RSA Laboratories. A Division of RSA Data Security

PKCS #1 v2.0: RSA Cryptography Standard

RSA Laboratories

October 1, 1998

Table of Contents

1. Introduction

This document provides recommendations for the implementation of public-key cryptography based on the RSA algorithm [18], covering the following aspects:

- cryptographic primitives
- encryption schemes
- signature schemes with appendix
- ASN.1 syntax for representing keys and for identifying the schemes

The recommendations are intended for general application within computer and communications systems, and as such include a fair amount of flexibility. It is expected that application standards based on these specifications may include additional constraints. The recommendations are intended to be compatible with draft standards currently being developed by the ANSI X9F1 [1] and IEEE P1363 working groups [14].

This document supersedes PKCS #1 version 1.5 [20].

Editor's note. It is expected that subsequent versions of PKCS #1 may cover other aspects of the RSA algorithm such as key size, key generation, key validation, and signature schemes with message recovery.

1.1 Overview

The organization of this document is as follows:

- Section 1 is an introduction.
- Section 2 defines some notation used in this document.
- Section 3 defines the RSA public and private key types.
- Sections 4 and 5 define several primitives, or basic mathematical operations. Data conversion primitives are in Section 4, and cryptographic primitives (encryption-decryption, signature-verification) are in Section 5.
- Section 6, 7 and 8 deal with the encryption and signature schemes in this document. Section 6 gives an overview. Section 7 defines an OAEP-based [2] encryption scheme along with the method found in PKCS #1 v1.5. Section 8 defines a signature scheme with appendix; the method is identical to that of PKCS #1 v1.5.
- Section 9 defines the encoding methods for the encryption and signature schemes in Sections 7 and 8.
- Section 10 defines the hash functions and the mask generation function used in this document.
- Section 11 defines the ASN.1 syntax for the keys defined in Section 3 and the schemes in Sections 7 and 8.
- Section 12 outlines the revision history of PKCS #1.
- Section 13 contains references to other publications and standards.

2. Notation

(n, e)	RSA public key
c	ciphertext representative, an integer between 0 and n-1
C	ciphertext, an octet string
d	private exponent
dP	p's exponent, a positive integer such that: $e \cdot dP \equiv 1 \pmod{(p-1)}$
dQ	q's exponent, a positive integer such that: $e \cdot dQ \equiv 1 \pmod{(q-1)}$
e	public exponent
EM	encoded message, an octet string
$emLen$	intended length in octets of an encoded message
H	hash value, an output of *Hash*
$Hash$	hash function
$hLen$	output length in octets of hash function *Hash*
K	RSA private key

k	length in octets of the modulus
l	intended length of octet string
lcm(. , .)	least common multiple of two nonnegative integers
m	message representative, an integer between 0 and n-1
M	message, an octet string
MGF	mask generation function
n	modulus
P	encoding parameters, an octet string
p, q	prime factors of the modulus
$qInv$	CRT coefficient, a positive integer less than p such that: $q \cdot qInv \equiv 1 \pmod{p}$
s	signature representative, an integer between 0 and n-1
S	signature, an octet string
x	a nonnegative integer
X	an octet string corresponding to x
\xor	bitwise exclusive-or of two octet strings
$\lambda(n)$	lcm(p-1, q-1), where $n = p$□q
||	concatenation operator
|| . ||	octet length operator

3. Key types

Two key types are employed in the primitives and schemes defined in this document: *RSA public key* and *RSA private key*. Together, an RSA public key and an RSA private key form an *RSA key pair*.

3.1 RSA public key

For the purposes of this document, an RSA public key consists of two components:

— n, the modulus, a nonnegative integer

— e, the public exponent, a nonnegative integer

In a *valid RSA public key*, the modulus n is a product of two odd primes p and q, and the public exponent e is an integer between 3 and n-1 satisfying gcd (e, $\lambda(n)$) = 1, where $\lambda(n)$ = lcm (p-1,q-1).

A recommended syntax for interchanging RSA public keys between implementations is given in Section 11.1.1; an implementation's internal representation may differ.

3.2 RSA private key

For the purposes of this document, an RSA private key may have either of two representations.

1. The first representation consists of the pair (n, d), where the components have the following meanings:

— n, the modulus, a nonnegative integer

— d, the private exponent, a nonnegative integer

2. The second representation consists of a quintuple (p, q, dP, dQ, $qInv$), where the components have the following meanings:

— p, the first factor, a nonnegative integer

— q, the second factor, a nonnegative integer

— dP, the first factor's exponent, a nonnegative integer

— dQ, the second factor's exponent, a nonnegative integer

— $qInv$, the CRT coefficient, a nonnegative integer

In a *valid RSA private key* with the first representation, the modulus n is the same as in the corresponding public key and is the product of two odd primes p and q, and the private exponent d is a positive integer less than n satisfying

$$e \cdot d \equiv 1 \pmod{\lambda(n)}$$

where e is the corresponding public exponent and $\lambda(n)$ is as defined above.

In a valid RSA private key with the second representation, the two factors p and q are the prime factors of the modulus n, the exponents dP and dQ are positive integers less than p and q respectively satisfying

$$e \cdot dP \equiv 1 \pmod{(p-1)}$$
$$e \cdot dQ \equiv 1 \pmod{(q-1)},$$

and the CRT coefficient $qInv$ is a positive integer less than p satisfying

$$q \cdot qInv \equiv 1 \pmod{p}.$$

A recommended syntax for interchanging RSA private keys between implementations, which includes components from both representations, is given in Section 11.1.2; an implementation's internal representation may differ.

4. Data conversion primitives

Two data conversion primitives are employed in the schemes defined in this document:

- I2OSP – Integer-to-Octet-String primitive
- OS2IP – Octet-String-to-Integer primitive

For the purposes of this document, and consistent with ASN.1 syntax, an octet string is an ordered sequence of octets (eight-bit bytes). The sequence is indexed from first (conventionally, leftmost) to last (rightmost). For purposes of conversion to and from integers, the first octet is considered the most significant in the following conversion primitives

4.1 I2OSP

I2OSP converts a nonnegative integer to an octet string of a specified length.

I2OSP (x, l)

Input:	x	nonnegative integer to be converted
	l	intended length of the resulting octet string
Output:	X	corresponding octet string of length l; or "integer too large"

Steps:

1. If $x \geq 256^l$, output "integer too large" and stop.

2. Write the integer x in its unique l-digit representation base 256:

$$x = x_{l-1}\, 256^{\,l-1} + x_{l-2}\, 256^{\,l-2} + \ldots + x_1\, 256 + x_0$$

where $0 \leq x_i < 256$ (note that one or more leading digits will be zero if

$x < 256^{\,l-1}$).

3. Let the octet X_i have the value x_{l-i} for $1 \leq i \leq l$. Output the octet string $X = X_1\, X_2 \ldots X_l$.

4.2 OS2IP

OS2IP converts an octet string to a nonnegative integer.

OS2IP (X)

Input:	X	octet string to be converted
Output:	x	corresponding nonnegative integer

Steps:

1. Let $X_1 X_2 \ldots X_l$ be the octets of X from first to last, and let x_{l-i} have value X_i for $1 \le i \le l$.
2. Let $x = x_{l-1}\, 256^{l-1} + x_{l-2}\, 256^{l-2} + \ldots + x_1\, 256 + x_0$.
3. Output x.

5. Cryptographic primitives

Cryptographic primitives are basic mathematical operations on which cryptographic schemes can be built. They are intended for implementation in hardware or as software modules, and are not intended to provide security apart from a scheme.

Four types of primitive are specified in this document, organized in pairs: encryption and decryption; and signature and verification.

The specifications of the primitives assume that certain conditions are met by the inputs, in particular that public and private keys are valid.

5.1 Encryption and decryption primitives

An encryption primitive produces a ciphertext representative from a message representative under the control of a public key, and a decryption primitive recovers the message representative from the ciphertext representative under the control of the corresponding private key.

One pair of encryption and decryption primitives is employed in the encryption schemes defined in this document and is specified here: RSAEP/RSADP. RSAEP and RSADP involve the same mathematical operation, with different keys as input.

The primitives defined here are the same as in the draft IEEE P1363 and are compatible with PKCS #1 v1.5.

The main mathematical operation in each primitive is exponentiation.

5.1.1 RSAEP

RSAEP ((n, e), m)

Input:	(n, e)	RSA public key
	m	message representative, an integer between 0 and n-1
Output:	c	ciphertext representative, an integer between 0 and n-1; or "message representative out of range"

Assumptions: public key (n, e) is valid

Steps:

1. If the message representative m is not between 0 and n-1, output "message representative out of range" and stop.

2. Let $c = m^e \bmod n$.

3. Output c.

5.1.2 RSADP

RSADP (K, c)

Input: K RSA private key, where K has one of the following forms:

— a pair (n, d)

— a quintuple (p, q, dP, dQ, $qInv$)

c ciphertext representative, an integer between 0 and n-1

Output: m message representative, an integer between 0 and n-1; or "ciphertext representative out of range"

Assumptions: private key K is valid

Steps:

1. If the ciphertext representative c is not between 0 and n-1, output "ciphertext representative out of range" and stop.

2. If the first form (n, d) of K is used:

   2.1 Let $m = c^d \bmod n$.

Else, if the second form (p, q, dP, dQ, $qInv$) of K is used:

2.2 Let $m_1 = c^{dP} \bmod p$.

2.3 Let $m_2 = c^{dQ} \bmod q$.

2.4 Let $h = qInv\ (m_1 - m_2) \bmod p$.

2.5 Let $m = m_2 + h\ q$.

3. Output m.

5.2 Signature and verification primitives

A signature primitive produces a signature representative from a message representative under the control of a private key, and a verification primitive recovers the message representative from the signature representative under the control of the corresponding public key. One pair of signature and verification primitives is employed in the signature schemes defined in this document and is specified here: RSASP1/RSAVP1.

The primitives defined here are the same as in the draft IEEE P1363 and are compatible with PKCS #1 v1.5.

The main mathematical operation in each primitive is exponentiation, as in the encryption and decryption primitives of Section 5.1. RSASP1 and RSAVP1 are the same as RSADP and RSAEP except for the names of their input and output arguments; they are distinguished as they are intended for different purposes.

5.2.1 RSASP1

RSASP1 (K, m)

Input:	K	RSA private key, where K has one of the following forms: — a pair (n, d) — a quintuple (p, q, dP, dQ, $qInv$)
	m	message representative, an integer between 0 and n-1
Output:	s	signature representative, an integer between 0 and n-1; or "message representative out of range"

Assumptions: private key K is valid

Steps:

1. If the message representative m is not between 0 and n-1, output “message representative out of range” and stop.

2. If the first form (n, d) of K is used:

   2.1 Let $s = m^d \bmod n$.

   Else, if the second form $(p, q, dP, dQ, qInv)$ of K is used:

   2.2 Let $s_1 = m^{dP} \bmod p$.

   2.3 Let $s_2 = m^{dQ} \bmod q$.

   2.4 Let $h = qInv\ (s_1 - s_2) \bmod p$.

   2.5 Let $s = s_2 + h\ q$.

3. Output s.

5.2.2 RSAVP1

RSAVP1 $((n, e), s)$

Input:	(n, e)	RSA public key
	s	signature representative, an integer between 0 and n-1
Output:	m	message representative, an integer between 0 and n-1; or “invalid”
Assumptions:	public key (n, e) is valid	

Steps:

1. If the signature representative s is not between 0 and n-1, output “invalid” and stop.

2. Let $m = s^e \bmod n$.

3. Output m.

6. Overview of schemes

A scheme combines cryptographic primitives and other techniques to achieve a particular security goal. Two types of scheme are specified in this document: encryption schemes and signature schemes with appendix.

The schemes specified in this document are limited in scope in that their operations consist only of steps to process data with a key, and do not include steps for obtaining or validating the key. Thus, in addition to the scheme operations, an application will typically include key management operations by which parties may select public and private keys for a scheme operation. The specific additional operations and other details are outside the scope of this document.

As was the case for the cryptographic primitives (Section 5), the specifications of scheme operations assume that certain conditions are met by the inputs, in particular that public and private keys are valid. The behavior of an implementation is thus unspecified when a key is invalid. The impact of such unspecified behavior depends on the application. Possible means of addressing key validation include explicit key validation by the application; key validation within the public-key infrastructure; and assignment of liability for operations performed with an invalid key to the party who generated the key.

7. Encryption schemes

An *encryption scheme* consists of an *encryption operation* and a *decryption operation*, where the encryption operation produces a ciphertext from a message with a recipient's public key, and the decryption operation recovers the message from the ciphertext with the recipient's corresponding private key.

An encryption scheme can be employed in a variety of applications. A typical application is a key establishment protocol, where the message contains key material to be delivered confidentially from one party to another. For instance, PKCS #7 [21] employs such a protocol to deliver a content-encryption key from a sender to a recipient; the encryption schemes defined here would be suitable key-encryption algorithms in that context.

Two encryption schemes are specified in this document: RSAES-OAEP and RSAES-PKCS1-v1_5. RSAES-OAEP is recommended for new applications; RSAES-PKCS1-v1_5 is included only for compatibility with existing applications, and is not recommended for new applications.

The encryption schemes given here follow a general model similar to that employed in IEEE P1363, by combining encryption and decryption primitives with an *encoding method* for encryption. The encryption operations apply a message encoding operation to a message to produce an encoded message, which is then converted to an integer message representative. An encryption primitive is applied to the message representative to produce the ciphertext. Reversing this, the decryption operations apply a decryption primitive to the ciphertext to recover a message representative, which is then converted to an octet string encoded message. A message decoding operation is applied to the encoded message to recover the message and verify the correctness of the decryption.

7.1 RSAES-OAEP

RSAES-OAEP combines the RSAEP and RSADP primitives (Sections 5.1.1 and 5.1.2) with the EME-OAEP encoding method (Section 9.1.1) EME-OAEP is based on the method found in [2]. It is compatible with the IFES scheme defined in the draft P1363 where the encryption and decryption primitives are IFEP-RSA and IFDP-RSA and the message encoding method is EME-OAEP. RSAES-OAEP can operate on messages of length up to k-2-2$hLen$ octets, where $hLen$ is the length of the hash function output for EME-OAEP and k is the length in octets of the recipient's RSA modulus.

Assuming that the hash function in EME-OAEP has appropriate properties, and the key size is sufficiently large, RSAEP-OAEP provides "plaintext-aware encryption," meaning that it is computationally infeasible to obtain full or partial information about a message from a ciphertext, and computationally infeasible to generate a valid ciphertext without knowing the corresponding message. Therefore, a chosen ciphertext attack is ineffective against a plaintext-aware encryption scheme such as RSAES-OAEP. We briefly note that to receive the full security benefit of RSAES-OAEP, it should not be used in a protocol involving RSAES-PKCS1-v1_5. It is possible that in a protocol in which both encryption schemes are present, an adaptive chosen ciphertext attack such as [4] would be useful.

Both the encryption and the decryption operations of RSAES-OAEP take the value of the parameter string P as input. In this version of PKCS #1, P is an octet string that is specified explicitly. See Section 11.2.1 for the relevant ASN.1 syntax.

7.1.1 Encryption operation

RSAES-OAEP-ENCRYPT ((n, e), M, P)

Input:	(n, e)	recipient's RSA public key
	M	message to be encrypted, an octet string of length at most k-2-2$hLen$, where k is the length in octets of the modulus n and $hLen$ is the length in octets of the hash function output for EME-OAEP
	P	encoding parameters, an octet string that may be empty
Output:	C	ciphertext, an octet string of length k; or "message too long"
Assumptions:	public key (n, e) is valid	

Steps:

1. Apply the EME-OAEP encoding operation (Section 9.1.1.2) to the message M and the encoding parameters P to produce an encoded message EM of length k-1 octets:

$$EM = \text{EME-OAEP-ENCODE}\ (M, P, k\text{-}1)$$

If the encoding operation outputs "message too long," then output "message too long" and stop.

2. Convert the encoded message EM to an integer message representative m:

$$m = \text{OS2IP}\ (EM)$$

3. Apply the RSAEP encryption primitive (Section 5.1.1) to the public key (n, e) and the message representative m to produce an integer ciphertext representative c:

$$c = \text{RSAEP}\ ((n, e), m)$$

4. Convert the ciphertext representative c to a ciphertext C of length k octets:

$$C = \text{I2OSP}\ (c, k)$$

5. Output the ciphertext C.

7.1.2 Decryption operation

RSAES-OAEP-DECRYPT (K, C, P)

Input:	K	recipient's RSA private key
	C	ciphertext to be decrypted, an octet string of length k, where k is the length in octets of the modulus n
	P	encoding parameters, an octet string that may be empty
Output:	M	message, an octet string of length at most k-2-2$hLen$, where $hLen$ is the length in octets of the hash function output for EME-OAEP; or "decryption error"

Steps:

1. If the length of the ciphertext C is not k octets, output "decryption error" and stop.

2. Convert the ciphertext C to an integer ciphertext representative c:

$$c = \text{OS2IP}\ (C)$$

3. Apply the RSADP decryption primitive (Section 5.1.2) to the private key K and the ciphertext representative c to produce an integer message representative m:

$$m = \text{RSADP}\ (K, c)$$

If RSADP outputs “ciphertext out of range,” then output “decryption error” and stop.

4. Convert the message representative m to an encoded message EM of length k-1 octets:

$$EM = \text{I2OSP}\ (m, k-1)$$

If I2OSP outputs “integer too large,” then output “decryption error” and stop.

5. Apply the EME-OAEP decoding operation to the encoded message EM and the encoding parameters P to recover a message M:

$$M = \text{EME-OAEP-Decode}\ (EM, P)$$

If the decoding operation outputs “decoding error,” then output “decryption error” and stop.

6. Output the message M.

Note. It is important that the error messages output in steps 4 and 5 be the same, otherwise an adversary may be able to extract useful information from the type of error message received. Error message information is used to mount a chosen ciphertext attack on PKCS #1 v1.5 encrypted messages in [4].

7.2 RSAES-PKCS1-v1_5

RSAES-PKCS1-v1_5 combines the RSAEP and RSADP primitives with the EME-PKCS1-v1_5 encoding method. It is the same as the encryption scheme in PKCS #1 v1.5. RSAES-PKCS1-v1_5 can operate on messages of length up to k-11 octets, although care should be taken to avoid certain attacks on low-exponent RSA due to Coppersmith, et al. when long messages are encrypted (see the third bullet in the notes below and [7]).

RSAES-PKCS1-v1_5 does not provide “plaintext aware” encryption. In particular, it is possible to generate valid ciphertexts without knowing the corresponding plaintexts, with a reasonable probability of success. This ability can be exploited in a chosen ciphertext attack as shown in [4]. Therefore, if RSAES-PKCS1-v1_5 is to be used, certain easily implemented countermeasures should be taken to thwart the attack found in [4]. The addition of structure to the data to be encoded, rigorous checking of PKCS #1 v1.5 conformance and other redundancy in decrypted messages, and the consolidation of error messages in a client-server protocol based on PKCS #1 v1.5 can all be effective countermeasures and don’t involve changes to a PKCS #1 v1.5-based protocol. These and other countermeasures are discussed in [5].

Notes. The following passages describe some security recommendations pertaining to the use of RSAES-PKCS1-v1_5. Recommendations from version 1.5 of this document are included as well as new recommendations motivated by cryptanalytic advances made in the intervening years.

- It is recommended that the pseudorandom octets in EME-PKCS1-v1_5 be generated independently for each encryption process, especially if the same data is input to more than one encryption process. Hastad's results [13] are one motivation for this recommendation.

- The padding string *PS* in EME-PKCS1-v1_5 is at least eight octets long, which is a security condition for public-key operations that prevents an attacker from recovering data by trying all possible encryption blocks.

- The pseudorandom octets can also help thwart an attack due to Coppersmith et al. [7] when the size of the message to be encrypted is kept small. The attack works on low-exponent RSA when similar messages are encrypted with the same public key. More specifically, in one flavor of the attack, when two inputs to RSAEP agree on a large fraction of bits (8/9) and low-exponent RSA ($e = 3$) is used to encrypt both of them, it may be possible to recover both inputs with the attack. Another flavor of the attack is successful in decrypting a single ciphertext when a large fraction (2/3) of the input to RSAEP is already known. For typical applications, the message to be encrypted is short (e.g., a 128-bit symmetric key) so not enough information will be known or common between two messages to enable the attack. However, if a long message is encrypted, or if part of a message is known, then the attack may be a concern. In any case, the RSAEP-OAEP scheme overcomes the attack.

7.2.1 Encryption operation

RSAES-PKCS1-v1_5-Encrypt ((n, e), M)

Input:	(n, e)	recipient's RSA public key
	M	message to be encrypted, an octet string of length at most k-11 octets, where k is the length in octets of the modulus n
Output:	C	ciphertext, an octet string of length k; or "message too long"

Steps:

1. Apply the EME-PKCS1-v1_5 encoding operation (Section 9.1.2.1) to the message M to produce an encoded message EM of length k-1 octets:

   $$EM = \text{EME-PKCS1-v1\_5-Encode}\ (M, k\text{-}1)$$

   If the encoding operation outputs "message too long," then output "message too long" and stop.

2. Convert the encoded message EM to an integer message representative m:

$$m = \text{OS2IP}\ (EM)$$

3. Apply the RSAEP encryption primitive (Section 5.1.1) to the public key (n, e) and the message representative m to produce an integer ciphertext representative c:

$$c = \text{RSAEP}\ ((n, e), m)$$

4. Convert the ciphertext representative c to a ciphertext C of length k octets:

$$C = \text{I2OSP}\ (c, k)$$

5. Output the ciphertext C.

7.2.2 Decryption operation

RSAES-PKCS1-v1_5-Decrypt (K, C)

Input:	K	recipient's RSA private key
	C	ciphertext to be decrypted, an octet string of length k, where k is the length in octets of the modulus n
Output:	M	message, an octet string of length at most k-11; or "decryption error"

Steps:

1. If the length of the ciphertext C is not k octets, output "decryption error" and stop.

2. Convert the ciphertext C to an integer ciphertext representative c:

$$c = \text{OS2IP}\ (C)$$

3. Apply the RSADP decryption primitive to the private key (n, d) and the ciphertext representative c to produce an integer message representative m:

$$m = \text{RSADP}\ ((n, d), c)$$

If RSADP outputs "ciphertext out of range," then output "decryption error" and stop.

4. Convert the message representative m to an encoded message EM of length k-1 octets:

$$EM = \text{I2OSP}\ (m, k\text{-}1)$$

If I2OSP outputs "integer too large," then output "decryption error" and stop.

5. Apply the EME-PKCS1-v1_5 decoding operation to the encoded message *EM* to recover a message *M*:

$$M = \text{EME-PKCS1-v1\_5-DECODE}\ (EM)$$

If the decoding operation outputs "decoding error," then output "decryption error" and stop.

6. Output the message *M*.

Note. It is important that only one type of error message is output by EME-PKCS1-v1_5, as ensured by steps 4 and 5. If this is not done, then an adversary may be able to use information extracted form the type of error message received to mount a chosen ciphertext attack such as the one found in [4].

8. Signature schemes with appendix

A *signature scheme with appendix* consists of a *signature generation operation* and a *signature verification operation*, where the signature generation operation produces a signature from a message with a signer's private key, and the signature verification operation verifies the signature on the message with the signer's corresponding public key. To verify a signature constructed with this type of scheme it is necessary to have the message itself. In this way, signature schemes with appendix are distinguished from signature schemes with message recovery, which are not supported in this document.

A signature scheme with appendix can be employed in a variety of applications. For instance, X.509 [6] employs such a scheme to authenticate the content of a certificate; the signature scheme with appendix defined here would be a suitable signature algorithm in that context. A related signature scheme could be employed in PKCS #7 [21], although for technical reasons, the current version of PKCS #7 separates a hash function from a signature scheme, which is different than what is done here.

One signature scheme with appendix is specified in this document: RSASSA-PKCS1-v1_5.

The signature scheme with appendix given here follows a general model similar to that employed in IEEE P1363, by combining signature and verification primitives with an encoding method for signatures. The signature generation operations apply a message encoding operation to a message to produce an encoded message, which is then converted to an integer message representative. A signature primitive is then applied to the message representative to produce the signature. The signature verification operations apply a signature verification primitive to the signature to recover a message representative, which is then converted to an octet string. The message encoding operation is again applied to the message, and the result is compared to the recovered octet string. If there is a match, the signature is considered valid. (Note that this approach assumes that the signature and verification primitives have the message-recovery form and the encoding method is deterministic, as is the case for RSASP1/RSAVP1 and EMSA-PKCS1-v1_5. The signature generation and verification operations have a different form in P1363 for other primitives and encoding methods.)

Editor's note. RSA Laboratories is investigating the possibility of including a scheme based on the PSS encoding methods specified in [3], which would be recommended for new applications.

8.1 RSASSA-PKCS1-v1_5

RSASSA-PKCS1-v1_5 combines the RSASP1 and RSAVP1 primitives with the EME-PKCS1-v1_5 encoding method. It is compatible with the IFSSA scheme defined in the draft P1363 where the signature and verification primitives are IFSP-RSA1 and IFVP-RSA1 and the message encoding method is EMSA-PKCS1-v1_5 (which is not defined in P1363). The length of messages on which RSASSA-PKCS1-v1_5 can operate is either unrestricted or constrained by a very large number, depending on the hash function underlying the message encoding method.

Assuming that the hash function in EMSA-PKCS1-v1_5 has appropriate properties and the key size is sufficiently large, RSASSA-PKCS1-v1_5 provides secure signatures, meaning that it is computationally infeasible to generate a signature without knowing the private key, and computationally infeasible to find a message with a given signature or two messages with the same signature. Also, in the encoding method EMSA-PKCS1-v1_5, a hash function identifier is embedded in the encoding. Because of this feature, an adversary must invert or find collisions of the particular hash function being used; attacking a different hash function than the one selected by the signer is not useful to the adversary.

8.1.1 Signature generation operation

RSASSA-PKCS1-v1_5-SIGN (K, M)

Input:	K	signer's RSA private key
	M	message to be signed, an octet string
Output:	S	signature, an octet string of length k, where k is the length in octets of the modulus n; "message too long" or "modulus too short"

Steps:

1. Apply the EMSA-PKCS1-v1_5 encoding operation (Section 9.2.1) to the message M to produce an encoded message EM of length k-1 octets:

$$EM = \text{EMSA-PKCS1-v1\_5-Encode}\ (M, k\text{-}1)$$

If the encoding operation outputs "message too long," then output "message too long" and stop. If the encoding operation outputs "intended encoded message length too short" then output "modulus too short".

2. Convert the encoded message *EM* to an integer message representative *m*:

$$m = \text{OS2IP}\ (EM)$$

3. Apply the RSASP1 signature primitive (Section 5.2.1) to the private key *K* and the message representative *m* to produce an integer signature representative *s*:

$$s = \text{RSASP1}\ (K, m)$$

4. Convert the signature representative *s* to a signature *S* of length *k* octets:

$$S = \text{I2OSP}\ (s, k)$$

5. Output the signature *S*.

8.1.2 Signature verification operation

RSASSA-PKCS1-v1_5-Verify ((*n*, *e*), *M, S*)

Input:	(*n*, *e*)	signer's RSA public key
	M	message whose signature is to be verified, an octet string
	S	signature to be verified, an octet string of length *k*, where *k* is the length in octets of the modulus *n*

Output: "valid signature," "invalid signature," or "message too long", or "modulus too short"

Steps:

1. If the length of the signature *S* is not *k* octets, output "invalid signature" and stop.

2. Convert the signature *S* to an integer signature representative *s*:

$$s = \text{OS2IP}\ (S)$$

3. Apply the RSAVP1 verification primitive (Section 5.2.2) to the public key (*n*, *e*) and the signature representative *s* to produce an integer message representative *m*:

$$m = \text{RSAVP1}\ ((n, e), s)$$

If RSAVP1 outputs “invalid” then output “invalid signature” and stop.

4. Convert the message representative m to an encoded message EM of length k-1 octets:

$$EM = \text{I2OSP}\ (m, k\text{-}1)$$

If I2OSP outputs “integer too large,” then output “invalid signature” and stop.

5. Apply the EMSA-PKCS1-v1_5 encoding operation (Section 9.2.1) to the message M to produce a second encoded message EM' of length k-1 octets:

$$EM' = \text{EMSA-PKCS1-v1\_5-ENCODE}\ (M, k\text{-}1)$$

If the encoding operation outputs “message too long,” then output “message too long” and stop. If the encoding operation outputs “intended encoded message length too short” then output “modulus too short”.

6. Compare the encoded message EM and the second encoded message EM'. If they are the same, output “valid signature”; otherwise, output “invalid signature.”

9. Encoding methods

Encoding methods consist of operations that map between octet string messages and integer message representatives.

Two types of encoding method are considered in this document: encoding methods for encryption, encoding methods for signatures with appendix.

9.1 Encoding methods for encryption

An encoding method for encryption consists of an *encoding operation* and a *decoding operation*. An encoding operation maps a message M to a message representative EM of a specified length; the decoding operation maps a message representative EM back to a message. The encoding and decoding operations are inverses.

The message representative EM will typically have some structure that can be verified by the decoding operation; the decoding operation will output “decoding error” if the structure is not present. The encoding operation may also introduce some randomness, so that different applications of the encoding operation to the same message will produce different representatives.

Two encoding methods for encryption are employed in the encryption schemes and are specified here: EME-OAEP and EME-PKCS1-v1_5.

9.1.1 EME-OAEP

This encoding method is parameterized by the choice of hash function and mask generation function. Suggested hash and mask generation functions are given in Section 10. This encoding method is based on the method found in [2]. illustrates the encoding operation.

9.1.1.1 Encoding operation

EME-OAEP-ENCODE (*M*, *P*, *emLen*)

Options:	*Hash*	hash function (*hLen* denotes the length in octets of the hash function output)
	MGF	mask generation function
Input:	*M*	message to be encoded, an octet string of length at most *emLen*-1-2*hLen*
	P	encoding parameters, an octet string
	emLen	intended length in octets of the encoded message, at least 2*hLen*+1
Output:	*EM*	encoded message, an octet string of length *emLen*; or “message too long” or “parameter string too long”

Steps:

1. If the length of *P* is greater than the input limitation for the hash function (2^{61}-1 octets for SHA-1) then output “parameter string too long” and stop.

2. If $||M|| > emLen\text{-}2hLen\text{-}1$ then output “message too long” and stop.

3. Generate an octet string *PS* consisting of $emLen\text{-}||M||\text{-}2hLen\text{-}1$ zero octets. The length of *PS* may be 0.

4. Let $pHash = Hash(P)$, an octet string of length *hLen*.

5. Concatenate *pHash*, *PS*, the message *M*, and other padding to form a data block *DB* as

$$DB = pHash \,||\, PS \,||\, 01 \,||\, M$$

6. Generate a random octet string *seed* of length *hLen*.

7. Let $dbMask = MGF(seed, emLen – hLen)$.

8. Let *maskedDB* = *DB* \xor *dbMask*.

9. Let *seedMask* = *MGF*(*maskedDB*, *hLen*).

10. Let *maskedSeed* = seed \xor *seedMask*.

11. Let *EM* = *maskedSeed* || *maskedDB*.

12. Output *EM*.

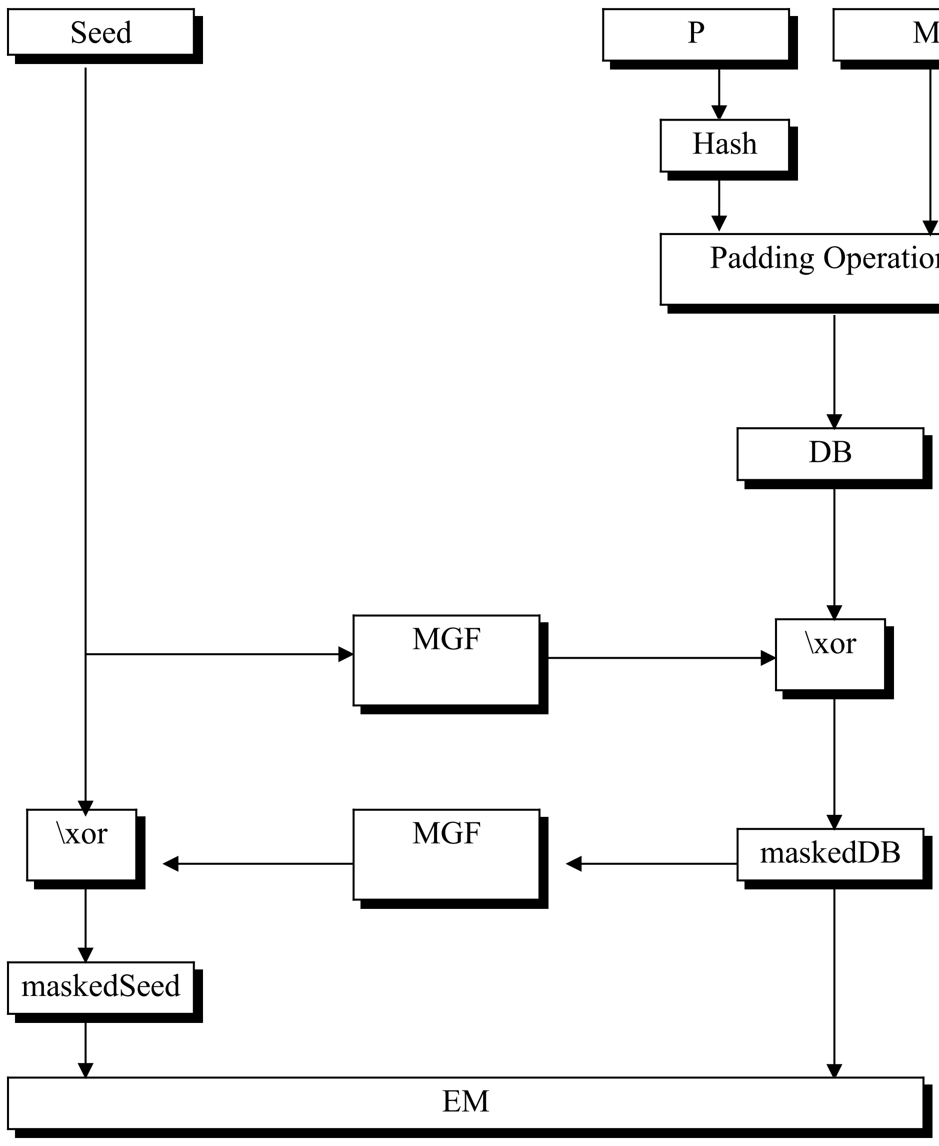

Figure 9-1: EME-OAEP

9.1.1.2 Decoding operation

EME-OAEP-Decode (*EM*, *P*)

Options:	*Hash*	hash function (*hLen* denotes the length in octets of the hash function output)
	MGF	mask generation function

Input:	*EM*	encoded message, an octet string of length at least 2*hLen*+1
	P	encoding parameters, an octet string
Output:	*M*	recovered message, an octet string of length at most $\|\|EM\|\|$-1-2*hLen*; or "decoding error"

Steps:

1. If the length of *P* is greater than the input limitation of the hash function (2^{61}-1 octets for SHA-1) then output "decoding error" and stop.

2. If $\|\|EM\|\| < 2hLen+1$, then output "decoding error" and stop.

3. Let *maskedSeed* be the first *hLen* octets of *EM* and let *maskedDB* be the remaining $\|\|EM\|\|$ - *hLen* octets.

4. Let *seedMask* = *MGF*(*maskedDB*, *hLen*).

5. Let *seed* = *maskedSeed* \xor *seedMask*.

6. Let *dbMask* = *MGF*(*seed*, $\|\|EM\|\|$ - *hLen*).

7. Let *DB* = *maskedDB* \xor *dbMask*.

8. Let *pHash* = *Hash*(*P*), an octet string of length *hLen*.

9. Separate *DB* into an octet string *pHash'* consisting of the first *hLen* octets of *DB*, a (possibly empty) octet string *PS* consisting of consecutive zero octets following *pHash'*, and a message *M* as

$$DB = pHash' \,\|\, PS \,\|\, 01 \,\|\, M$$

   If there is no 01 octet to separate *PS* from *M*, output "decoding error" and stop.

10. If *pHash*' does not equal *pHash*, output "decoding error" and stop.

11. Output *M*.

9.1.2 EME-PKCS1-v1_5

This encoding method is the same as in PKCS #1 v1.5, Section 8: Encryption Process.

9.1.2.1 Encoding operation

EME-PKCS1-v1_5-Encode (M, $emLen$)

Input:	M	message to be encoded, an octet string of length at most $emLen$-10
	$emLen$	intended length in octets of the encoded message
Output:	EM	encoded message, an octet string of length $emLen$; or “message too long”

Steps:

1. If the length of the message M is greater than $emLen$ – 10 octets, output “message too long” and stop.

2. Generate an octet string PS of length $emLen$-$||M||$-2 consisting of pseudorandomly generated nonzero octets. The length of PS will be at least 8 octets.

3. Concatenate PS, the message M, and other padding to form the encoded message EM as

$$EM = 02 \,||\, PS \,||\, 00 \,||\, M$$

4. Output EM.

9.1.2.2 Decoding operation

EME-PKCS1-v1_5-Decode (EM)

Input:	EM	encoded message, an octet string of length at least 10
Output:	M	recovered message, an octet string of length at most $||EM||$-10; or “decoding error”

Steps:

1. If the length of the encoded message EM is less than 10, output “decoding error” and stop.

2. Separate the encoded message EM into an octet string PS consisting of nonzero octets and a message M as

$$EM = 02 \,||\, PS \,||\, 00 \,||\, M.$$

   If the first octet of EM is not 02, or if there is no 00 octet to separate PS from M, output “decoding error” and stop.

3. If the length of *PS* is less than 8 octets, output "decoding error" and stop.

4. Output *M*.

9.2 Encoding methods for signatures with appendix

An *encoding method for signatures with appendix*, for the purposes of this document, consists of an encoding operation. An encoding operation maps a message *M* to a message representative *EM* of a specified length. (In future versions of this document, encoding methods may be added that also include a decoding operation.)

One encoding method for signatures with appendix is employed in the encryption schemes and is specified here: EMSA-PKCS1-v1_5.

9.2.1 EMSA-PKCS1-v1_5

This encoding method only has an encoding operation.

EMSA-PKCS1-v1_5-ENCODE (*M*, *emLen*)

Option:	*Hash*	hash function (*hLen* denotes the length in octets of the hash function output)
Input:	*M*	message to be encoded
	emLen	intended length in octets of the encoded message, at least $\|\|T\|\| + 10$, where *T* is the DER encoding of a certain value computed during the encoding operation
Output:	*EM*	encoded message, an octet string of length *emLen*; or "message too long" or "intended encoded message length too short"

Steps:

1. Apply the hash function to the message *M* to produce a hash value *H*:

$$H = Hash(M).$$

   If the hash function outputs "message too long," then output "message too long".

2. Encode the algorithm ID for the hash function and the hash value into an ASN.1 value of type `DigestInfo` (see Section 11) with the Distinguished Encoding Rules (DER), where the type `DigestInfo` has the syntax

```
DigestInfo ::= SEQUENCE {
  digestAlgorithm AlgorithmIdentifier,
  digest OCTET STRING }
```

The first field identifies the hash function and the second contains the hash value. Let *T* be the DER encoding.

3. If *emLen* is less than ||*T*|| + 10 then output "intended encoded message length too short".

4. Generate an octet string *PS* consisting of *emLen*-||*T*||-2 octets with value FF (hexadecimal). The length of *PS* will be at least 8 octets.

5. Concatenate *PS*, the DER encoding *T*, and other padding to form the encoded message *EM* as

$$EM = 01 \parallel PS \parallel 00 \parallel T$$

6. Output *EM*.

10. Auxiliary functions

This section specifies the hash functions and the mask generation functions that are mentioned in the encoding methods (Section 9).

10.1 Hash functions

Hash functions are used in the operations contained in Sections 7, 8 and 9. Hash functions are deterministic, meaning that the output is completely determined by the input. Hash functions take octet strings of variable length, and generate fixed length octet strings. The hash functions used in the operations contained in Sections 7, 8 and 9 should be *collision resistant*. This means that it is infeasible to find two distinct inputs to the hash function that produce the same output. A collision resistant hash function also has the desirable property of being *one-way*; this means that given an output, it is infeasible to find an input whose hash is the specified output. The property of collision resistance is especially desirable for RSASSA-PKCS1-v1_5, as it makes it infeasible to forge signatures. In addition to the requirements, the hash function should yield a mask generation function (Section 10.2) with pseudorandom output.

Three hash functions are recommended for the encoding methods in this document: MD2 [15], MD5 [17], and SHA-1 [16]. For the EME-OAEP encoding method, only SHA-1 is recommended. For the EMSA-PKCS1-v1_5 encoding method, SHA-1 is recommended for new applications. MD2 and MD5 are recommended only for compatibility with existing applications based on PKCS #1 v1.5.

The hash functions themselves are not defined here; readers are referred to the appropriate references ([15], [17] and [16]).

Note. Version 1.5 of this document also allowed for the use of MD4 in signature schemes. The cryptanalysis of MD4 has progressed significantly in the intervening years. For example, Dobbertin [10] demonstrated how to find collisions for MD4 and that the first two rounds of MD4 are not one-way [11]. Because of these results and others (e.g. [9]), MD4 is no longer recommended. There have also been advances in the cryptanalysis of MD2 and MD5, although not enough to warrant removal from existing applications. Rogier and Chauvaud [19] demonstrated how to find collisions in a modified version of MD2. No one has demonstrated how to find collisions for the full MD5 algorithm, although partial results have been found (e.g. [8]). For new applications, to address these concerns, SHA-1 is preferred.

10.2 Mask generation functions

A mask generation function takes an octet string of variable length and a desired output length as input, and outputs an octet string of the desired length. There may be restrictions on the length of the input and output octet strings, but such bounds are generally very large. Mask generation functions are deterministic; the octet string output is completely determined by the input octet string. The output of a mask generation function should be pseudorandom, that is, if the seed to the function is unknown, it should be infeasible to distinguish the output from a truly random string. The plaintext-awareness of RSAES-OAEP relies on the random nature of the output of the mask generation function, which in turn relies on the random nature of the underlying hash.

One mask generation function is recommended for the encoding methods in this document, and is defined here: *MGF1*, which is based on a hash function. Future versions of this document may define other mask generation functions.

10.2.1 MGF1

MGF1 is a Mask Generation Function based on a hash function.

MGF1 (Z, l)

Options:	*Hash*	hash function (*hLen* denotes the length in octets of the hash function output)
Input:	Z	seed from which mask is generated, an octet string
	l	intended length in octets of the mask, at most 2^{32} *hLen*
Output:	*mask*	mask, an octet string of length l; or "mask too long"

Steps:

1. If $l > 2^{32}$ *hLen*, output "mask too long" and stop.

2. Let T be the empty octet string.

3. For *counter* from 0 to $\lceil l / hLen \rceil$-1, do the following:

   a. Convert *counter* to an octet string C of length 4 with the primitive I2OSP:

   $$C = \text{I2OSP}(counter, 4)$$

   b. Concatenate the hash of the seed Z and C to the octet string T:

   $$T = T \,||\, Hash\,(Z \,||\, C)$$

4. Output the leading l octets of T as the octet string *mask*.

11. ASN.1

11.1 Key representation

This section defines ASN.1 object identifiers for RSA public and private keys, and defines the types `RSAPublicKey` and `RSAPrivateKey`. The intended application of these definitions includes X.509 certificates, PKCS #8 [22], and PKCS #12 [23].

The object identifier `rsaEncryption` identifies RSA public and private keys as defined in Sections 11.1.1 and 11.1.2. The `parameters` field associated with this OID in an `AlgorithmIdentifier` shall have type `NULL`.

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

All of the definitions in this section are the same as in PKCS #1 v1.5.

11.1.1 Public-key syntax

An RSA public key should be represented with the ASN.1 type `RSAPublicKey`:

```
RSAPublicKey ::= SEQUENCE {
  modulus INTEGER, -- n
  publicExponent INTEGER -- e }
```

(This type is specified in X.509 and is retained here for compatibility.)

The fields of type `RSAPublicKey` have the following meanings:

- `modulus` is the modulus n.

- `publicExponent` is the public exponent *e*.

11.1.2 Private-key syntax

An RSA private key should be represented with ASN.1 type `RSAPrivateKey`:

```
RSAPrivateKey ::= SEQUENCE {
  version Version,
  modulus INTEGER, -- n
  publicExponent INTEGER, -- e
  privateExponent INTEGER, -- d
  prime1 INTEGER, -- p
  prime2 INTEGER, -- q
  exponent1 INTEGER, -- d mod (p-1)
  exponent2 INTEGER, -- d mod (q-1)
  coefficient INTEGER -- (inverse of q) mod p }

Version ::= INTEGER
```

The fields of type `RSAPrivateKey` have the following meanings:

- `version` is the version number, for compatibility with future revisions of this document. It shall be 0 for this version of the document.
- `modulus` is the modulus *n*.
- `publicExponent` is the public exponent *e*.
- `privateExponent` is the private exponent *d*.
- `prime1` is the prime factor *p* of *n*.
- `prime2` is the prime factor *q* of *n*.
- `exponent1` is *d* mod $(p-1)$.
- `exponent2` is *d* mod $(q-1)$.
- `coefficient` is the Chinese Remainder Theorem coefficient q^{-1} mod *p*.

11.2 Scheme identification

This section defines object identifiers for the encryption and signature schemes. The schemes compatible with PKCS #1 v1.5 have the same definitions as in PKCS #1 v1.5. The intended application of these definitions includes X.509 certificates and PKCS #7.

11.2.1 Syntax for RSAES-OAEP

The object identifier `id-RSAES-OAEP` identifies the RSAES-OAEP encryption scheme.

```
id-RSAES-OAEP OBJECT IDENTIFIER ::= {pkcs-1 7}
```

The `parameters` field associated with this OID in an `AlgorithmIdentifier` shall have type `RSAEP-OAEP-params`:

```
RSAES-OAEP-params ::=  SEQUENCE {
  hashFunc [0] AlgorithmIdentifier {{oaepDigestAlgorithms}}
    DEFAULT sha1Identifier,
  maskGenFunc [1] AlgorithmIdentifier {{pkcs1MGFAlgorithms}}
    DEFAULT mgf1SHA1Identifier,
  pSourceFunc [2] AlgorithmIdentifier
    {{pkcs1pSourceAlgorithms}}
    DEFAULT pSpecifiedEmptyIdentifier }
```

The fields of type `RSAES-OAEP-params` have the following meanings:

- `hashFunc` identifies the hash function. It shall be an algorithm ID with an OID in the set `oaepDigestAlgorithms`, which for this version shall consist of `id-sha1`, identifying the SHA-1 hash function. The `parameters` field for `id-sha1` shall have type `NULL`.

```
oaepDigestAlgorithms ALGORITHM-IDENTIFIER ::= {
  {NULL IDENTIFIED BY id-sha1} }

id-sha1 OBJECT IDENTIFIER ::=
  {iso(1) identified-organization(3) oiw(14) secsig(3)
    algorithms(2) 26}
```

  The default hash function is SHA-1:

```
sha1Identifier ::= AlgorithmIdentifier {id-sha1, NULL}
```

- `maskGenFunc` identifies the mask generation function. It shall be an algorithm ID with an OID in the set `pkcs1MGFAlgorithms`, which for this version shall consist of `id-mgf1`, identifying the *MGF1* mask generation function (see Section 10.2.1).

The `parameters` field for `id-mgf1` shall have type `AlgorithmIdentifier`, identifying the hash function on which *MGF1* is based, where the OID for the hash function shall be in the set `oaepDigestAlgorithms`.

```
pkcs1MGFAlgorithms ALGORITHM-IDENTIFIER ::= {
  {AlgorithmIdentifier {{oaepDigestAlgorithms}} IDENTIFIED
    BY id-mgf1} }

id-mgf1 OBJECT IDENTIFIER ::= {pkcs-1 8}
```

The default mask generation function is *MGF1* with SHA-1:

```
mgf1SHA1Identifier ::= AlgorithmIdentifier {
  id-mgf1, sha1Identifier }
```

- `pSourceFunc` identifies the source (and possibly the value) of the encoding parameters *P*. It shall be an algorithm ID with an OID in the set `pkcs1pSourceAlgorithms`, which for this version shall consist of `id-pSpecified`, indicating that the encoding parameters are specified explicitly. The `parameters` field for `id-pSpecified` shall have type `OCTET STRING`, containing the encoding parameters.

```
pkcs1pSourceAlgorithms ALGORITHM-IDENTIFIER ::= {
  {OCTET STRING IDENTIFIED BY id-pSpecified} }

id-pSpecified OBJECT IDENTIFIER ::= {pkcs-1 9}
```

The default encoding parameters is an empty string (so that *pHash* in EME-OAEP will contain the hash of the empty string):

```
pSpecifiedEmptyIdentifier ::= AlgorithmIdentifier {
  id-pSpecified, OCTET STRING SIZE (0) }
```

If all of the default values of the fields in `RSAES-OAEP-params` are used, then the algorithm identifier will have the following value:

```
RSAES-OAEP-Default-Identifier ::= AlgorithmIdentifier {
  id-RSAES-OAEP,
  {sha1Identifier,
   mgf1SHA1Identifier,
   pSpecifiedEmptyIdentifier } }
```

11.2.2 Syntax for RSAES-PKCS1-v1_5

The object identifier `rsaEncryption` (Section 11.1) identifies the RSAES-PKCS1-v1_5 encryption scheme. The `parameters` field associated with this OID in an

`AlgorithmIdentifier` shall have type `NULL`. This is the same as in PKCS #1 v1.5.

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

11.2.3 Syntax for RSASSA-PKCS1-v1_5

The object identifier for RSASSA-PKCS1-v1_5 shall be one of the following. The choice of OID depends on the choice of hash algorithm: MD2, MD5 or SHA-1. Note that if either MD2 or MD5 is used then the OID is just as in PKCS #1 v1.5. For each OID, the `parameters` field associated with this OID in an `AlgorithmIdentifier` shall have type `NULL`.

If the hash function to be used is MD2, then the OID should be:

```
md2WithRSAEncryption ::= {pkcs-1 2}
```

If the hash function to be used is MD5, then the OID should be:

```
md5WithRSAEncryption ::= {pkcs-1 4}
```

If the hash function to be used is SHA-1, then the OID should be:

```
sha1WithRSAEncryption ::= {pkcs-1 5}
```

In the `digestInfo` type mentioned in Section 9.2.1 the OIDS for the digest algorithm are the following:

```
id-SHA1 OBJECT IDENTIFIER ::=
     {iso(1)  identified-organization(3)  oiw(14)  secsig(3)
algorithms(2) 26 }

md2 OBJECT IDENTIFIER ::=
     {iso(1)    member-body(2)    us(840)    rsadsi(113549)
digestAlgorithm(2) 2}

md5 OBJECT IDENTIFIER ::=
     {iso(1)    member-body(2)    us(840)    rsadsi(113549)
digestAlgorithm(2) 5}
```

The parameters field of the digest algorithm has ASN.1 type `NULL` for these OIDs.

12. Revision history

Versions 1.0–1.3

Versions 1.0–1.3 were distributed to participants in RSA Data Security, Inc.'s Public-Key Cryptography Standards meetings in February and March 1991.

Version 1.4

Version 1.4 was part of the June 3, 1991 initial public release of PKCS. Version 1.4 was published as NIST/OSI Implementors' Workshop document SEC-SIG-91-18.

Version 1.5

Version 1.5 incorporated several editorial changes, including updates to the references and the addition of a revision history. The following substantive changes were made:

- Section 10: "MD4 with RSA" signature and verification processes were added.
- Section 11: `md4WithRSAEncryption` object identifier was added.

Version 2.0

Version 2.0 incorporates major editorial changes in terms of the document structure, and introduces the RSAEP-OAEP encryption scheme. This version continues to support the encryption and signature processes in version 1.5, although the hash algorithm MD4 is no longer allowed due to cryptanalytic advances in the intervening years.

13. References

[1] ANSI, *ANSI X9.44: Key Management Using Reversible Public Key Cryptography for the Financial Services Industry*. Working Draft.

[2] M. Bellare and P. Rogaway. *Optimal Asymmetric Encryption-How to Encrypt with RSA*. In Advances in Cryptology-Eurocrypt '94, pp. 92-111, Springer-Verlag, 1994.

[3] M. Bellare and P. Rogaway. *The Exact Security of Digital Signatures-How to Sign with RSA and Rabin*. In Advances in Cryptology-Eurocrypt '96, pp. 399-416, Springer-Verlag, 1996.

[4] D. Bleichenbacher. *Chosen Ciphertext Attacks against Protocols Based on the RSA Encryption Standard PKCS #1*. To appear in Advances in Cryptology-Crypto '98.

[5] D. Bleichenbacher, B. Kaliski and J. Staddon. *Recent Results on PKCS #1: RSA Encryption Standard*. RSA Laboratories' Bulletin, Number 7, June 24, 1998.

[6] CCITT. *Recommendation X.509: The Directory-Authentication Framework. 1988.*

[7] D. Coppersmith, M. Franklin, J. Patarin and M. Reiter. *Low-Exponent RSA with Related Messages*. In Advances in Cryptology-Eurocrypt `96, pp. 1-9, Springer-Verlag, 1996.

[8] B. den Boer and Bosselaers. *Collisions for the Compression Function of MD5*. Advances in Cryptology-Eurocrypt `93, pp. 293-304, Springer-Verlag, 1994.

[9] B. den Boer, and A. Bosselaers. *An Attack on the Last Two Rounds of MD4*. In Advances in Cryptology-Crypto `91, pp.194-203, Springer-Verlag, 1992.

[10] H. Dobbertin. *Cryptanalysis of MD4*. Fast Software Encryption. Lecture Notes in Computer Science, Springer-Verlag 1996, pp. 55-72.

[11] H. Dobbertin. *Cryptanalysis of MD5 Compress*. Presented at the rump session of Eurocrypt `96, May 14, 1996

[12] H. Dobbertin.*The First Two Rounds of MD4 are Not One-Way*. Fast Software Encryption. Lecture Notes in Computer Science, Springer-Verlag 1998, pp. 284-292.

[13] J. Hastad. *Solving Simultaneous Modular Equations of Low Degree*. SIAM Journal of Computing, 17, 1988, pp. 336-341.

[14] IEEE. *IEEE P1363: Standard Specifications for Public Key Cryptography*. Draft Version 4.

[15] B. Kaliski. *RFC 1319: The MD2 Message-Digest Algorithm*. Internet Activities Board, April 1992.

[16] National Institute of Standards and Technology (NIST). *FIPS Publication 180-1: Secure Hash Standard*. April 1994.

[17] R. Rivest. *RFC 1321: The MD5 Message-Digest Algorithm*. Internet Activities Board, April 1992.

[18] R. Rivest, A. Shamir and L. Adleman. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*. Communications of the ACM, 21(2), pp. 120-126, February 1978.

[19] N. Rogier and P. Chauvaud. *The Compression Function of MD2 is not Collision Free*. Presented at Selected Areas of Cryptography `95. Carleton University, Ottawa, Canada. May 18-19, 1995.

[20] RSA Laboratories. *PKCS #1: RSA Encryption Standard*. Version 1.5, November 1993.

[21] RSA Laboratories. *PKCS #7: Cryptographic Message Syntax Standard*. Version 1.5, November 1993.

[22] RSA Laboratories. *PKCS #8: Private-Key Information Syntax Standard*. Version 1.2, November 1993.

[23] RSA Laboratories. *PKCS #12: Personal Information Exchange Syntax Standard*. Version 1.0, DRAFT, April 1997.