editor.org/in-notes/rfc4196.txt
CMS    ftp://ftp.rfc-editor.org/in-notes/rfc4010.txt

TLS cipher suites that use SEED include:
      CipherSuite TLS_RSA_WITH_SEED_CBC_SHA      = { 0x00, 0x96};
      CipherSuite TLS_DH_DSS_WITH_SEED_CBC_SHA   = { 0x00, 0x97};
      CipherSuite TLS_DH_RSA_WITH_SEED_CBC_SHA   = { 0x00, 0x98};
      CipherSuite TLS_DHE_DSS_WITH_SEED_CBC_SHA  = { 0x00, 0x99};
      CipherSuite TLS_DHE_RSA_WITH_SEED_CBC_SHA  = { 0x00, 0x9A};
      CipherSuite TLS_DH_anon_WITH_SEED_CBC_SHA  = { 0x00, 0x9B};
```

As with any block cipher, it can be used in the ECB, CBC, OFB and CFB modes of operation, as well as in a MAC algorithm such as HMAC.

OIDs have been published for all these uses. A list may be seen at http://www.alvestrand.no/objectid/1.2.410.200004.1.html

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SEED_KEY_GEN					✓		
CKM_SEED_ECB			✓				
CKM_SEED_CBC			✓				
CKM_SEED_CBC_PAD	✓					✓	
CKM_SEED_MAC_GENERAL			✓				

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SEED_MAC				✓			
CKM_SEED_ECB_ENCRYPT_DATA							✓
CKM_SEED_CBC_ENCRYPT_DATA							✓

6.35.1 Definitions

This section defines the key type "CKK_SEED" for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

```
CKM_SEED_KEY_GEN
CKM_SEED_ECB
CKM_SEED_CBC
CKM_SEED_MAC
CKM_SEED_MAC_GENERAL
CKM_SEED_CBC_PAD
```

For all of these mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** are always 16.

6.35.2 SEED secret key objects

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

Table 84, SEED Secret Key Object Attributes

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

- Refer to [PKCS #11-B] table 15 for footnotes.

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

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_SEED;
CK_UTF8CHAR label[] = "A SEED secret key object";
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
```



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

6.35.3 SEED key generation

The SEED key generation mechanism, denoted CKM_SEED_KEY_GEN, is a key generation mechanism for SEED.

It does not have a parameter.

The mechanism generates SEED keys.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the SEED key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

6.35.4 SEED-ECB

SEED-ECB, denoted **CKM_SEED_ECB**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on SEED and electronic codebook mode.

It does not have a parameter.

6.35.5 SEED-CBC

SEED-CBC, denoted **CKM_SEED_CBC**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on SEED and cipher-block chaining mode.

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

6.35.6 SEED-CBC with PKCS padding

SEED-CBC with PKCS padding, denoted **CKM_SEED_CBC_PAD**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on SEED; cipher-block chaining mode; and the block cipher padding method detailed in PKCS #7.

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

6.35.7 General-length SEED-MAC

General-length SEED-MAC, denoted **CKM_SEED_MAC_GENERAL**, is a mechanism for single- and multiple-part signatures and verification, based on SEED and data authentication as defined in 0.

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

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

6.35.8 SEED-MAC

SEED-MAC, denoted by **CKM_SEED_MAC**, is a special case of the general-length SEED-MAC mechanism. SEED-MAC always produces and verifies MACs that are half the block size in length.

It does not have a parameter.

6.36 Key derivation by data encryption - SEED

These mechanisms allow derivation of keys using the result of an encryption operation as the key value. They are for use with the C_DeriveKey function.

6.36.1 Definitions

Mechanisms:

```
CKM_SEED_ECB_ENCRYPT_DATA
CKM_SEED_CBC_ENCRYPT_DATA

typedef struct CK_SEED_CBC_ENCRYPT_DATA_PARAMS
        CK_CBC_ENCRYPT_DATA_PARAMS;
typedef CK_CBC_ENCRYPT_DATA_PARAMS CK_PTR
CK_CBC_ENCRYPT_DATA_PARAMS_PTR;
```

6.36.2 Mechanism Parameters

Table 85, Mechanism Parameters for SEED-based key derivation

CKM_SEED_ECB_ENCRYPT_DATA	Uses CK_KEY_DERIVATION_STRING_DATA structure. Parameter is the data to be encrypted and must be a multiple of 16 long.
CKM_SEED_CBC_ENCRYPT_DATA	Uses CK_CBC_ENCRYPT_DATA_PARAMS.

	Parameter is an 16 byte IV value followed by the data. The data value part must be a multiple of 16 bytes long.

6.37 OTP

6.37.1 Usage overview

OTP tokens represented as PKCS #11 mechanisms may be used in a variety of ways. The usage cases can be categorized according to the type of sought functionality.

6.37.2 Case 1: Generation of OTP values

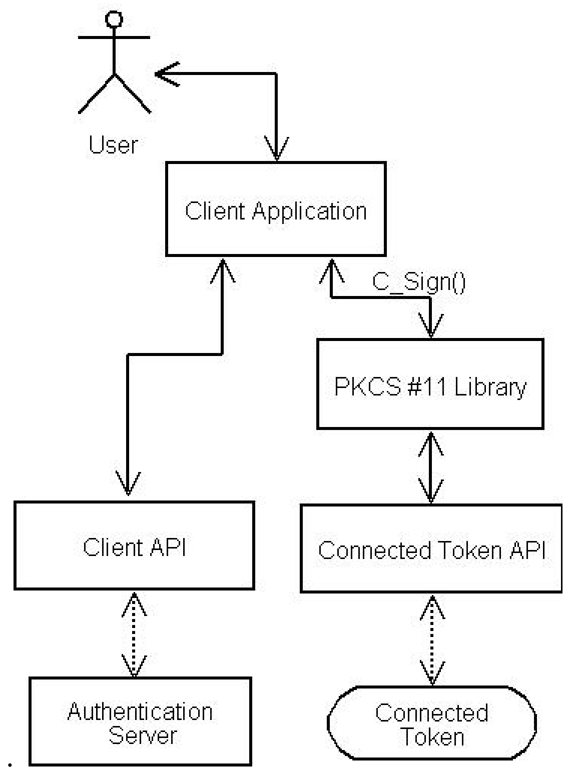

Figure 1: Retrieving OTP values through C_Sign

Figure 1 shows an integration of PKCS #11 into an application that needs to authenticate users holding OTP tokens. In this particular example, a connected hardware token is used, but a software token is equally possible. The application invokes **C_Sign** to retrieve the OTP value from the token. In the example, the application then passes the retrieved OTP value to a client API that sends it via the network to an authentication server. The client API may implement a standard authentication protocol such as RADIUS [RFC

2865] or EAP [RFC 3748], or a proprietary protocol such as that used by RSA Security's ACE/Agent® software.

6.37.3 Case 2: Verification of provided OTP values

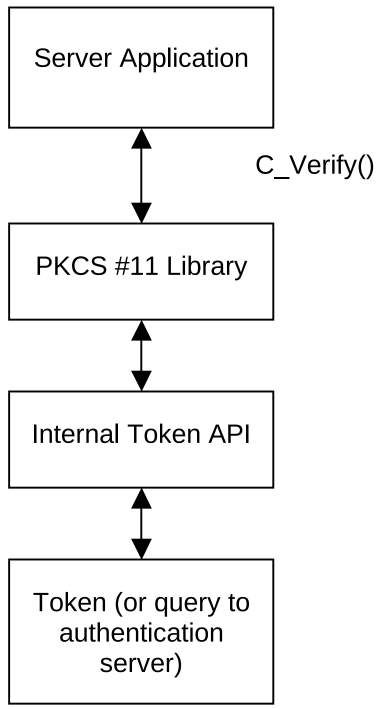

Figure 2: Server-side verification of OTP values

Figure 2 illustrates the server-side equivalent of the scenario depicted in Figure 1. In this case, a server application invokes **C_Verify** with the received OTP value as the signature value to be verified.

6.37.4 Case 3: Generation of OTP keys

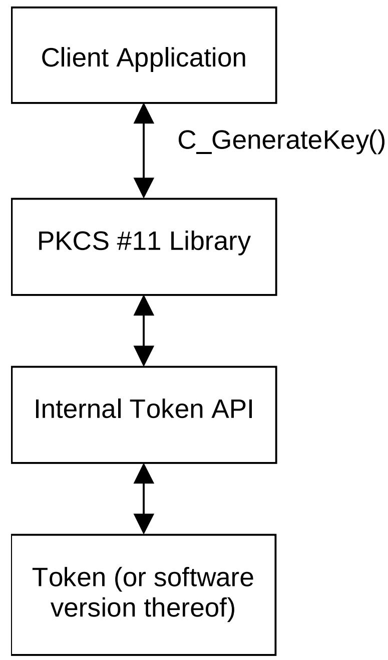

Figure 3: Generation of an OTP key

Figure 3 shows an integration of PKCS #11 into an application that generates OTP keys. The application invokes **C_GenerateKey** to generate an OTP key of a particular type on the token. The key may subsequently be used as a basis to generate OTP values.

6.37.5 OTP objects

6.37.5.1 Key objects

OTP key objects (object class **CKO_OTP_KEY**) hold secret keys used by OTP tokens. The following table defines the attributes common to all OTP keys, in addition to the attributes defined for secret keys, all of which are inherited by this class:

Table 86: Common OTP key attributes

Attribute	Data type	Meaning
CKA_OTP_FORMAT	CK_ULONG	Format of OTP values produced with this key: CK_OTP_FORMAT_DECIMAL = Decimal (default) (UTF8-encoded) CK_OTP_FORMAT_HEXADECIMAL = Hexadecimal (UTF8-encoded) CK_OTP_FORMAT_ALPHANUMERIC = Alphanumeric (UTF8-encoded) CK_OTP_FORMAT_BINARY = Only binary values.
CKA_OTP_LENGTH[9]	CK_ULONG	Default length of OTP values (in the CKA_OTP_FORMAT) produced with this key.
CKA_OTP_USER_FRIENDLY_MODE[9]	CK_BBOOL	Set to CK_TRUE when the token is capable of returning OTPs suitable for human consumption. See the description of CKF_USER_FRIENDLY_OTP below.
CKA_OTP _CHALLENGE_REQUIREMENT[9]	CK_ULONG	Parameter requirements when generating or verifying OTP values with this key: CK_OTP_PARAM_MANDATORY = A challenge must be supplied. CK_OTP_PARAM_OPTIONAL = A challenge may be supplied but need not be. CK_OTP_PARAM_IGNORED = A challenge, if supplied, will be ignored.
CKA_OTP_TIME_REQUIREMENT[9]	CK_ULONG	Parameter requirements when generating or verifying OTP values with this key: CK_OTP_PARAM_MANDATORY = A time value must be supplied. CK_OTP_PARAM_OPTIONAL = A time value may be supplied but need not be. CK_OTP_PARAM_IGNORED = A time value, if supplied, will be ignored.
CKA_OTP_COUNTER_REQUIREMENT[9]	CK_ULONG	Parameter requirements when generating or verifying OTP values with this key: CK_OTP_PARAM_MANDATORY = A counter value must be supplied. CK_OTP_PARAM_OPTIONAL = A counter value may be supplied but need not be. CK_OTP_PARAM_IGNORED = A counter value, if supplied, will be ignored.

Attribute	Data type	Meaning
CKA_OTP_PIN_REQUIREMENT[9]	CK_ULONG	Parameter requirements when generating or verifying OTP values with this key: CK_OTP_PARAM_MANDATORY = A PIN value must be supplied. CK_OTP_PARAM_OPTIONAL = A PIN value may be supplied but need not be (if not supplied, then library will be responsible for collecting it) CK_OTP_PARAM_IGNORED = A PIN value, if supplied, will be ignored.
CKA_OTP_COUNTER	Byte array	Value of the associated internal counter. Default value is empty (i.e. *ulValueLen* = 0).
CKA_OTP_TIME	RFC 2279 string	Value of the associated internal UTC time in the form YYYYMMDDhhmmss. Default value is empty (i.e. *ulValueLen*= 0).
CKA_OTP_USER_IDENTIFIER	RFC 2279 string	Text string that identifies a user associated with the OTP key (may be used to enhance the user experience). Default value is empty (i.e. *ulValueLen* = 0).
CKA_OTP_SERVICE_IDENTIFIER	RFC 2279 string	Text string that identifies a service that may validate OTPs generated by this key. Default value is empty (i.e. *ulValueLen* = 0).
CKA_OTP_SERVICE_LOGO	Byte array	Logotype image that identifies a service that may validate OTPs generated by this key. Default value is empty (i.e. *ulValueLen* = 0).
CKA_OTP_SERVICE_LOGO_TYPE	RFC 2279 string	MIME type of the CKA_OTP_SERVICE_LOGO attribute value. Default value is empty (i.e. *ulValueLen* = 0).
CKA_VALUE[1, 4, 6, 7]	Byte array	Value of the key.
CKA_VALUE_LEN[2, 3]	CK_ULONG	Length in bytes of key value.

Refer to [PKCS #11-B] Table 15 for table footnotes.

Note: A Cryptoki library may support PIN-code caching in order to reduce user interactions. An OTP-PKCS #11 application should therefore always consult the state of the CKA_OTP_PIN_REQUIREMENT attribute before each call to **C_SignInit**, as the value of this attribute may change dynamically.

For OTP tokens with multiple keys, the keys may be enumerated using **C_FindObjects**. The **CKA_OTP_SERVICE_IDENTIFIER** and/or the **CKA_OTP_SERVICE_LOGO** attribute may be used to distinguish between keys. The actual choice of key for a particular operation is however application-specific and beyond the scope of this document.

For all OTP keys, the CKA_ALLOWED_MECHANISMS attribute should be set as required.

6.37.6 OTP-related notifications

This document extends the set of defined notifications as follows:

CKN_OTP_CHANGED	Cryptoki is informing the application that the OTP for a key on a connected token just changed. This notification is particularly useful when applications wish to display the current OTP value for time-based mechanisms.

6.37.7 OTP mechanisms

The following table shows, for the OTP mechanisms defined in this document, their support by different cryptographic operations. For any particular token, of course, a particular operation may well support only a subset of the mechanisms listed. There is also no guarantee that a token that supports one mechanism for some operation supports any other mechanism for any other operation (or even supports that same mechanism for any other operation).

Table 87: OTP mechanisms vs. applicable functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SECURID_KEY_GEN					✓		
CKM_SECURID		✓					
CKM_HOTP_KEY_GEN					✓		
CKM_HOTP		✓					
CKM_ACTI_KEY_GEN					✓		
CKM_ACTI		✓					

The remainder of this section will present in detail the OTP mechanisms and the parameters that are supplied to them.

6.37.7.1 OTP mechanism parameters

♦ **CK_PARAM_TYPE**

CK_PARAM_TYPE is a value that identifies an OTP parameter type. It is defined as follows:


```
typedef CK_ULONG CK_PARAM_TYPE;
```

The following **CK_PARAM_TYPE** types are defined:

Table 88: OTP parameter types

Parameter	Data type	Meaning
CK_OTP_PIN	RFC 2279 string	A UTF8 string containing a PIN for use when computing or verifying PIN-based OTP values.
CK_OTP_CHALLENGE	Byte array	Challenge to use when computing or verifying challenge-based OTP values.
CK_OTP_TIME	RFC 2279 string	UTC time value in the form YYYYMMDDhhmmss to use when computing or verifying time-based OTP values.
CK_OTP_COUNTER	Byte array	Counter value to use when computing or verifying counter-based OTP values.
CK_OTP_FLAGS	CK_FLAGS	Bit flags indicating the characteristics of the sought OTP as defined below.
CK_OTP_OUTPUT_LENGTH	CK_ULONG	Desired output length (overrides any default value). A Cryptoki library will return CKR_MECHANISM_PARAM_INVALID if a provided length value is not supported.
CK_OTP_FORMAT	CK_ULONG	Returned OTP format (allowed values are the same as for CKA_OTP_FORMAT). This parameter is only intended for **C_Sign** output, see below. When not present, the returned OTP format will be the same as the value of the CKA_OTP_FORMAT attribute for the key in question.
CK_OTP_VALUE	Byte array	An actual OTP value. This parameter type is intended for **C_Sign** output, see below.

The following table defines the possible values for the CK_OTP_FLAGS type:

Table 89: OTP Mechanism Flags

Bit flag	Mask	Meaning
CKF_NEXT_OTP	0x00000001	True (i.e. set) if the OTP computation shall be for the next OTP, rather than the current one (current being interpreted in the context of the algorithm, e.g. for the current counter value or current time window). A Cryptoki library shall return CKR_MECHANISM_PARAM_INVALID if the CKF_NEXT_OTP flag is set and the OTP mechanism in question does not support the concept of "next" OTP or the library is not capable of generating the next OTP[**].
CKF_EXCLUDE_TIME	0x00000002	True (i.e. set) if the OTP computation must not include a time value. Will have an effect only on mechanisms that do include a time value in the OTP computation and then only if the mechanism (and token) allows exclusion of this value. A Cryptoki library shall return CKR_MECHANISM_PARAM_INVALID if exclusion of the value is not allowed.
CKF_EXCLUDE_COUNTER	0x00000004	True (i.e. set) if the OTP computation must not include a counter value. Will have an effect only on mechanisms that do include a counter value in the OTP computation and then only if the mechanism (and token) allows exclusion of this value. A Cryptoki library shall return CKR_MECHANISM_PARAM_INVALID if exclusion of the value is not allowed.
CKF_EXCLUDE_CHALLENGE	0x00000008	True (i.e. set) if the OTP computation must not include a challenge. Will have an effect only on mechanisms that do include a challenge in the OTP computation and then only if the mechanism (and token) allows exclusion of this value. A Cryptoki library shall return CKR_MECHANISM_PARAM_INVALID if exclusion of the value is not allowed.

[**] Applications that may need to retrieve the next OTP should be prepared to handle this situation. For example, an application could store the OTP value returned by C_Sign so that, if a next OTP is required, it can compare it to the OTP value returned by subsequent calls to C_Sign should it turn out that the library does not support the CKF_NEXT_OTP flag.

Bit flag	Mask	Meaning
CKF_EXCLUDE_PIN	0x00000010	True (i.e. set) if the OTP computation must not include a PIN value. Will have an effect only on mechanisms that do include a PIN in the OTP computation and then only if the mechanism (and token) allows exclusion of this value. A Cryptoki library shall return CKR_MECHANISM_PARAM_INVALID if exclusion of the value is not allowed.
CKF_USER_FRIENDLY_OTP	0x00000020	True (i.e. set) if the OTP returned shall be in a form suitable for human consumption. If this flag is set, and the call is successful, then the returned CK_OTP_VALUE shall be a UTF8-encoded printable string. A Cryptoki library shall return CKR_MECHANISM_PARAM_INVALID if this flag is set when CKA_OTP_USER_FRIENDLY_MODE for the key in question is CK_FALSE.

Note: Even if CKA_OTP_FORMAT is not set to CK_OTP_FORMAT_BINARY, then there may still be value in setting the CKF_USER_FRIENDLY flag (assuming CKA_USER_FRIENDLY_MODE is CK_TRUE, of course) if the intent is for a human to read the generated OTP value, since it may become shorter or otherwise better suited for a user. Applications that do not intend to provide a returned OTP value to a user should not set the CKF_USER_FRIENDLY_OTP flag.

♦ **CK_OTP_PARAM; CK_OTP_PARAM_PTR**

CK_OTP_PARAM is a structure that includes the type, value, and length of an OTP parameter. It is defined as follows:

```
typedef struct CK_OTP_PARAM {
   CK_PARAM_TYPE type;
   CK_VOID_PTR pValue;
   CK_ULONG ulValueLen;
} CK_OTP_PARAM;
```

The fields of the structure have the following meanings:

type the parameter type

pValue pointer to the value of the parameter

ulValueLen length in bytes of the value

If a parameter has no value, then *ulValueLen* = 0, and the value of *pValue* is irrelevant. Note that *pValue* is a "void" pointer, facilitating the passing of arbitrary values. Both the application and the Cryptoki library must ensure that the pointer can be safely cast to the expected type (*i.e.*, without word-alignment errors).

CK_OTP_PARAM_PTR is a pointer to a **CK_OTP_PARAM.**

CK_OTP_PARAMS; CK_OTP_PARAMS_PTR

CK_OTP_PARAMS is a structure that is used to provide parameters for OTP mechanisms in a generic fashion. It is defined as follows:

```
typedef struct CK_OTP_PARAMS {
   CK_OTP_PARAM_PTR pParams;
   CK_ULONG ulCount;
} CK_OTP_PARAMS;
```

The fields of the structure have the following meanings:

pParams	pointer to an array of OTP parameters
ulCount	the number of parameters in the array

CK_OTP_PARAMS_PTR is a pointer to a **CK_OTP_PARAMS**.

When calling **C_SignInit** or **C_VerifyInit** with a mechanism that takes a **CK_OTP_PARAMS** structure as a parameter, the **CK_OTP_PARAMS** structure shall be populated in accordance with the **CKA_OTP_*X*_REQUIREMENT** key attributes for the identified key, where ***X*** is **PIN**, **CHALLENGE**, **TIME**, or **COUNTER**.

For example, if **CKA_OTP_TIME_REQUIREMENT** = CK_OTP_PARAM_MANDATORY, then the **CK_OTP_TIME** parameter shall be present. If **CKA_OTP_TIME_REQUIREMENT** = CK_OTP_PARAM_OPTIONAL, then a **CK_OTP_TIME** parameter may be present. If it is not present, then the library may collect it (during the **C_Sign** call). If **CKA_OTP_TIME_REQUIREMENT** = CK_OTP_PARAM_IGNORED, then a provided **CK_OTP_TIME** parameter will always be ignored. Additionally, a provided **CK_OTP_TIME** parameter will always be ignored if CKF_EXCLUDE_TIME is set in a **CK_OTP_FLAGS** parameter. Similarly, if this flag is set, a library will not attempt to collect the value itself, and it will also instruct the token not to make use of any internal value, subject to token policies. It is an error (**CKR_MECHANISM_PARAM_INVALID**) to set the CKF_EXCLUDE_TIME flag when the **CKA_TIME_REQUIREMENT** attribute is CK_OTP_PARAM_MANDATORY.

The above discussion holds for all **CKA_OTP_*X*_REQUIREMENT** attributes (*i.e.*, **CKA_OTP_PIN_REQUIREMENT**, **CKA_OTP_CHALLENGE_REQURIEMENT**, **CKA_OTP_COUNTER_REQUIREMENT**, **CKA_OTP_TIME_REQUIREMENT**). A library may set a particular **CKA_OTP_*X*_REQUIREMENT** attribute to CK_OTP_PARAM_OPTIONAL even if it is required by the mechanism as long as the token (or the library itself) has the capability of providing the value to the computation. One example of this is a token with an on-board clock.

In addition, applications may use the **CK_OTP_FLAGS**, the **CK_OTP_OUTPUT_FORMAT** and the **CK_OUTPUT_LENGTH** parameters to set additional parameters.

CK_OTP_SIGNATURE_INFO, CK_OTP_SIGNATURE_INFO_PTR

CK_OTP_SIGNATURE_INFO is a structure that is returned by all OTP mechanisms in successful calls to **C_Sign** (**C_SignFinal**). The structure informs applications of actual parameter values used in particular OTP computations in addition to the OTP value itself. It is used by all mechanisms for which the key belongs to the class CKO_OTP_KEY and is defined as follows:

```
typedef struct CK_OTP_SIGNATURE_INFO {
   CK_OTP_PARAM_PTR pParams;
   CK_ULONG ulCount;
} CK_OTP_SIGNATURE_INFO;
```

The fields of the structure have the following meanings:

pParams pointer to an array of OTP parameter values

ulCount the number of parameters in the array

After successful calls to **C_Sign** or **C_SignFinal** with an OTP mechanism, the *pSignature* parameter will be set to point to a **CK_OTP_SIGNATURE_INFO** structure. One of the parameters in this structure will be the OTP value itself, identified with the **CK_OTP_VALUE** tag. Other parameters may be present for informational purposes, e.g. the actual time used in the OTP calculation. In order to simplify OTP validations, authentication protocols may permit authenticating parties to send some or all of these parameters in addition to OTP values themselves. Applications should therefore check for their presence in returned **CK_OTP_SIGNATURE_INFO** values whenever such circumstances apply.

Since **C_Sign** and **C_SignFinal** follows the convention described in Section 11.2 on producing output, a call to **C_Sign** (or **C_SignFinal**) with *pSignature* set to NULL_PTR will return (in the *pulSignatureLen* parameter) the required number of bytes to hold the **CK_OTP_SIGNATURE_INFO** structure *as well as all the data in all its **CK_OTP_PARAM** components*. If an application allocates a memory block based on this information, it shall therefore not subsequently de-allocate components of such a received value but rather de-allocate the complete **CK_OTP_PARAMS** structure itself. A Cryptoki library that is called with a non-NULL *pSignature* pointer will assume that it points to a *contiguous* memory block of the size indicated by the *pulSignatureLen* parameter.

When verifying an OTP value using an OTP mechanism, *pSignature* shall be set to the OTP value itself, e.g. the value of the **CK_OTP_VALUE** component of a **CK_OTP_PARAMS** structure returned by a call to **C_Sign**. The **CK_OTP_PARAMS** value supplied in the **C_VerifyInit** call sets the values to use in the verification operation.

CK_OTP_SIGNATURE_INFO_PTR points to a **CK_OTP_SIGNATURE_INFO.**

6.37.8 RSA SecurID

6.37.8.1 RSA SecurID secret key objects

RSA SecurID secret key objects (object class **CKO_OTP_KEY,** key type **CKK_SECURID**) hold RSA SecurID secret keys. The following table defines the RSA SecurID secret key object attributes, in addition to the common attributes defined for this object class:

Table 90: RSA SecurID secret key object attributes

Attribute	Data type	Meaning
CKA_OTP_TIME_INTERVAL[1]	CK_ULONG	Interval between OTP values produced with this key, in seconds. Default is 60.

Refer to [PKCS #11-B] Table 15 for table footnotes.

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

```
CK_OBJECT_CLASS class = CKO_OTP_KEY;
CK_KEY_TYPE keyType = CKK_SECURID;
CK_DATE endDate = {...};
CK_UTF8CHAR label[] = “RSA SecurID secret key object”;
CK_BYTE keyId[]= {...};
CK_ULONG outputFormat = CK_OTP_FORMAT_DECIMAL;
CK_ULONG outputLength = 6;
CK_ULONG needPIN = CK_OTP_PARAM_MANDATORY;
CK_ULONG timeInterval = 60;
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
   {CKA_CLASS, &class, sizeof(class)},
   {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   {CKA_END_DATE, &endDate, sizeof(endDate)},
   {CKA_TOKEN, &true, sizeof(true)},
   {CKA_SENSITIVE, &true, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)-1},
   {CKA_SIGN, &true, sizeof(true)},
   {CKA_VERIFY, &true, sizeof(true)},
   {CKA_ID, keyId, sizeof(keyId)},
   {CKA_OTP_FORMAT, &outputFormat, sizeof(outputFormat)},
   {CKA_OTP_LENGTH, &outputLength, sizeof(outputLength)},
   {CKA_OTP_PIN_REQUIREMENT, &needPIN, sizeof(needPIN)},
   {CKA_OTP_TIME_INTERVAL, &timeInterval,
        sizeof(timeInterval)},
   {CKA_VALUE, value, sizeof(value)}
};
```


6.37.9 RSA SecurID key generation

The RSA SecurID key generation mechanism, denoted **CKM_SECURID_KEY_GEN**, is a key generation mechanism for the RSA SecurID algorithm.

It does not have a parameter.

The mechanism generates RSA SecurID keys with a particular set of attributes as specified in the template for the key.

The mechanism contributes at least the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_VALUE_LEN**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the RSA SecurID key type may be specified in the template for the key, or else are assigned default initial values

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

6.37.10RSA SecurID OTP generation and validation

CKM_SECURID is the mechanism for the retrieval and verification of RSA SecurID OTP values.

The mechanism takes a pointer to a **CK_OTP_PARAMS** structure as a parameter.

When signing or verifying using the **CKM_SECURID** mechanism, *pData* shall be set to NULL_PTR and *ulDataLen* shall be set to 0.

6.37.11Return values

Support for the **CKM_SECURID** mechanism extends the set of return values for **C_Verify** with the following values:

- CKR_NEW_PIN_MODE: The supplied OTP was not accepted and the library requests a new OTP computed using a new PIN. The new PIN is set through means out of scope for this document.
- CKR_NEXT_OTP: The supplied OTP was correct but indicated a larger than normal drift in the token's internal state (e.g. clock, counter). To ensure this was not due to a temporary problem, the application should provide the next one-time password to the library for verification.

6.37.12 OATH HOTP

6.37.12.1 OATH HOTP secret key objects

HOTP secret key objects (object class **CKO_OTP_KEY,** key type **CKK_HOTP**) hold generic secret keys and associated counter values.

The **CKA_OTP_COUNTER** value may be set at key generation; however, some tokens may set it to a fixed initial value. Depending on the token's security policy, this value may not be modified and/or may not be revealed if the object has its **CKA_SENSITIVE** attribute set to CK_TRUE or its **CKA_EXTRACTABLE** attribute set to CK_FALSE.

For HOTP keys, the **CKA_OTP_COUNTER** value shall be an 8 bytes unsigned integer in big endian (i.e. network byte order) form. The same holds true for a **CK_OTP_COUNTER** value in a **CK_OTP_PARAM** structure.

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

```
CK_OBJECT_CLASS class = CKO_OTP_KEY;
CK_KEY_TYPE keyType = CKK_HOTP;
CK_UTF8CHAR label[] = "HOTP secret key object";
CK_BYTE keyId[]= {...};
CK_ULONG outputFormat = CK_OTP_FORMAT_DECIMAL;
CK_ULONG outputLength = 6;
CK_DATE endDate = {...};
CK_BYTE counterValue[8] = {0};
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
   {CKA_CLASS, &class, sizeof(class)},
   {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   {CKA_END_DATE, &endDate, sizeof(endDate)},
   {CKA_TOKEN, &true, sizeof(true)},
   {CKA_SENSITIVE, &true, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)-1},
   {CKA_SIGN, &true, sizeof(true)},
   {CKA_VERIFY, &true, sizeof(true)},
   {CKA_ID, keyId, sizeof(keyId)},
   {CKA_OTP_FORMAT, &outputFormat, sizeof(outputFormat)},
   {CKA_OTP_LENGTH, &outputLength, sizeof(outputLength)},
   {CKA_OTP_COUNTER, counterValue, sizeof(counterValue)},
   {CKA_VALUE, value, sizeof(value)}
};
```

6.37.12.2 HOTP key generation

The HOTP key generation mechanism, denoted **CKM_HOTP_KEY_GEN**, is a key generation mechanism for the HOTP algorithm.

It does not have a parameter.

The mechanism generates HOTP keys with a particular set of attributes as specified in the template for the key.

The mechanism contributes at least the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_OTP_COUNTER**, **CKA_VALUE** and **CKA_VALUE_LEN** attributes to the new key. Other attributes supported by the HOTP key type may be specified in the template for the key, or else are assigned default initial values.

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

6.37.12.3 HOTP OTP generation and validation

CKM_HOTP is the mechanism for the retrieval and verification of HOTP OTP values based on the current internal counter, or a provided counter.

The mechanism takes a pointer to a **CK_OTP_PARAMS** structure as a parameter.

As for the **CKM_SECURID** mechanism, when signing or verifying using the **CKM_HOTP** mechanism, *pData* shall be set to NULL_PTR and *ulDataLen* shall be set to 0.

For verify operations, the counter value **CK_OTP_COUNTER** must be provided as a **CK_OTP_PARAM** parameter to **C_VerifyInit**. When verifying an OTP value using the **CKM_HOTP** mechanism, *pSignature* shall be set to the OTP value itself, e.g. the value of the **CK_OTP_VALUE** component of a **CK_OTP_PARAMS** structure in the case of an earlier call to **C_Sign**.

6.37.13ActivIdentity ACTI

6.37.13.1 ACTI secret key objects

ACTI secret key objects (object class **CKO_OTP_KEY,** key type **CKK_ACTI**) hold ActivIdentity ACTI secret keys.

For ACTI keys, the **CKA_OTP_COUNTER** value shall be an 8 bytes unsigned integer in big endian (i.e. network byte order) form. The same holds true for the **CK_OTP_COUNTER** value in the **CK_OTP_PARAM** structure.

The **CKA_OTP_COUNTER** value may be set at key generation; however, some tokens may set it to a fixed initial value. Depending on the token’s security policy, this value may not be modified and/or may not be revealed if the object has its

CKA_SENSITIVE attribute set to CK_TRUE or its **CKA_EXTRACTABLE** attribute set to CK_FALSE.

The **CKA_OTP_TIME** value may be set at key generation; however, some tokens may set it to a fixed initial value. Depending on the token's security policy, this value may not be modified and/or may not be revealed if the object has its **CKA_SENSITIVE** attribute set to CK_TRUE or its **CKA_EXTRACTABLE** attribute set to CK_FALSE.

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

```
CK_OBJECT_CLASS class = CKO_OTP_KEY;
CK_KEY_TYPE keyType = CKK_ACTI;
CK_UTF8CHAR label[] = "ACTI secret key object";
CK_BYTE keyId[]= {...};
CK_ULONG outputFormat = CK_OTP_FORMAT_DECIMAL;
CK_ULONG outputLength = 6;
CK_DATE endDate = {...};
CK_BYTE counterValue[8] = {0};
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
   {CKA_CLASS, &class, sizeof(class)},
   {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   {CKA_END_DATE, &endDate, sizeof(endDate)},
   {CKA_TOKEN, &true, sizeof(true)},
   {CKA_SENSITIVE, &true, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)-1},
   {CKA_SIGN, &true, sizeof(true)},
   {CKA_VERIFY, &true, sizeof(true)},
   {CKA_ID, keyId, sizeof(keyId)},
   {CKA_OTP_FORMAT, &outputFormat,
   sizeof(outputFormat)},
   {CKA_OTP_LENGTH, &outputLength,
   sizeof(outputLength)},
   {CKA_OTP_COUNTER, counterValue,
   sizeof(counterValue)},
   {CKA_VALUE, value, sizeof(value)}
};
```

6.37.13.2 ACTI key generation

The ACTI key generation mechanism, denoted **CKM_ACTI_KEY_GEN**, is a key generation mechanism for the ACTI algorithm.

It does not have a parameter.

The mechanism generates ACTI keys with a particular set of attributes as specified in the template for the key.

The mechanism contributes at least the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_VALUE** and **CKA_VALUE_LEN** attributes to the new key. Other attributes supported by the ACTI key type may be specified in the template for the key, or else are assigned default initial values.

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

6.37.14ACTI OTP generation and validation

CKM_ACTI is the mechanism for the retrieval and verification of ACTI OTP values.

The mechanism takes a pointer to a **CK_OTP_PARAMS** structure as a parameter.

When signing or verifying using the **CKM_ACTI** mechanism, *pData* shall be set to NULL_PTR and *ulDataLen* shall be set to 0.

When verifying an OTP value using the **CKM_ACTI** mechanism, *pSignature* shall be set to the OTP value itself, e.g. the value of the **CK_OTP_VALUE** component of a **CK_OTP_PARAMS** structure in the case of an earlier call to **C_Sign**.

6.38 CT-KIP

6.38.1 Principles of Operation

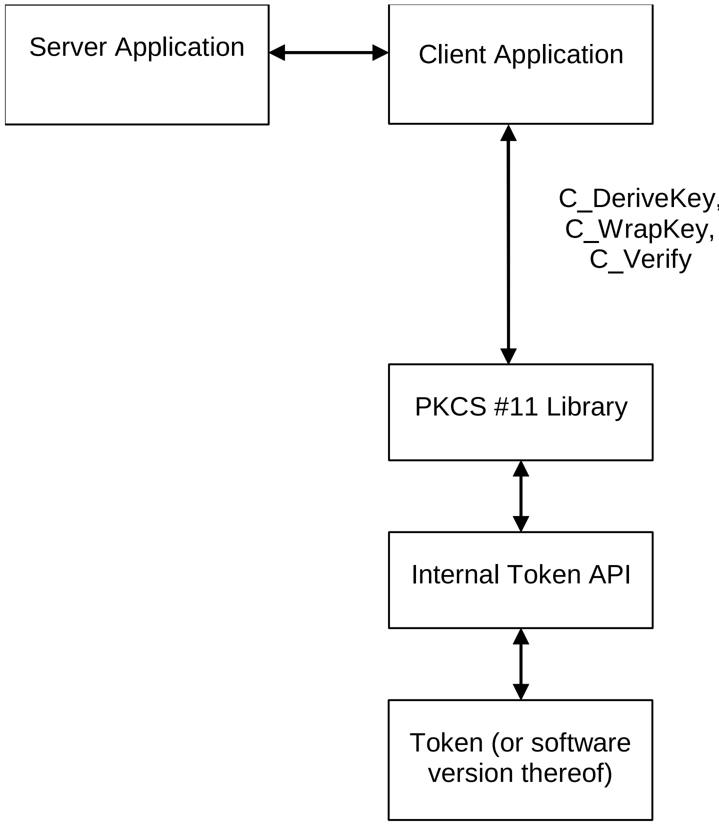

Figure 4: PKCS #11 and CT-KIP integration

Figure 3 shows an integration of PKCS #11 into an application that generates cryptographic keys through the use of CT-KIP. The application invokes **C_DeriveKey** to derive a key of a particular type on the token. The key may subsequently be used as a basis to e.g., generate one-time password values. The application communicates with a CT-KIP server that participates in the key derivation and stores a copy of the key in its database. The key is transferred to the server in wrapped form, after a call to **C_WrapKey**. The server authenticates itself to the client and the client verifies the authentication by calls to **C_Verify**.

6.38.2 Mechanisms

The following table shows, for the mechanisms defined in this document, their support by different cryptographic operations. For any particular token, of course, a particular operation may well support only a subset of the mechanisms listed. There is also no guarantee that a token that supports one mechanism for some operation supports any other mechanism for any other operation (or even supports that same mechanism for any other operation).

Table 91: Mechanisms vs. applicable functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_KIP_DERIVE							✓
CKM_KIP_WRAP						✓	
CKM_KIP_MAC		✓					

The remainder of this section will present in detail the mechanisms and the parameters that are supplied to them.

6.38.3 Definitions

Mechanisms:

```
CKM_KIP_DERIVE
CKM_KIP_WRAP
CKM_KIP_MAC
```

6.38.4 CT-KIP Mechanism parameters

♦ **CK_KIP_ PARAMS; CK_KIP_ PARAMS_PTR**

CK_KIP_PARAMS is a structure that provides the parameters to all the CT-KIP related mechanisms: The **CKM_KIP_DERIVE** key derivation mechanism, the **CKM_KIP_WRAP** key wrap and key unwrap mechanism, and the **CKM_KIP_MAC** signature mechanism. The structure is defined as follows:

```
typedef struct CK_KIP_PARAMS {
   CK_MECHANISM_PTR  pMechanism;
   CK_OBJECT_HANDLE  hKey;
   CK_BYTE_PTR       pSeed;
   CK_ULONG          ulSeedLen;
} CK_KIP_PARAMS;
```

The fields of the structure have the following meanings:

pMechanism pointer to the underlying cryptographic mechanism (e.g. AES, SHA-256), see further 0, Appendix D

hKey handle to a key that will contribute to the entropy of the derived key (CKM_KIP_DERIVE) or will be used in the MAC operation (CKM_KIP_MAC)

pSeed pointer to an input seed

ulSeedLen length in bytes of the input seed

CK_KIP_PARAMS_PTR is a pointer to a **CK_KIP_PARAMS** structure.

6.38.5 CT-KIP key derivation

The CT-KIP key derivation mechanism, denoted **CKM_KIP_DERIVE**, is a key derivation mechanism that is capable of generating secret keys of potentially any type, subject to token limitations.

It takes a parameter of type **CK_KIP_PARAMS** which allows for the passing of the desired underlying cryptographic mechanism as well as some other data. In particular, when the *hKey* parameter is a handle to an existing key, that key will be used in the key derivation in addition to the *hBaseKey* of **C_DeriveKey**. The *pSeed* parameter may be used to seed the key derivation operation.

The mechanism derives a secret key with a particular set of attributes as specified in the attributes of the template for the key.

The mechanism contributes the **CKA_CLASS** and **CKA_VALUE** attributes to the new key. Other attributes supported by the key type may be specified in the template for the key, or else will be assigned default initial values. Since the mechanism is generic, the **CKA_KEY_TYPE** attribute should be set in the template, if the key is to be used with a particular mechanism.

6.38.6 CT-KIP key wrap and key unwrap

The CT-KIP key wrap and unwrap mechanism, denoted **CKM_KIP_WRAP**, is a key wrap mechanism that is capable of wrapping and unwrapping generic secret keys.

It takes a parameter of type **CK_KIP_PARAMS**, which allows for the passing of the desired underlying cryptographic mechanism as well as some other data. It does not make use of the *hKey* parameter of **CK_KIP_PARAMS**.

6.38.7 CT-KIP signature generation

The CT-KIP signature (MAC) mechanism, denoted **CKM_KIP_MAC**, is a mechanism used to produce a message authentication code of arbitrary length. The keys it uses are secret keys.

It takes a parameter of type **CK_KIP_PARAMS**, which allows for the passing of the desired underlying cryptographic mechanism as well as some other data. The mechanism does not make use of the *pSeed* and the *ulSeedLen* parameters of **CT_KIP_PARAMS**.

This mechanism produces a MAC of the length specified by *pulSignatureLen* parameter in calls to **C_Sign**.

If a call to **C_Sign** with this mechanism fails, then no output will be generated.

6.39 GOST

Table 1, Mechanisms vs. Functions

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

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_GOST28147_KEY_GEN					√		
CKM_ GOST28147_ECB	√					√	
CKM_GOST28147	√					√	
CKM_ GOST28147_MAC		√					
CKM_ GOST28147_KEY_WRAP						√	
CKM_GOSTR3411				√			
CKM_GOSTR3411_HMAC		√					
CKM_GOSTR3410_KEY_PAIR_GEN					√		
CKM_GOSTR3410		√1					
CKM_GOSTR3410_WITH_ GOST3411		√					
CKM_GOSTR3410_KEY_WRAP						√	
CKM_GOSTR3410_DERIVE							√

[1] Single-part operations only

6.40 GOST 28147-89

GOST 28147-89 is a block cipher with 64-bit block size and 256-bit keys.

6.40.1 Definitions

This section defines the key type "CKK_GOST28147" for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects and domain parameter objects.

Mechanisms:

```
CKM_GOST28147_KEY_GEN
          CKM_GOST28147_ECB
          CKM_GOST28147
          CKM_GOST28147_MAC
          CKM_GOST28147_KEY_WRAP
```

6.40.2 GOST 28147-89 secret key objects

GOST 28147-89 secret key objects (object class **CKO_SECRET_KEY,** key type **CKK_GOST28147**) hold GOST 28147-89 keys. The following table defines the

GOST 28147-89 secret key object attributes, in addition to the common attributes defined for this object class:

Table 2, GOST 28147-89 Secret Key Object Attributes

Attribute	Data type	Meaning
CKA_VALUE[1,4,6,7]	Byte array	32 bytes in little endian order
CKA_GOST28147_PARAMS[1,3,5]	Byte array	DER-encoding of the object identifier indicating the data object type of GOST 28147-89. When key is used the domain parameter object of key type CKK_GOST28147 must be specified with the same attribute CKA_OBJECT_ID

Refer to [PKCS #11-B] Table 15 for footnotes

The following is a sample template for creating a GOST 28147-89 secret key object:

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_GOST28147;
CK_UTF8CHAR label[] = “A GOST 28147-89 secret key
        object”;
CK_BYTE value[32] = {...};
CK_BYTE params_oid[] = {0x06, 0x07, 0x2a, 0x85, 0x03,
        0x02, 0x02, 0x1f, 0x00};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_ENCRYPT, &true, sizeof(true)},
    {CKA_GOST28147_PARAMS, params_oid,
        sizeof(params_oid)},
    {CKA_VALUE, value, sizeof(value)}
};
```

6.40.3 GOST 28147-89 domain parameter objects

GOST 28147-89 domain parameter objects (object class **CKO_DOMAIN_PARAMETERS,** key type **CKK_GOST28147**) hold GOST 28147-89 domain parameters.

The following table defines the GOST 28147-89 domain parameter object attributes, in addition to the common attributes defined for this object class:

Table 3, GOST 28147-89 Domain Parameter Object Attributes

Attribute	Data Type	Meaning
CKA_VALUE[1]	Byte array	DER-encoding of the domain parameters as it was introduced in [4] section 8.1 (type *Gost28147-89-ParamSetParameters*)
CKA_OBJECT_ID[1]	Byte array	DER-encoding of the object identifier indicating the domain parameters

Refer to [PKCS #11-B] Table 15 for footnotes

For any particular token, there is no guarantee that a token supports domain parameters loading up and/or fetching out. Furthermore, applications, that make direct use of domain parameters objects, should take in account that **CKA_VALUE** attribute may be inaccessible.

The following is a sample template for creating a GOST 28147-89 domain parameter object:

```
CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
CK_KEY_TYPE keyType = CKK_GOST28147;
CK_UTF8CHAR label[] = “A GOST 28147-89 cryptographic
        parameters object”;
CK_BYTE oid[] = {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02,
        0x02, 0x1f, 0x00};
CK_BYTE value[] = {
0x30,0x62,
0x04,0x40,
0x4c,0xde,0x38,0x9c,0x29,0x89,0xef,0xb6,0xff,0xeb,0x56,0xc5,0x5e,0xc2,0x9b,0x02,
0x98,0x75,0x61,0x3b,0x11,0x3f,0x89,0x60,0x03,0x97,0x0c,0x79,0x8a,0xa1,0xd5,0x5d,
0xe2,0x10,0xad,0x43,0x37,0x5d,0xb3,0x8e,0xb4,0x2c,0x77,0xe7,0xcd,0x46,0xca,0xfa,
0xd6,0x6a,0x20,0x1f,0x70,0xf4,0x1e,0xa4,0xab,0x03,0xf2,0x21,0x65,0xb8,0x44,0xd8,
0x02,0x01,0x00,
0x02,0x01,0x40,
0x30,0x0b,0x06,0x07,0x2a,0x85,0x03,0x02,0x02,0x0e,0x00,0x05,0x00
};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_OBJECT_ID, oid, sizeof(oid)},
    {CKA_VALUE, value, sizeof(value)}
};
```

6.40.4 GOST 28147-89 key generation

The GOST 28147-89 key generation mechanism, denoted **CKM_GOST28147_KEY_GEN**, is a key generation mechanism for GOST 28147-89.

It does not have a parameter.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the GOST 28147-89 key type may be specified for objects of object class **CKO_SECRET_KEY**.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** are not used.

6.40.5 GOST 28147-89-ECB

GOST 28147-89-ECB, denoted **CKM_GOST28147_ECB**, is a mechanism for single and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on GOST 28147-89 and electronic codebook mode.

It does not have a parameter.

This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to wrap/unwrap every secret key that it supports.

For wrapping (**C_WrapKey**), the mechanism encrypts the value of the **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size so that the resulting length is a multiple of the block size.

For unwrapping (**C_UnwrapKey**), the mechanism decrypts the wrapped key, and truncates the result according to the **CKA_KEY_TYPE** attribute of the template and, if it has one, and the key type supports it, the **CKA_VALUE_LEN** attribute of the template. The mechanism contributes the result as the **CKA_VALUE** attribute of the new key.

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

Table 4, GOST 28147-89-ECB: Key And Data Length

Function	Key type	Input length	Output length
C_Encrypt	CKK_GOST28147	Multiple of block size	Same as input length
C_Decrypt	CKK_GOST28147	Multiple of block size	Same as input length
C_WrapKey	CKK_GOST28147	Any	Input length rounded up to multiple of block size
C_UnwrapKey	CKK_GOST28147	Multiple of	Determined by type of key

		block size	being unwrapped

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure are not used.

6.40.6 GOST 28147-89 encryption mode except ECB

GOST 28147-89 encryption mode except ECB, denoted **CKM_GOST28147**, is a mechanism for single and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on [GOST 28147-89] and CFB, counter mode, and additional CBC mode defined in [RFC 4357] section 2. Encryption's parameters are specified in object identifier of attribute **CKA_GOST28147_PARAMS**.

It has a parameter, a 8-byte initialization vector. This parameter may be omitted then a zero initialization vector is used.

This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to wrap/unwrap every secret key that it supports.

For wrapping (**C_WrapKey**), the mechanism encrypts the value of the **CKA_VALUE** attribute of the key that is wrapped.

For unwrapping (**C_UnwrapKey**), the mechanism decrypts the wrapped key, and contributes the result as the **CKA_VALUE** attribute of the new key.

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

Table 5, GOST 28147-89 encryption modes except ECB: Key And Data Length

Function	**Key type**	**Input length**	**Output length**
C_Encrypt	CKK_GOST28147	Any	For counter mode and CFB is the same as input length. For CBC is the same as input length padded on the trailing end with up to block size so that the resulting length is a multiple of the block size
C_Decrypt	CKK_GOST28147	Any	
C_WrapKey	CKK_GOST28147	Any	
C_UnwrapKey	CKK_GOST28147	Any	

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure are not used.

6.40.7 GOST 28147-89-MAC

GOST 28147-89-MAC, denoted **CKM_GOST28147_MAC**, is a mechanism for data integrity and authentication based on GOST 28147-89 and key meshing algorithms [RFC 4357] section 2.3.

MACing parameters are specified in object identifier of attribute **CKA_GOST28147_PARAMS**.

The output bytes from this mechanism are taken from the start of the final GOST 28147-89 cipher block produced in the MACing process.

It has a parameter, a 8-byte MAC initialization vector. This parameter may be omitted then a zero initialization vector is used.

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

Table 6, GOST28147-89-MAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	CKK_GOST28147	Any	4 bytes
C_Verify	CKK_GOST28147	Any	4 bytes

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure are not used.

GOST 28147-89 keys wrapping/unwrapping with GOST 28147-89

GOST 28147-89 keys as a KEK (key encryption keys) for encryption GOST 28147-89 keys, denoted by **CKM_GOST28147_KEY_WRAP**, is a mechanism for key wrapping; and key unwrapping, based on GOST 28147-89. Its purpose is to encrypt and decrypt keys have been generated by key generation mechanism for GOST 28147-89.

For wrapping (**C_WrapKey**), the mechanism first computes MAC from the value of the **CKA_VALUE** attribute of the key that is wrapped and then encrypts in ECB mode the value of the **CKA_VALUE** attribute of the key that is wrapped. The result is 32 bytes of the key that is wrapped and 4 bytes of MAC.

For unwrapping (**C_UnwrapKey**), the mechanism first decrypts in ECB mode the 32 bytes of the key that was wrapped and then computes MAC from the unwrapped key. Then compared together 4 bytes MAC has computed and 4 bytes MAC of the input. If these two MACs do not match the wrapped key is disallowed. The mechanism contributes the result as the **CKA_VALUE** attribute of the unwrapped key.

It has a parameter, a 8-byte MAC initialization vector. This parameter may be omitted then a zero initialization vector is used.

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

Table 7, GOST 28147-89 keys as KEK: Key And Data Length

Function	Key type	Input length	Output length
C_WrapKey	CKK_GOST28147	32 bytes	36 bytes
C_UnwrapKey	CKK_GOST28147	32 bytes	36 bytes

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure are not used.

GOST R 34.11-94

GOST R 34.11-94 is a mechanism for message digesting, following the hash algorithm with 256-bit message digest defined in [GOST R 34.11-94].

6.40.8 Definitions

This section defines the key type “CKK_GOSTR3411” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of domain parameter objects.

Mechanisms:

```
            CKM_GOSTR3411
   CKM_GOSTR3411_HMAC
```

6.40.9 GOST R 34.11-94 domain parameter objects

GOST R 34.11-94 domain parameter objects (object class **CKO_DOMAIN_PARAMETERS,** key type **CKK_GOSTR3411**) hold GOST R 34.11-94 domain parameters.

The following table defines the GOST R 34.11-94 domain parameter object attributes, in addition to the common attributes defined for this object class:

Table 8, GOST R 34.11-94 Domain Parameter Object Attributes

Attribute	Data Type	Meaning
CKA_VALUE[1]	Byte array	DER-encoding of the domain parameters as it was introduced in [4] section 8.2 (type *GostR3411-94-ParamSetParameters*)
CKA_OBJECT_ID[1]	Byte array	DER-encoding of the object identifier indicating the domain parameters

Refer to [PKCS #11-B] Table 15 for footnotes

For any particular token, there is no guarantee that a token supports domain parameters loading up and/or fetching out. Furthermore, applications, that make direct use of domain parameters objects, should take in account that **CKA_VALUE** attribute may be inaccessible.

The following is a sample template for creating a GOST R 34.11-94 domain parameter object:

```
CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
CK_KEY_TYPE keyType = CKK_GOSTR3411;
CK_UTF8CHAR label[] = “A GOST R34.11-94 cryptographic
        parameters object”;
CK_BYTE oid[] = {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02,
        0x02, 0x1e, 0x00};
CK_BYTE value[] = {
0x30,0x64,
0x04,0x40,
0x4e,0x57,0x64,0xd1,0xab,0x8d,0xcb,0xbf,0x94,0x1a,0x7a,0x4d,0x2c,0xd1,0x10,0x10,
0xd6,0xa0,0x57,0x35,0x8d,0x38,0xf2,0xf7,0x0f,0x49,0xd1,0x5a,0xea,0x2f,0x8d,0x94,
0x62,0xee,0x43,0x09,0xb3,0xf4,0xa6,0xa2,0x18,0xc6,0x98,0xe3,0xc1,0x7c,0xe5,0x7e,
0x70,0x6b,0x09,0x66,0xf7,0x02,0x3c,0x8b,0x55,0x95,0xbf,0x28,0x39,0xb3,0x2e,0xcc,
0x04,0x20,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_OBJECT_ID, oid, sizeof(oid)},
    {CKA_VALUE, value, sizeof(value)}
};
```

6.40.10GOST R 34.11-94 digest

GOST R 34.11-94 digest, denoted **CKM_GOSTR3411,** is a mechanism for message digesting based on GOST R 34.11-94 hash algorithm [GOST R 34.11-94].

As a parameter this mechanism utilizes a DER-encoding of the object identifier. A mechanism parameter may be missed then parameters of the object identifier *id-GostR3411-94-CryptoProParamSet* [RFC 4357] (section 11.2) must be used.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 9, GOST R 34.11-94: Data Length

Function	Input length	Digest length
C_Digest	Any	32 bytes

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure are not used.

6.40.11GOST R 34.11-94 HMAC

GOST R 34.11-94 HMAC mechanism, denoted **CKM_GOSTR3411_HMAC**, is a mechanism for signatures and verification. It uses the HMAC construction, based on the GOST R 34.11-94 hash function [GOST R 34.11-94] and core HMAC algorithm [RFC 2104]. The keys it uses are of generic key type **CKK_GENERIC_SECRET** or **CKK_GOST28147**.

To be conformed to GOST R 34.11-94 hash algorithm [GOST R 34.11-94] the block length of core HMAC algorithm is 32 bytes long (see [RFC 2104] section 2, and [RFC 4357] section 3).

As a parameter this mechanism utilizes a DER-encoding of the object identifier. A mechanism parameter may be missed then parameters of the object identifier *id-GostR3411-94-CryptoProParamSet* [RFC 4357] (section 11.2) must be used.

Signatures (MACs) produced by this mechanism are of 32 bytes long.

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

Table 10, GOST R 34.11-94 HMAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	CKK_GENERIC_SECRET or CKK_GOST28147	Any	32 byte
C_Verify	CKK_GENERIC_SECRET or CKK_GOST28147	Any	32 bytes

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure are not used.

6.41 GOST R 34.10-2001

GOST R 34.10-2001 is a mechanism for single- and multiple-part signatures and verification, following the digital signature algorithm defined in [GOST R 34.10-2001].

6.41.1 Definitions

This section defines the key type "CKK_GOSTR3410" for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects and domain parameter objects.

Mechanisms:

```
          CKM_GOSTR3410_KEY_PAIR_GEN
CKM_GOSTR3410
CKM_GOSTR3410_WITH_GOSTR3411
CKM_GOSTR3410
CKM_GOSTR3410_KEY_WRAP
CKM_GOSTR3410_DERIVE
```

6.41.2 GOST R 34.10-2001 public key objects

GOST R 34.10-2001 public key objects (object class **CKO_PUBLIC_KEY,** key type **CKK_GOSTR3410**) hold GOST R 34.10-2001 public keys.

The following table defines the GOST R 34.10-2001 public key object attributes, in addition to the common attributes defined for this object class:

Table 11, GOST R 34.10-2001 Public Key Object Attributes

Attribute	Data Type	Meaning
CKA_VALUE[1,4]	Byte array	64 bytes for public key; 32 bytes for each coordinates X and Y of elliptic curve point P(X, Y) in little endian order
CKA_GOSTR3410PARAMS[1,3]	Byte array	DER-encoding of the object identifier indicating the data object type of GOST R 34.10-2001. When key is used the domain parameter object of key type CKK_GOSTR3410 must be specified with the same attribute CKA_OBJECT_ID
CKA_GOSTR3411PARAMS[1,3,8]	Byte array	DER-encoding of the object identifier indicating the data object type of GOST R 34.11-94. When key is used the domain parameter object of key type CKK_GOSTR3411 must be specified with the same attribute CKA_OBJECT_ID
CKA_GOST28147_PARAMS[8]	Byte array	DER-encoding of the object identifier indicating the data object type of GOST 28147-89. When key is used the domain parameter object of key type CKK_GOST28147 must be specified with the same attribute CKA_OBJECT_ID. The attribute value may be omitted

Refer to [PKCS #11-B] Table 15 for footnotes

The following is a sample template for creating an GOST R 34.10-2001 public key object:

```
CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_GOSTR3410;
CK_UTF8CHAR label[] = "A GOST R34.10-2001 public key
        object";
CK_BYTE gostR3410params_oid[] = {0x06, 0x07, 0x2a, 0x85,
```

```
        0x03, 0x02, 0x02, 0x23, 0x00};
CK_BYTE gostR3411params_oid[] = {0x06, 0x07, 0x2a, 0x85,
        0x03, 0x02, 0x02, 0x1e, 0x00};
CK_BYTE gost28147params_oid[] = {0x06, 0x07, 0x2a, 0x85,
        0x03, 0x02, 0x02, 0x1f, 0x00};
CK_BYTE value[64] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_GOSTR3410PARAMS, gostR3410params_oid,
        sizeof(gostR3410params_oid)},
    {CKA_GOSTR3411PARAMS, gostR3411params_oid,
        sizeof(gostR3411params_oid)},
    {CKA_GOST28147_PARAMS, gost28147params_oid,
        sizeof(gost28147params_oid)},
    {CKA_VALUE, value, sizeof(value)}
};
```

6.41.3 GOST R 34.10-2001 private key objects

GOST R 34.10-2001 private key objects (object class **CKO_PRIVATE_KEY,** key type **CKK_GOSTR3410**) hold GOST R 34.10-2001 private keys.

The following table defines the GOST R 34.10-2001 private key object attributes, in addition to the common attributes defined for this object class:

Table 12, GOST R 34.10-2001 Private Key Object Attributes

Attribute	Data Type	Meaning
CKA_VALUE[1,4,6,7]	Byte array	32 bytes for private key in little endian order
CKA_GOSTR3410PARAMS[1,4,6]	Byte array	DER-encoding of the object identifier indicating the data object type of GOST R 34.10-2001. When key is used the domain parameter object of key type CKK_GOSTR3410 must be specified with the same attribute CKA_OBJECT_ID
CKA_GOSTR3411PARAMS[1,4,6,8]	Byte array	DER-encoding of the object identifier indicating the data object type of GOST R 34.11-94. When key is used the domain parameter object of key type CKK_GOSTR3411 must be specified with the same attribute CKA_OBJECT_ID
CKA_GOST28147_PARAMS4[4,6,8]	Byte array	DER-encoding of the object identifier indicating the data object type of GOST 28147-89. When key is used the domain parameter object of key type CKK_GOST28147 must be specified with the same attribute CKA_OBJECT_ID. The attribute value may be omitted

Refer to [PKCS #11-B] Table 15 for footnotes

Note that when generating an GOST R 34.10-2001 private key, the GOST R 34.10-2001 domain parameters are *not* specified in the key's template. This is because GOST R 34.10-2001 private keys are only generated as part of an GOST R 34.10-2001 key *pair*, and the GOST R 34.10-2001 domain parameters for the pair are specified in the template for the GOST R 34.10-2001 public key.

The following is a sample template for creating an GOST R 34.10-2001 private key object:

```
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_GOSTR3410;
```



```
CK_UTF8CHAR label[] = “A GOST R34.10-2001 private key
        object”;
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE gostR3410params_oid[] = {0x06, 0x07, 0x2a, 0x85,
        0x03, 0x02, 0x02, 0x23, 0x00};
CK_BYTE gostR3411params_oid[] = {0x06, 0x07, 0x2a, 0x85,
        0x03, 0x02, 0x02, 0x1e, 0x00};
CK_BYTE gost28147params_oid[] = {0x06, 0x07, 0x2a, 0x85,
        0x03, 0x02, 0x02, 0x1f, 0x00};
CK_BYTE value[32] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_SUBJECT, subject, sizeof(subject)},
    {CKA_ID, id, sizeof(id)},
    {CKA_SENSITIVE, &true, sizeof(true)},
    {CKA_SIGN, &true, sizeof(true)},
    {CKA_GOSTR3410PARAMS, gostR3410params_oid,
        sizeof(gostR3410params_oid)},
    {CKA_GOSTR3411PARAMS, gostR3411params_oid,
        sizeof(gostR3411params_oid)},
    {CKA_GOST28147_PARAMS, gost28147params_oid,
        sizeof(gost28147params_oid)},
    {CKA_VALUE, value, sizeof(value)}
};
```

6.41.4 GOST R 34.10-2001 domain parameter objects

GOST R 34.10-2001 domain parameter objects (object class **CKO_DOMAIN_PARAMETERS,** key type **CKK_GOSTR3410**) hold GOST R 34.10-2001 domain parameters.

The following table defines the GOST R 34.10-2001 domain parameter object attributes, in addition to the common attributes defined for this object class:

Table 13, GOST R 34.10-2001 Domain Parameter Object Attributes

Attribute	Data Type	Meaning
CKA_VALUE[1]	Byte array	DER-encoding of the domain parameters as it was introduced in [4] section 8.4 (type *GostR3410-2001-ParamSetParameters*)
CKA_OBJECT_ID[1]	Byte array	DER-encoding of the object identifier indicating the domain parameters

Refer to [PKCS #11-B] Table 15 for footnotes

For any particular token, there is no guarantee that a token supports domain parameters loading up and/or fetching out. Furthermore, applications, that make direct use of domain parameters objects, should take in account that **CKA_VALUE** attribute may be inaccessible.

The following is a sample template for creating a GOST R 34.10-2001 domain parameter object:

```
   CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
   CK_KEY_TYPE keyType = CKK_GOSTR3410;
   CK_UTF8CHAR label[] = “A GOST R34.10-2001 cryptographic
           parameters object”;
   CK_BYTE oid[] = {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02,
           0x02, 0x23, 0x00};
   CK_BYTE value[] = {
 0x30,0x81,0x90,
 0x02,0x01,0x07,
 0x02,0x20,
 0x5f,0xbf,0xf4,0x98,0xaa,0x93,0x8c,0xe7,0x39,0xb8,0xe0,0x22,0xfb,0xaf,0xef,0x40,
 0x56,0x3f,0x6e,0x6a,0x34,0x72,0xfc,0x2a,0x51,0x4c,0x0c,0xe9,0xda,0xe2,0x3b,0x7e,
 0x02,0x21,0x00,
 0x80,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x04,0x31,
 0x02,0x21,0x00,
 0x80,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x01,
 0x50,0xfe,0x8a,0x18,0x92,0x97,0x61,0x54,0xc5,0x9c,0xfc,0x19,0x3a,0xcc,0xf5,0xb3,
 0x02,0x01,0x02,
 0x02,0x20,
 0x08,0xe2,0xa8,0xa0,0xe6,0x51,0x47,0xd4,0xbd,0x63,0x16,0x03,0x0e,0x16,0xd1,0x9c,
 0x85,0xc9,0x7f,0x0a,0x9c,0xa2,0x67,0x12,0x2b,0x96,0xab,0xbc,0xea,0x7e,0x8f,0xc8
   };
   CK_BBOOL true = CK_TRUE;
   CK_ATTRIBUTE template[] = {
       {CKA_CLASS, &class, sizeof(class)},
       {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
       {CKA_TOKEN, &true, sizeof(true)},
       {CKA_LABEL, label, sizeof(label)-1},
       {CKA_OBJECT_ID, oid, sizeof(oid)},
       {CKA_VALUE, value, sizeof(value)}
   };
```

6.41.5 GOST R 34.10-2001 mechanism parameters

♦ **CK_GOSTR3410_KEY_WRAP_PARAMS**

CK_GOSTR3410_KEY_WRAP_PARAMS is a structure that provides the parameters to the **CKM_GOSTR3410_KEY_WRAP** mechanism. It is defined as follows:

```
typedef struct CK_GOSTR3410_KEY_WRAP_PARAMS {
        CK_BYTE_PTR      pWrapOID;
        CK_ULONG         ulWrapOIDLen;
        CK_BYTE_PTR      pUKM;
        CK_ULONG         ulUKMLen;
        CK_OBJECT_HANDLE hKey;
} CK_GOSTR3410_KEY_WRAP_PARAMS;
```

The fields of the structure have the following meanings:

pWrapOID	pointer to a data with DER-encoding of the object identifier indicating the data object type of GOST 28147-89. If pointer takes NULL_PTR value in C_WrapKey operation then parameters are specified in object identifier of attribute CKA_GOSTR3411PARAMS must be used. For C_UnwrapKey operation the pointer is not used and must take NULL_PTR value anytime
ulWrapOIDLen	length of data with DER-encoding of the object identifier indicating the data object type of GOST 28147-89
pUKM	pointer to a data with UKM. If pointer takes NULL_PTR value in C_WrapKey operation then random value of UKM will be used. If pointer takes non-NULL_PTR value in C_UnwrapKey operation then the pointer value will be compared with UKM value of wrapped key. If these two values do not match the wrapped key will be rejected
ulUKMLen	length of UKM data. If *pUKM*-pointer is different from NULL_PTR then equal to 8
hKey	key handle. Key handle of a sender for C_WrapKey operation. Key handle of a receiver for C_UnwrapKey operation. When key handle takes CK_INVALID_HANDLE value then an ephemeral (one time) key pair of a sender will be used

- **CK_GOSTR3410_DERIVE_PARAMS**

CK_GOSTR3410_DERIVE_PARAMS is a structure that provides the parameters to the **CKM_GOSTR3410_DERIVE** mechanism. It is defined as follows:

```
typedef struct CK_GOSTR3410_DERIVE_PARAMS {
  CK_EC_KDF_TYPE kdf;
  CK_BYTE_PTR    pPublicData;
  CK_ULONG       ulPublicDataLen;
  CK_BYTE_PTR    pUKM;
  CK_ULONG       ulUKMLen;
} CK_GOSTR3410_DERIVE_PARAMS;
```

The fields of the structure have the following meanings:

kdf additional key diversification algorithm identifier. Possible values are CKD_NULL and CKD_CPDIVERSIFY_KDF. In case of CKD_NULL, result of the key derivation function described in [RFC 4357], section 5.2 is used directly; In case of CKD_CPDIVERSIFY_KDF, the resulting key value is additionaly processed with algorithm from [RFC 4357], section 6.5.

pPublicData[1] pointer to data with public key of a receiver

ulPublicDataLen length of data with public key of a receiver (must be 64)

pUKM pointer to a UKM data

ulUKMLen length of UKM data in bytes (must be 8)

[1] Public key of a receiver is an octet string of 64 bytes long. The public key octets correspond to the concatenation of X and Y coordinates of a point. Any one of them is 32 bytes long and represented in little endian order.

6.41.6 GOST R 34.10-2001 key pair generation

The GOST R 34.10-2001 key pair generation mechanism, denoted **CKM_GOSTR3410_KEY_PAIR_GEN**, is a key pair generation mechanism for GOST R 34.10-2001.

This mechanism does not have a parameter.

The mechanism generates GOST R 34.10-2001 public/private key pairs with particular GOST R 34.10-2001 domain parameters, as specified in the **CKA_GOSTR3410PARAMS**, **CKA_GOSTR3411PARAMS**, and **CKA_GOST28147_PARAMS** attributes of the template for the public key. Note that **CKA_GOST28147_PARAMS** attribute may not be present in the template.

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_VALUE**, and **CKA_GOSTR3410PARAMS**, **CKA_GOSTR3411PARAMS**, **CKA_GOST28147_PARAMS** attributes to the new private key.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure are not used.

6.41.7 GOST R 34.10-2001 without hashing

The GOST R 34.10-2001 without hashing mechanism, denoted **CKM_GOSTR3410**, is a mechanism for single-part signatures and verification for GOST R 34.10-2001. (This mechanism corresponds only to the part of GOST R 34.10-2001 that processes the 32-bytes hash value; it does not compute the hash value.)

This mechanism does not have a parameter.

For the purposes of these mechanisms, a GOST R 34.10-2001 signature is an octet string of 64 bytes long. The signature octets correspond to the concatenation of the GOST R 34.10-2001 values *s* and *r'*, both represented as a 32 bytes octet string in big endian order with the most significant byte first [RFC 4490] section 3.2, and [RFC 4491] section 2.2.2.

The input for the mechanism is an octet string of 32 bytes long with digest has computed by means of GOST R 34.11-94 hash algorithm in the context of signed or should be signed message.

Table 14, GOST R 34.10-2001 without hashing: Key And Data Length

Function	Key type	Input length	Output length
C_Sign[1]	CKK_GOSTR3410	32 bytes	64 bytes
C_Verify[1]	CKK_GOSTR3410	32 bytes	64 bytes

[1] Single-part operations only.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure are not used.

6.41.8 GOST R 34.10-2001 with GOST R 34.11-94

The GOST R 34.10-2001 with GOST R 34.11-94, denoted **CKM_GOSTR3410_WITH_GOSTR3411**, is a mechanism for signatures and verification for GOST R 34.10-2001. This mechanism computes the entire GOST R 34.10-2001 specification, including the hashing with GOST R 34.11-94 hash algorithm.

As a parameter this mechanism utilizes a DER-encoding of the object identifier indicating GOST R 34.11-94 data object type. A mechanism parameter may be missed then parameters are specified in object identifier of attribute **CKA_GOSTR3411PARAMS** must be used.

For the purposes of these mechanisms, a GOST R 34.10-2001 signature is an octet string of 64 bytes long. The signature octets correspond to the concatenation of the GOST R 34.10-2001 values *s* and *r'*, both represented as a 32 bytes octet string in big endian order with the most significant byte first [RFC 4490] section 3.2, and [RFC 4491] section 2.2.2.

The input for the mechanism is signed or should be signed message of any length. Single- and multiple-part signature operations are available.

Table 15, GOST R 34.10-2001 with GOST R 34.11-94: Key And Data Length

Function	Key type	Input length	Output length
C_Sign	CKK_GOSTR3410	Any	64 bytes
C_Verify	CKK_GOSTR3410	Any	64 bytes

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure are not used.

6.41.9 GOST 28147-89 keys wrapping/unwrapping with GOST R 34.10-2001

GOST R 34.10-2001 keys as a KEK (key encryption keys) for encryption GOST 28147 keys, denoted by **CKM_GOSTR3410_KEY_WRAP**, is a mechanism for key wrapping; and key unwrapping, based on GOST R 34.10-2001. Its purpose is to encrypt and decrypt keys have been generated by key generation mechanism for GOST 28147-89. An encryption algorithm from [RFC 4490] (section 5.2) must be used. Encrypted key is a DER-encoded structure of ASN.1 *GostR3410-KeyTransport* type [RFC 4490] section 4.2.

It has a parameter, a **CK_GOSTR3410_KEY_WRAP_PARAMS** structure defined in section 6.41.5.

For unwrapping (**C_UnwrapKey**), the mechanism decrypts the wrapped key, and contributes the result as the **CKA_VALUE** attribute of the new key.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure are not used.

6.41.9.1 Common key derivation with assistance of GOST R 34.10-2001 keys

Common key derivation, denoted **CKM_GOSTR3410_DERIVE,** is a mechanism for key derivation with assistance of GOST R 34.10-2001 private and public keys. The key of the mechanism must be of object class **CKO_DOMAIN_PARAMETERS** and key type **CKK_GOSTR3410**. An algorithm for key derivation from [RFC 4357] (section 5.2) must be used.

The mechanism contributes the result as the **CKA_VALUE** attribute of the new private key. All other attributes must be specified in a template for creating private key object.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure are not used.

A Manifest constants

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

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

```
#define CKK_RSA             0x00000000
#define CKK_DSA             0x00000001
#define CKK_DH              0x00000002
#define CKK_ECDSA           0x00000003
#define CKK_EC              0x00000003
#define CKK_X9_42_DH        0x00000004
#define CKK_GENERIC_SECRET  0x00000010
#define CKK_RC2             0x00000011
#define CKK_RC4             0x00000012
#define CKK_DES             0x00000013
#define CKK_DES2            0x00000014
#define CKK_DES3            0x00000015
#define CKK_CDMF            0x0000001E
#define CKK_AES             0x0000001F
#define CKK_BLOWFISH        0x00000020
#define CKK_TWOFISH         0x00000021
#define CKK_ARIA            0x00000024
#define CKK_CAMELLIA        0x00000025
#define CKK_SEED            0x00000026
#define CKK_MD5_HMAC        0x00000027
#define CKK_SHA_1_HMAC      0x00000028
#define CKK_RIPEMD128_HMAC  0x00000029
#define CKK_RIPEMD160_HMAC  0x0000002A
#define CKK_SHA256_HMAC     0x0000002B
#define CKK_SHA384_HMAC     0x0000002C
#define CKK_SHA512_HMAC     0x0000002D
#define CKK_SHA224_HMAC     0x0000002E
#define CKK_GOSTR3410       0x00000030
#define CKK_GOSTR3411       0x00000031
#define CKK_GOST28147       0x00000032

#define CKK_VENDOR_DEFINED  0x80000000

#define CKC_X_509           0x00000000
#define CKC_X_509_ATTR_CERT 0x00000001
#define CKC_WTLS            0x00000002
#define CKC_VENDOR_DEFINED  0x80000000

#define CKD_NULL                        0x00000001
#define CKD_SHA1_KDF                    0x00000002
#define CKD_SHA1_KDF_ASN1               0x00000003
#define CKD_SHA1_KDF_CONCATENATE        0x00000004
#define CKD_SHA224_KDF                  0x00000005
#define CKD_SHA256_KDF                  0x00000006
#define CKD_SHA384_KDF                  0x00000007
#define CKD_SHA512_KDF                  0x00000008
#define CKD_CPDIVERSIFY_KDF             0x00000009


#define CKM_RSA_PKCS_KEY_PAIR_GEN           0x00000000
#define CKM_RSA_PKCS                        0x00000001
```

```
#define CKM_RSA_9796                        0x00000002
#define CKM_RSA_X_509                       0x00000003
#define CKM_SHA1_RSA_PKCS                   0x00000006
#define CKM_RSA_PKCS_OAEP                   0x00000009
#define CKM_RSA_X9_31_KEY_PAIR_GEN          0x0000000A
#define CKM_RSA_X9_31                       0x0000000B
#define CKM_SHA1_RSA_X9_31                  0x0000000C
#define CKM_RSA_PKCS_PSS                    0x0000000D
#define CKM_SHA1_RSA_PKCS_PSS               0x0000000E
#define CKM_DSA_KEY_PAIR_GEN                0x00000010
#define CKM_DSA                             0x00000011
#define CKM_DSA_SHA1                        0x00000012
#define CKM_DH_PKCS_KEY_PAIR_GEN            0x00000020
#define CKM_DH_PKCS_DERIVE                  0x00000021
#define CKM_X9_42_DH_KEY_PAIR_GEN           0x00000030
#define CKM_X9_42_DH_DERIVE                 0x00000031
#define CKM_X9_42_DH_HYBRID_DERIVE          0x00000032
#define CKM_X9_42_MQV_DERIVE                0x00000033
#define CKM_SHA256_RSA_PKCS                 0x00000040
#define CKM_SHA384_RSA_PKCS                 0x00000041
#define CKM_SHA512_RSA_PKCS                 0x00000042
#define CKM_SHA256_RSA_PKCS_PSS             0x00000043
#define CKM_SHA384_RSA_PKCS_PSS             0x00000044
#define CKM_SHA512_RSA_PKCS_PSS             0x00000045
#define CKM_RC2_KEY_GEN                     0x00000100
#define CKM_DES2_KEY_GEN                    0x00000130
#define CKM_DES3_KEY_GEN                    0x00000131
#define CKM_DES3_ECB                        0x00000132
#define CKM_DES3_CBC                        0x00000133
#define CKM_DES3_MAC                        0x00000134
#define CKM_DES3_MAC_GENERAL                0x00000135
#define CKM_DES3_CBC_PAD                    0x00000136
#define CKM_DES3_CMAC_GENERAL               0x00000137
#define CKM_DES3_CMAC                       0x00000138
#define CKM_CDMF_KEY_GEN                    0x00000140
#define CKM_CDMF_ECB                        0x00000141
#define CKM_CDMF_CBC                        0x00000142
#define CKM_CDMF_MAC                        0x00000143
#define CKM_CDMF_MAC_GENERAL                0x00000144
#define CKM_CDMF_CBC_PAD                    0x00000145
#define CKM_DES_OFB64                       0x00000150
#define CKM_DES_OFB8                        0x00000151
#define CKM_DES_CFB64                       0x00000152
#define CKM_DES_CFB8                        0x00000153
#define CKM_SHA_1                           0x00000220
#define CKM_SHA_1_HMAC                      0x00000221
#define CKM_SHA_1_HMAC_GENERAL              0x00000222
#define CKM_SHA256                          0x00000250
#define CKM_SHA256_HMAC                     0x00000251
#define CKM_SHA256_HMAC_GENERAL             0x00000252
#define CKM_SHA384                          0x00000260
#define CKM_SHA384_HMAC                     0x00000261
#define CKM_SHA384_HMAC_GENERAL             0x00000262
#define CKM_SHA512                          0x00000270
#define CKM_SHA512_HMAC                     0x00000271
#define CKM_SHA512_HMAC_GENERAL             0x00000272
#define CKM_GENERIC_SECRET_KEY_GEN          0x00000350
#define CKM_CONCATENATE_BASE_AND_KEY        0x00000360
#define CKM_CONCATENATE_BASE_AND_DATA       0x00000362
#define CKM_CONCATENATE_DATA_AND_BASE       0x00000363
```



```
#define CKM_XOR_BASE_AND_DATA                0x00000364
#define CKM_EXTRACT_KEY_FROM_KEY             0x00000365
#define CKM_SSL3_PRE_MASTER_KEY_GEN          0x00000370
#define CKM_SSL3_MASTER_KEY_DERIVE           0x00000371
#define CKM_SSL3_KEY_AND_MAC_DERIVE          0x00000372
#define CKM_SSL3_MASTER_KEY_DERIVE_DH        0x00000373
#define CKM_TLS_PRE_MASTER_KEY_GEN           0x00000374
#define CKM_TLS_MASTER_KEY_DERIVE            0x00000375
#define CKM_TLS_KEY_AND_MAC_DERIVE           0x00000376
#define CKM_TLS_MASTER_KEY_DERIVE_DH         0x00000377
#define CKM_TLS_PRF                          0x00000378
#define CKM_SSL3_MD5_MAC                     0x00000380
#define CKM_SSL3_SHA1_MAC                    0x00000381
#define CKM_MD5_KEY_DERIVATION               0x00000390
#define CKM_MD2_KEY_DERIVATION               0x00000391
#define CKM_SHA1_KEY_DERIVATION              0x00000392
#define CKM_SHA256_KEY_DERIVATION            0x00000393
#define CKM_SHA384_KEY_DERIVATION            0x00000394
#define CKM_SHA512_KEY_DERIVATION            0x00000395
#define CKM_PBE_SHA1_DES3_EDE_CBC            0x000003A8
#define CKM_PBE_SHA1_DES2_EDE_CBC            0x000003A9
#define CKM_PBE_SHA1_RC2_128_CBC             0x000003AA
#define CKM_PBE_SHA1_RC2_40_CBC              0x000003AB
#define CKM_PKCS5_PBKD2                      0x000003B0
#define CKM_PBA_SHA1_WITH_SHA1_HMAC          0x000003C0
#define CKM_WTLS_PRE_MASTER_KEY_GEN          0x000003D0
#define CKM_WTLS_MASTER_KEY_DERIVE           0x000003D1
#define CKM_WTLS_MASTER_KEY_DERVIE_DH_ECC    0x000003D2
#define CKM_WTLS_PRF                         0x000003D3
#define CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE   0x000003D4
#define CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE   0x000003D5
#define CKM_KEY_WRAP_LYNKS                   0x00000400
#define CKM_KEY_WRAP_SET_OAEP                0x00000401
#define CKM_CMS_SIG                          0x00000500
#define CKM_ECDSA_KEY_PAIR_GEN               0x00001040
#define CKM_EC_KEY_PAIR_GEN                  0x00001040
#define CKM_ECDSA                            0x00001041
#define CKM_ECDSA_SHA1                       0x00001042
#define CKM_ECDH1_DERIVE                     0x00001050
#define CKM_ECDH1_COFACTOR_DERIVE            0x00001051
#define CKM_ECMQV_DERIVE                     0x00001052
#define CKM_AES_KEY_GEN                      0x00001080
#define CKM_AES_ECB                          0x00001081
#define CKM_AES_CBC                          0x00001082
#define CKM_AES_MAC                          0x00001083
#define CKM_AES_MAC_GENERAL                  0x00001084
#define CKM_AES_CBC_PAD                      0x00001085
#define CKM_AES_CMAC_GENERAL                 0x00001089
#define CKM_AES_CMAC                         0x0000108A
#define CKM_BLOWFISH_KEY_GEN                 0x00001090
#define CKM_BLOWFISH_CBC                     0x00001091
#define CKM_TWOFISH_KEY_GEN                  0x00001092
#define CKM_TWOFISH_CBC                      0x00001093
#define CKM_DES_ECB_ENCRYPT_DATA             0x00001100
#define CKM_DES_CBC_ENCRYPT_DATA             0x00001101
#define CKM_DES3_ECB_ENCRYPT_DATA            0x00001102
#define CKM_DES3_CBC_ENCRYPT_DATA            0x00001103
#define CKM_AES_ECB_ENCRYPT_DATA             0x00001104
#define CKM_AES_CBC_ENCRYPT_DATA             0x00001105
#define CKM_DSA_PARAMETER_GEN                0x00002000
```

```
#define CKM_DH_PKCS_PARAMETER_GEN           0x00002001
#define CKM_X9_42_DH_PARAMETER_GEN          0x00002002
```

```
#define CKM_SHA224                          0x00000255
#define CKM_SHA224_HMAC                     0x00000256
#define CKM_SHA224_HMAC_GENERAL             0x00000257
#define CKM_SHA224_RSA_PKCS                 0x00000046
#define CKM_SHA224_RSA_PKCS_PSS             0x00000047
#define CKM_SHA224_KEY_DERIVATION           0x00000396
#define CKG_MGF1_SHA224                     0x00000005
#define CKM_AES_CTR                         0x00001086
#define CKM_AES_CTS                         0x00001089
#define CKM_KIP_DERIVE                      0x00000510
#define CKM_KIP_WRAP                        0x00000511
#define CKM_KIP_MAC                         0x00000512

#define CKM_CAMELLIA_KEY_GEN                0x00000550
#define CKM_CAMELLIA_ECB                    0x00000551
#define CKM_CAMELLIA_CBC                    0x00000552
#define CKM_CAMELLIA_MAC                    0x00000553
#define CKM_CAMELLIA_MAC_GENERAL            0x00000554
#define CKM_CAMELLIA_CBC_PAD                0x00000555
#define CKM_CAMELLIA_ECB_ENCRYPT_DATA       0x00000556
#define CKM_CAMELLIA_CBC_ENCRYPT_DATA       0x00000557
#define CKM_ARIA_KEY_GEN                    0x00000560
#define CKM_ARIA_ECB                        0x00000561
#define CKM_ARIA_CBC                        0x00000562
#define CKM_ARIA_MAC                        0x00000563
#define CKM_ARIA_MAC_GENERAL                0x00000564
#define CKM_ARIA_CBC_PAD                    0x00000565
#define CKM_ARIA_ECB_ENCRYPT_DATA           0x00000566
#define CKM_ARIA_CBC_ENCRYPT_DATA           0x00000567

#define CKM_SEED_KEY_GEN                    0x00000650
#define CKM_SEED_ECB                        0x00000651
#define CKM_SEED_CBC                        0x00000652
#define CKM_SEED_MAC                        0x00000653
#define CKM_SEED_MAC_GENERAL                0x00000654
#define CKM_SEED_CBC_PAD                    0x00000655
#define CKM_SEED_ECB_ENCRYPT_DATA           0x00000656
#define CKM_SEED_CBC_ENCRYPT_DATA           0x00000657
#define CKM_AES_GCM                         0x00001087
#define CKM_AES_CCM                         0x00001088
#define CKM_AES_OFB                         0x00002104
#define CKM_AES_CFB64                       0x00002105
#define CKM_AES_CFB8                        0x00002106
#define CKM_AES_CFB128                      0x00002107
#define CKM_BLOWFISH_CBC_PAD                0x00001094
#define CKM_TWOFISH_CBC_PAD                 0x00001095

#define CKM_AES_KEY_WRAP                    0x00001090
#define CKM_AES_KEY_WRAP_PAD                0x00001091

#define CKM_RSA_PKCS_TPM_1_1                0x00004001
#define CKM_RSA_PKCS_OAEP_TPM_1_1           0x00004002

#define CKM_GOSTR3410_KEY_PAIR_GEN          0x00001200
#define CKM_GOSTR3410                       0x00001201
#define CKM_GOSTR3410_WITH_GOSTR3411        0x00001202
#define CKM_GOSTR3410_KEY_WRAP              0x00001203
#define CKM_GOSTR3410_DERIVE                0x00001204
#define CKM_GOSTR3411                       0x00001210
#define CKM_GOSTR3411_HMAC                  0x00001211
```

```
#define CKM_GOST28147_KEY_GEN              0x00001220
#define CKM_GOST28147_ECB                  0x00001221
#define CKM_GOST28147                      0x00001222
#define CKM_GOST28147_MAC                  0x00001223
#define CKM_GOST28147_KEY_WRAP             0x00001224
#define CKA_GOSTR3410_PARAMS               0x00000250
#define CKA_GOSTR3411_PARAMS               0x00000251
#define CKA_GOST28147_PARAMS               0x00000252


#define CKM_VENDOR_DEFINED                 0x80000000
```

A.1 OTP Definitions

Note: A C or C++ source file in a Cryptoki application or library can define all the types, mechanisms, and other constants described here by including the header file otp-pkcs11.h. When including the otp-pkcs11.h header file, it should be preceded by an inclusion of the top-level Cryptoki header file pkcs11.h, and the source file must also specify the preprocessor directives indicated in Section 8 of [PKCS #11-B].

A.2 Object classes

```
#define CKO_OTP_KEY                    0x00000008
```

A.3 Key types

```
#define CKK_SECURID                    0x00000022
#define CKK_HOTP                       0x00000023
#define CKK_ACTI                       0x00000024
```

A.4 Mechanisms

```
#define CKM_SECURID_KEY_GEN            0x00000280
#define CKM_SECURID                    0x00000282

#define CKM_HOTP_KEY_GEN               0x00000290
#define CKM_HOTP                       0x00000291

#define CKM_ACTI_KEY_GEN               0x000002A0
#define CKM_ACTI                       0x000002A1
```

A.5 Attributes

```
#define CKA_OTP_FORMAT                  0x00000220
#define CKA_OTP_LENGTH                  0x00000221
#define CKA_OTP_TIME_INTERVAL           0x00000222
#define CKA_OTP_USER_FRIENDLY_MODE      0x00000223
#define CKA_OTP_CHALLENGE_REQUIREMENT   0x00000224
#define CKA_OTP_TIME_REQUIREMENT        0x00000225
#define CKA_OTP_COUNTER_REQUIREMENT     0x00000226
#define CKA_OTP_PIN_REQUIREMENT         0x00000227
```



```
#define CKA_OTP_USER_IDENTIFIER       0x0000022A
#define CKA_OTP_SERVICE_IDENTIFIER    0x0000022B
#define CKA_OTP_SERVICE_LOGO          0x0000022C
#define CKA_OTP_SERVICE_LOGO_TYPE     0x0000022D
#define CKA_OTP_COUNTER               0x0000022E
#define CKA_OTP_TIME                  0x0000022F
```

A.6 Attribute constants

```
#define CK_OTP_FORMAT_DECIMAL        0
#define CK_OTP_FORMAT_HEXADECIMAL    1
#define CK_OTP_FORMAT_ALPHANUMERIC   2
#define CK_OTP_FORMAT_BINARY         3

#define CK_OTP_PARAM_IGNORED         0
#define CK_OTP_PARAM_OPTIONAL        1
#define CK_OTP_PARAM_MANDATORY       2
```

A.7 Other constants

```
#define CK_OTP_VALUE                 0
#define CK_OTP_PIN                   1
#define CK_OTP_CHALLENGE             2
#define CK_OTP_TIME                  3
#define CK_OTP_COUNTER               4
#define CK_OTP_FLAGS                 5
#define CK_OTP_OUTPUT_LENGTH         6
#define CK_OTP_FORMAT                7

#define CKF_NEXT_OTP                 0x00000001
#define CKF_EXCLUDE_TIME             0x00000002
#define CKF_EXCLUDE_COUNTER          0x00000004
#define CKF_EXCLUDE_CHALLENGE        0x00000008
#define CKF_EXCLUDE_PIN              0x00000010
#define CKF_USER_FRIENDLY_OTP        0x00000020
```

A.8 Notifications

```
#define CKN_OTP_CHANGED              1
```

A.9 Return values

```
#define CKR_NEW_PIN_MODE   0x000001B0
#define CKR_NEXT_OTP       0x000001B1
```

B. OTP Example code

B.1 Disclaimer concerning sample code

For the sake of brevity, sample code presented herein is somewhat incomplete. In particular, initial steps needed to create a session with a cryptographic token are not shown, and the error handling is simplified.

B.2 OTP retrieval

The following sample code snippet illustrates the retrieval of an OTP value from an OTP token using the **C_Sign** function. The sample demonstrates the generality of the approach described herein and does not include any OTP mechanism-specific knowledge.

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;
CK_RV rv;
CK_SLOT_ID slotId;
CK_OBJECT_CLASS class = CKO_OTP_KEY;
CK_ATTRIBUTE template[] = {
  {CKA_CLASS, &class, sizeof(class)} };
CK_UTF8CHAR time[] = {...};
/* UTC time value for OTP, or NULL */
CK_UTF8CHAR pin[] = {...};
/* User PIN, or NULL */
CK_BYTE counter[] = {...};
/* Counter value, or NULL */
CK_BYTE challenge[] = {...};
/* Challenge, or NULL */
CK_MECHANISM_TYPE_PTR allowedMechanisms = NULL_PTR;
CK_MECHANISM_INFO mechanismInfo;
CK_MECHANISM mechanism;
CK_ULONG i, ulOTPLen, ulKeyCount, ulChalReq, ulPINReq,
      ulTimeReq, ulCounterReq;
CK_ATTRIBUTE mechanisms[] = { {CKA_ALLOWED_MECHANISMS,
      NULL_PTR, 0} };
CK_ATTRIBUTE attributes[] = {
  {CKA_OTP_CHALLENGE_REQUIREMENT, &ulChalReq,
      sizeof(ulChalReq)},
  {CKA_OTP_PIN_REQUIREMENT, &ulPINReq,
      sizeof(ulPINReq)},
  {CKA_OTP_COUNTER_REQUIREMENT, &ulCounterReq,
      sizeof(ulCounterReq)},
  {CKA_OTP_TIME_REQUIREMENT, &ulTimeReq,
      sizeof(ulTimeReq)} };

CK_OTP_PARAM param[4];
CK_OTP_PARAMS params;
CK_BYTE *pOTP; /* Storage for OTP result */
```



```
do {

  /* N.B.: Minimal error and memory handling in this
     sample code. */

  /* Find first OTP key on the token. */
  if ((rv = C_FindObjectsInit(hSession, template, 1))
      != CKR_OK) {
    break;
  };
  if ((rv = C_FindObjects(hSession, &hKey, 1,
      &ulKeyCount)) != CKR_OK) {
    break;
  };
  if (ulKeyCount == 0) {
    /* No OTP key found */
    break;
  }
  rv = C_FindObjectsFinal(hSession);

  /* Find a suitable OTP mechanism. */
  if ((rv = C_GetAttributeValue(hSession, hKey,
      mechanisms, 1)) != CKR_OK) {
    break;
  };

  if ((allowedMechanisms = (CK_MECHANISM_TYPE_PTR)
      malloc(mechanisms[0].ulValueLen)) == 0) {
    break;
  };

  mechanisms[0].pValue = allowedMechanisms;
  if ((rv = C_GetAttributeValue(hSession, hKey,
      mechanisms, 1)) != CKR_OK) {
    break;
  };

  for (i = 0; i < mechanisms[0].ulValueLen/
      sizeof(CK_MECHANISM_TYPE); ++i) {
    if ((rv = C_GetMechanismInfo(slotId,
      allowedMechanisms[i], &mechanismInfo)) == CKR_OK)
      {
      if (mechanismInfo.flags & CKF_SIGN) {
        break;
      }
    }
  }

  if (i == mechanisms[0].ulValueLen) {
```



```
  break;
}

mechanism.mechanism = allowedMechanisms[i];
free(allowedMechanisms);

/* Set required mechanism parameters based on
   the key attributes. */
if ((rv = C_GetAttributeValue(hSession, hKey,
     attributes, sizeof(attributes) /
     sizeof(attributes[0]))) != CKR_OK) {
  break;
}

i = 0;
if (ulPINReq == CK_OTP_PARAM_MANDATORY) {
  /* PIN value needed. */
  param[i].type = CK_OTP_PIN;
  param[i].pValue = pin;
  param[i++].ulValueLen = sizeof(pin) - 1;
}
if (ulChalReq == CK_OTP_PARAM_MANDATORY) {
  /* Challenge neded. */
  param[i].type = CK_OTP_CHALLENGE;
  param[i].pValue = challenge;
  param[i++].ulValueLen = sizeof(challenge);
}
if (ulTimeReq == CK_OTP_PARAM_MANDATORY) {
  /* Time needed (would not normally be
     the case if token has its own clock). */
  param[i].type = CK_OTP_TIME;
  param[i].pValue = time;
  param[i++].ulValueLen = sizeof(time) -1;
}
if (ulCounterReq == CK_OTP_PARAM_MANDATORY) {
  /* Counter value needed (would not normally
     be the case if token has its own counter.*/
  param[i].type = CK_OTP_COUNTER;
  param[i].pValue = counter;
  param[i++].ulValueLen = sizeof(counter);
}

params.pParams = param;
params.ulCount = i;

mechanism.pParameter = &params;
mechanism.ulParameterLen = sizeof(params);

/* Sign to get the OTP value. */
if ((rv = C_SignInit(hSession, &mechanism, hKey))
```

```
        != CKR_OK) {
        break;
      }

      /* Get the buffer length needed for the OTP Value
         and any associated data. */
      if ((rv = C_Sign(hSession, NULL_PTR, 0, NULL_PTR,
          &ulOTPLen)) != CKR_OK) {
        break;
      };

      if ((pOTP = malloc(ulOTPLen)) == NULL_PTR) {
        break;
      };

      /* Get the actual OTP value and any
         associated data. */
      if ((rv = C_Sign(hSession, NULL_PTR, 0, pOTP,
               &ulOTPLen)) != CKR_OK) {
        break;
      }

      /* Traverse the returned pOTP here. The actual
         OTP value is in CK_OTP_VALUE in pOTP. */

    } while (0);
```

B.3 User-friendly mode OTP token

This sample demonstrates an application retrieving a user-friendly OTP value. The code is the same as in B.1 except for the following:

```
  /* Add these variable declarations */

    CK_FLAGS flags = CKF_USER_FRIENDLY_OTP;
    CK_BBOOL bUserFriendlyMode;
    CK_ULONG ulFormat;

  /* Replace the declaration of the "attributes" and the
     "param" variables with: */

    CK_ATTRIBUTE attributes[] = {
      {CKA_OTP_CHALLENGE_REQUIREMENT, &ulChalReq,
      sizeof(ulChalReq)},
      {CKA_OTP_PIN_REQUIREMENT, &ulPINReq,
      sizeof(ulPINReq)},
      {CKA_OTP_COUNTER_REQUIREMENT, &ulCounterReq,
      sizeof(ulCounterReq)},
      {CKA_OTP_TIME_REQUIREMENT, &ulTimeReq,
      sizeof(ulTimeReq)},
```

```
      {CKA_OTP_USER_FRIENDLY_MODE, &bUserFriendlyMode,
      sizeof(bUserFriendlyMode)},
      {CKA_OTP_FORMAT, &ulFormat,
      sizeof(ulFormat)}
    };

      CK_OTP_PARAM param[5];

  /* Replace the assignment of the "pParam" component
     of the "params" variable with: */

      if (bUserFriendlyMode == CK_TRUE) {
        /* Token supports user-friendly OTPs */
        param[i].type = CK_OTP_FLAGS;
        param[i].pValue = &flags;
        param[i++].ulValueLen = sizeof(CK_FLAGS);
      } else if (ulFormat == CK_OTP_FORMAT_BINARY) {
        /* Some kind of error since a user-friendly
           OTP cannot be returned to an application
           that needs it. */
        break;
      };

      params.pParams = param;

  /* Further processing is as in B.1. */
```

B.4 OTP verification

The following sample code snippet illustrates the verification of an OTP value from an RSA SecurID token, using the **C_Verify** function. The desired UTC time, if a time is specified, is supplied in the CK_OTP_PARAMS structure, as is the user's PIN.

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;
CK_UTF8CHAR time[] = {...};
/* UTC time value for OTP, or NULL */
CK_UTF8CHAR pin[] = {...};
/* User PIN or NULL (if collected by library) */
CK_OTP_PARAM param[] = {
  {CK_OTP_TIME, time, sizeof(time)-1},
  {CK_OTP_PIN, pin, sizeof(pin)-1}
};
CK_OTP_PARAMS params = {param, 2};
CK_MECHANISM mechanism = {CKM_SECURID, &params,
        sizeof(params)};
CK_ULONG ulKeyCount;
CK_RV rv;
CK_BYTE OTP[] = {...};     /* Supplied OTP value. */
```



```
   CK_ULONG ulOTPLen = strlen((CK_CHAR_PTR)OTP);
   CK_OBJECT_CLASS class = CKO_OTP_KEY;
   CK_KEY_TYPE keyType = CKK_SECURID;

   CK_ATTRIBUTE template[] = {
     {CKA_CLASS, &class, sizeof(class)},
     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   };

   /* Find the RSA SecurID key on the token. */
   rv = C_FindObjectsInit(hSession, template, 2);
   if (rv == CKR_OK) {
      rv = C_FindObjects(hSession, &hKey, 1, &ulKeyCount);
      rv = C_FindObjectsFinal(hSession);
   }

   if ((rv != CKR_OK) || (ulKeyCount == 0)) {
      printf(" \nError: unable to find RSA SecurID key on
           token.\n");
      return(rv);
}

   rv = C_VerifyInit(hSession, &mechanism, hKey);
   if (rv == CKR_OK) {
     ulOTPLen = sizeof(OTP);
     rv = C_Verify(hSession, NULL_PTR, 0, OTP, ulOTPLen);
   }

   switch(rv) {
       case CKR_OK:
           printf("\nSupplied OTP value verified.\n");
           break;

       case CKR_SIGNATURE_INVALID:
           printf("\nSupplied OTP value not verified.\n");
           break;

       default:
           printf("\nError:Unable to verify OTP value.\n");
           break;
   }

   return(rv);
```

C. Using PKCS #11 with CT-KIP

A suggested procedure to perform CT-KIP with a cryptographic token through the PKCS #11 interface using the mechanisms defined herein is as follows:

a. On the client side,

I. The client selects a suitable slot and token (e.g. through use of the <TokenID> or the <PlatformInfo> element of the CT-KIP trigger message).

II. Optionally, a nonce *R* is generated, e.g. by calling **C_SeedRandom** and **C_GenerateRandom**.

III. The client sends its first message to the server, potentially including the nonce *R*.

b. On the server side,

I. A nonce R_S is generated, e.g. by calling **C_SeedRandom** and **C_GenerateRandom**.

II. If the server needs to authenticate its first CT-KIP message, and use of **CKM_KIP_MAC** has been negotiated, it calls **C_SignInit** with **CKM_KIP_MAC** as the mechanism followed by a call to **C_Sign**. In the call to **C_SignInit**, K_{AUTH} (see 0) shall be the signature key, the *hKey* parameter in the **CK_KIP_PARAMS** structure shall be set to NULL_PTR, the *pSeed* parameter of the **CT_KIP_PARAMS** structure shall also be set to NULL_PTR and the *ulSeedLen* parameter shall be set to zero. In the call to **C_Sign**, the *pData* parameter shall be set to point to (the concatenation of the nonce *R*, if received, and) the nonce R_S (see 0 for a definition of the variables), and the *ulDataLen* parameter shall hold the length of the (concatenated) string. The desired length of the MAC shall be specified through the *pulSignatureLen* parameter as usual.

III. The server sends its first message to the client, including R_S, the server's public key *K* (or an identifier for a shared secret key *K*), and optionally the MAC.

c. On the client side,

I. If a MAC was received, it is verified. If the MAC does not verify, or was required but not received, the protocol session ends with a failure.

II. If the MAC verified, or was not required and not present, a generic secret key, R_C, is generated by calling **C_GenerateKey** with the **CKM_GENERIC_SECRET_KEY_GEN** mechanism. The *pTemplate* attribute shall have **CKA_EXTRACTABLE** and **CKA_SENSITIVE** set to **CK_TRUE**, and should have **CKA_ALLOWED_MECHANISMS** set to **CKM_KIP_DERIVE** only.

III. The generic secret key R_C is wrapped by calling **C_WrapKey**. If the server's public key is used to wrap R_C, and that key is temporary only, then the **CKA_EXTRACTABLE** attribute of R_C shall be set to CK_FALSE once R_C has been wrapped and the server's public key is to be destroyed. If a shared secret key is used to wrap R_C, and use of the CT-KIP key wrapping algorithm was negotiated, then the **CKM_KIP_WRAP** mechanism shall be used. The *hKey* handle in the **CK_KIP_PARAMS** structure shall be set to NULL_PTR. The *pSeed* parameter in the **CK_KIP_PARAMS** structure shall point to the nonce R_S provided by the CT-KIP server, and the *ulSeedLen* parameter shall indicate the length of R_S. The *hWrappingKey* parameter in the call to **C_WrapKey** shall be set to refer to the wrapping key.

IV. The client sends its second message to the server, including the wrapped generic secret key R_C.

d. On the server side,

I. Once the wrapped generic secret key R_C has been received, the server calls **C_UnwrapKey**. If use of the CT-KIP key wrapping algorithm was negotiated, then **CKM_KIP_WRAP** shall be used to unwrap R_C. When calling **C_UnwrapKey**, the **CK_KIP_PARAMS** structure shall be set as described in c.III above. The *hUnwrappingKey* function parameter shall refer to the shared secret key and the *pTemplate* function parameter shall have **CKA_SENSITIVE** set to **CK_TRUE**, **CKA_KEY_TYPE** set to **CKK_GENERIC_SECRET** and should have **CKA_ALLOWED_MECHANISMS** set to **CKM_KIP_DERIVE** only. This will return a handle to the generic secret key R_C.

II. A token key, K_{TOKEN}, is derived from R_C by calling **C_DeriveKey** with the **CKM_KIP_DERIVE** mechanism, using R_C as *hBaseKey*. The *hKey* handle in the **CK_KIP_PARAMS** structure shall refer either to the public key supplied by the CT-KIP server, or alternatively, the shared secret key indicated by the server. The *pSeed* parameter shall point to the nonce R_S provided by the CT-KIP server, and the *ulSeedLen* parameter shall indicate the length of R_S. The *pTemplate* attribute shall be set in accordance with local policy and as negotiated in the protocol. This will return a handle to the token key, K_{TOKEN}.

III. For the server's last CT-KIP message to the client, if use of the CT-KIP MAC algorithm has been negotiated, then the MAC is calculated by calling **C_SignInit** with the **CKM_KIP_MAC** mechanism followed by a call to **C_Sign**. In the call to **C_SignInit**, K_{AUTH} (see 0) shall be the signature key, the *hKey* parameter in the **CK_KIP_PARAMS** structure shall be a handle to the generic secret key R_C, the *pSeed* parameter of the **CT_KIP_PARAMS** structure shall be set to NULL_PTR, and the *ulSeedLen* parameter shall be set to zero. In the call to **C_Sign**, the *pData* parameter shall be set to NULL_PTR and the *ulDataLen* parameter shall be set to 0. The desired length of the MAC shall be specified through the *pulSignatureLen* parameter as usual.

IV. The server sends its second message to the client, including the MAC.

e. On the client side,

I. The MAC is verified in a reciprocal fashion as it was generated by the server. If use of the **CKM_KIP_MAC** mechanism was negotiated, then in the call to **C_VerifyInit,** the *hKey* parameter in the **CK_KIP_PARAMS** structure shall refer to R_C, the *pSeed* parameter shall be set to NULL_PTR, and *ulSeedLen* shall be set to 0. The *hKey* parameter of **C_VerifyInit** shall refer to K_{AUTH}. In the call to **C_Verify**, *pData* shall be set to NULL_PTR, *ulDataLen* to 0, *pSignature* to the MAC value received from the server, and *ulSignatureLen* to the length of the MAC. If the MAC does not verify the protocol session ends with a failure.

II. A token key, K_{TOKEN}, is derived from R_C by calling **C_DeriveKey** with the **CKM_KIP_DERIVE** mechanism, using R_C as *hBaseKey*. The *hKey* handle in the **CK_KIP_PARAMS** structure shall be set to NULL_PTR as token policy must dictate use of the same key as was used to wrap R_C. The *pSeed* parameter shall point to the nonce R_S provided by the CT-KIP server, and the *ulSeedLen* parameter shall indicate the length of R_S. The *pTemplate* attribute shall be set in accordance with local policy and as negotiated and expressed in the protocol. In particular, the value of the **<KeyID>** element in the server's response message may be used as **CKA_ID.** The call to **C_DeriveKey** will, if successful, return a handle to K_{TOKEN}. [††]

[††] When K_{AUTH} is the newly generated K_{TOKEN}, the client will need to call **C_DeriveKey** before calling **C_VerifyInit** and **C_Verify** (since the *hKey* parameter of **C_VerifyInit** shall refer to K_{TOKEN}). In this case, the token should not allow K_{TOKEN} to be used for any other operation than the verification of the MAC value until the MAC has successfully been verified.

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 Mechanisms v2.30.

