PKCS #1 v2.1: RSA Cryptography Standard

RSA Laboratories

DRAFT 2 — January 5, 2001

Editor's note: This is the second draft of PKCS #1 v2.1, which is available for a 30-day public review period. Please send comments and suggestions, both technical and editorial, to pkcs-editor@rsasecurity.com or pkcs-tng@rsasecurity.com.

Table of Contents

1. Introduction

This document provides recommendations for the implementation of public-key cryptography based on the RSA algorithm [36], 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 standards and draft standards currently being developed by the ANSI X9F1 [1] and IEEE P1363 working groups [25][26].

This document supersedes PKCS #1 version 2.0 [38] but includes compatible techniques.

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.
- Sections 6, 7 and 8 deal with the encryption and signature schemes in this document. Section 6 gives an overview. Along with the methods found in PKCS #1 v1.5, Section 7 defines an OAEP-based [3] encryption scheme and Section 8 defines a PSS-based [4][5] signature scheme with appendix.
- Section 9 defines the encoding methods for the encryption and signature schemes in Sections 7 and 8.
- Appendix A defines the ASN.1 syntax for the keys defined in Section 3 and the schemes in Sections 7 and 8.
- Appendix B defines the hash functions and the mask generation function used in this document, including ASN.1 syntax for the techniques.

- Appendix C gives an ASN.1 module.
- Appendices D, E, F and G cover intellectual property issues, outline the revision history of PKCS #1, give references to other publications and standards, and provide general information about the Public-Key Cryptography 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:

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

d_i additional factor r_i's exponent, a positive integer such that:

$$e \cdot d_i \equiv 1 \pmod{(r_i - 1)},\ i = 3, \ldots, f$$

e public exponent

f number of prime factors of the modulus, $f \geq 2$

EM encoded message, an octet string

$emLen$ (intended) length in octets of an encoded message EM

GCD (. , .) greatest common divisor of two nonnegative integers

H hash value, an output of $Hash$

$Hash$ hash function

$hLen$ output length in octets of hash function $Hash$

k length in octets of the modulus

K RSA private key

l	intended length of octet string
LCM (. , … , .)	least common multiple of a list of nonnegative integers
m	message representative, an integer between 0 and n–1
M	message, an octet string
mask	mask, an octet string
MGF	mask generation function
mLen	length in octets of a message
n	modulus, $n = r_1 \cdot r_2 \cdot \ldots \cdot r_f$, $f \geq 2$
P	encoding parameters, an octet string
p, q	first two prime factors of the modulus
qInv	CRT coefficient, a positive integer less than p such that: $$q \cdot qInv \equiv 1 \pmod{p}$$
r_i	prime factors of the modulus, including $r_1 = p$, $r_2 = q$, and additional factors if any
s	signature representative, an integer between 0 and n–1
S	signature, an octet string
salt	salt value, an octet string
sLen	length in octets of *salt*
t_i	additional factor r_i's CRT coefficient, a positive integer less than r_i such that $$r_1 \cdot r_2 \cdot \ldots \cdot r_{(i-1)} \cdot t_i \equiv 1 \pmod{r_i}, \; i = 3, \ldots, f$$
x	a nonnegative integer
X	an octet string corresponding to x
Z	seed from which mask is generated, an octet string
$\lambda(n)$	LCM $(r_1 - 1, r_2 - 1, \ldots, r_f - 1)$

$\oplus$ bitwise exclusive-or of two octet strings

$\|$ concatenation operator

$\|\cdot\|$ octet length operator

Note. The CRT can be applied in a non-recursive as well as a recursive way. In this document a recursive approach following Garner's algorithm [21] is used. See also Note 1 in Section 3.2.

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*.

This specification supports so-called "multi-prime" RSA where the modulus may have more than two prime factors. The benefit of multi-prime RSA is lower computational cost for the decryption and signature primitives, provided that the CRT (Chinese Remainder Theorem) is used. Better performance can be achieved on single processor platforms, but to a greater extent on multiprocessor platforms, where the modular exponentiations involved can be done in parallel.

For a discussion on how multi-prime affects the security of the RSA cryptosystem, the reader is referred to [43][42].

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 f distinct odd primes r_i, $i = 1, 2, \ldots, f$, where $f \geq 2$ and the public exponent e is an integer between 3 and $n-1$ satisfying GCD $(e, \lambda(n)) = 1$, where $\lambda(n) = \text{LCM}\ (r_1 - 1, \ldots, r_f - 1)$. By convention, the first two primes r_1 and r_2 may also be denoted p and q respectively.

A recommended syntax for interchanging RSA public keys between implementations is given in Appendix A.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 $(p, q, dP, dQ, qInv)$ and a (possibly empty) sequence of triplets (r_i, d_i, t_i), $i = 3, \ldots, f$, one for each prime not in the quintuple, 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 (first) CRT coefficient, a nonnegative integer

— r_i, the i^{th} factor, a nonnegative integer

— d_i, the i^{th} factor's exponent, a nonnegative integer

— t_i, the i^{th} factor's 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 f distinct odd primes r_i, $i = 1, 2, \ldots, f$, where $f \geq 2$. 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 *first two* prime factors of the modulus n (i.e., r_1 and r_2), 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

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

If $f > 2$, the representation will include one or more triplets (r_i, d_i, t_i), $i = 3, \ldots, f$. The factors r_i are the additional prime factors of the modulus n. Each exponent d_i satisfies

$$e \cdot d_i \equiv 1 \pmod{(r_i - 1)},\ i = 3, \ldots, f.$$

Each CRT coefficient t_i, $i = 3, \ldots, f$, is a positive integer less than r_i satisfying

$$R_i \cdot t_i \equiv 1 \pmod{r_i},$$

where $R_i = r_1 \cdot r_2 \cdot \ldots \cdot r_{(i-1)}$.

A recommended syntax for interchanging RSA private keys between implementations, which includes components from both representations, is given in Appendix A.1.2; an implementation's internal representation may differ.

Notes.

1. The definition of the CRT coefficients here and the formulas that use them in the primitives in Section 5 generally follows Garner's algorithm [21] (see also Algorithm 14.71 in [31]). However, for compatibility with the representations of RSA private keys in PKCS #1 v2.0 and previous versions, the roles of p and q are reversed compared to the rest of the primes. Thus, the first CRT coefficient, *qInv*, is defined as the inverse of q mod p, rather than as the inverse of R_1 mod r_2, i.e., of p mod q.

2. Quisquater and Couvreur [34] observed the benefit of applying the Chinese Remainder Theorem to RSA operations.

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

Errors: "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 integer 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} be the integer value of the octet X_i for $1 \leq i \leq l$.

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 IFEP-RSA/IFDP-RSA in IEEE Std 1363-2000 [25] (except that support for multi-prime RSA has been added) 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
Errors:	"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, $qInv$) and a (possibly empty) sequence of triplets (r_i, d_i, t_i), i = 3, …, f

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

Output: m message representative, an integer between 0 and n–1

Errors: "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, $qInv$) and (r_i, d_i, t_i) of K is used:

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

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

   2.4 If $f > 2$, then let $m_i = c^{d_i} \bmod r_i$, i = 3, …, f.

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

   2.6 Let $m = m_2 + q \cdot h$.

   2.7 If $f > 2$, then let $R = r_1$ and for i = 3 to f do

      2.7.1 Let $R = R \cdot r_{(i-1)}$.

      2.7.2 Let $h = (m_i - m) \cdot t_i \pmod{r_i}$.

      2.7.3 Let $m = m + R \cdot h$.

3. Output m.

Note. Steps 2.2–2.7 can be rewritten as a single loop, provided that one reverses the order of p and q. For consistency with PKCS #1 v2.0, however, the first two primes p and q are treated separately from the additional primes.

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 IFSP-RSA1/IFVP-RSA1 in IEEE 1363-2000 [25] (except that support for multi-prime RSA has been added) 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, $qInv$) and a (possibly empty) sequence of triplets (r_i, d_i, t_i), i = 3, …, f

m message representative, an integer between 0 and n–1

Output: s signature representative, an integer between 0 and n–1

Errors: "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 $(p, q, dP, dQ, qInv)$ and (r_i, d_i, t_i) of K is used:

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

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

2.4 If $f > 2$, then let $s_i = m^{d_i} \bmod r_i$, $i = 3, \ldots, f$.

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

2.6 Let $s = s_2 + q \cdot h$.

2.7 If $f > 2$, then let $R = r_1$ and for $i = 3$ to f do

2.7.1 Let $R = R \cdot r_{(i-1)}$.

2.7.2 Let $h = (s_i - s) \cdot t_i \pmod{r_i}$.

2.7.3 Let $s = s + R \cdot h$.

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
Errors:	"signature representative out of range"	
Assumptions:	public key (n, e) is valid	

Steps:

1. If the signature representative s is not between 0 and n–1, output "signature representative out of range" 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.

A generally good cryptographic practice is to employ a given key pair in only one scheme. This avoids the risk that vulnerability in one scheme may compromise the security of the other, and may be essential to maintain provable security. As an example, suppose a key pair is employed in both RSAES-OAEP (Section 7.1) and RSAES-PKCS1-v1_5 (Section 7.2). Although RSAES-OAEP by itself would resist attack, an opponent could exploit a weakness in the implementation of RSAES-PKCS1-v1_5 to recover messages encrypted with either scheme. As another example, suppose a key pair is employed in both RSASSA-PKCS1-v1_5 (Section 8.1) and RSASSA-PSS (Section 8.2). Then the security proof for RSASSA-PSS would no longer be sufficient since the proof does not account for the possibility the signatures might be generated with a second scheme. No vulnerability is apparent in this case but the proof of security is lost. Similar considerations may apply if a key pair is employed in one of the schemes defined here and a variant defined elsewhere.

There may be situations in which only one key pair is available and it needs to be employed in multiple schemes, e.g., an encryption and a signature scheme. In such a case, additional security evaluation is necessary. RSAES-OAEP and RSASSA-PSS are designed in a way that prevents an opponent from exploiting interactions between the schemes, so would be appropriate for such a situation. RSAES-PKCS1-v1_5 and RSASSA-PKCS1-v1_5 have traditionally been employed together (indeed, this is the model introduced by PKCS #1 v1.5), without any known bad interactions. But in general, it is prudent to limit a key pair to a single scheme and purpose.

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 [39] 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 Std 1363-2000 [25], 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 [3]. It is compatible with the IFES scheme defined in IEEE Std 1363-2000 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 $k–2–2hLen$ 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 mask generation function in EME-OAEP has appropriate properties, and the key size is sufficiently large, RSAES-OAEP is semantically secure against adaptive chosen ciphertext attacks; see the note below for further discussion. 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 [6] 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 Appendix A.2.1 for the relevant ASN.1 syntax.

Note. Recent results have helpfully clarified the security properties of the OAEP encoding method. The background is as follows. In 1994, Bellare and Rogaway [3] introduced a security concept that they denoted *plaintext awareness* (PA94). They proved that if an encryption primitive (e.g., RSAEP) is hard to invert without the private key, then the corresponding OAEP-based encryption scheme is plaintext-aware[1], meaning roughly that an adversary cannot produce a valid ciphertext without actually "knowing" the underlying plaintext. Plaintext awareness of an encryption scheme is closely related to the resistance of the scheme against *chosen ciphertext attacks*. In such attacks, an adversary is given the opportunity to send queries to an oracle simulating the decryption primitive. Using the results of these queries, the adversary attempts to decrypt a challenge ciphertext.

However, there are *two* flavors of chosen ciphertext attacks, and PA94 implies security against only one of them. The difference relies on what the adversary is allowed to do after she is given the challenge ciphertext. The *indifferent* attack scenario (denoted CCA1) does not admit any queries to the decryption oracle after the adversary is given the challenge ciphertext, whereas the *adaptive* scenario (denoted CCA2) does (except that the decryption oracle refuses to decrypt the challenge ciphertext once it is published). In 1998, Bellare and Rogaway, together with Desai and Pointcheval [2], came up with a new, stronger notion of plaintext awareness (PA98) that does imply security against CCA2.

To summarize, there have been two potential sources for misconception: that PA94 and PA98 are equivalent concepts; or that CCA1 and CCA2 are equivalent concepts. Either assumption leads to the conclusion that the Bellare-Rogaway paper implies security of OAEP against CCA2[2], which it does not. OAEP has never been proven secure against CCA2; in fact, Victor Shoup [42] ingeniously demonstrated recently that such a proof does not exist in the general case. Put briefly, Shoup showed that an adversary in the CCA2 scenario who knows how to *partially* invert the encryption primitive but does not know how to invert it *completely* may well be able to break the scheme. For example, one may imagine an attacker who is able to break RSAES-OAEP if she is able to recover all but the first 20 bytes of an integer encrypted with RSAEP. Such an attacker does not need to be able to fully invert RSAEP, because she does not use the first 20 octets in her attack.

Still, RSAES-OAEP *is* secure against CCA2, which was proved by Fujisaki, Okamoto, Pointcheval, and Stern [20] shortly after the announcement of Shoup's result. Using clever lattice reduction techniques, they managed to show how to invert RSAEP completely given a sufficiently large part of the pre-image. This observation, combined with a proof that OAEP is secure against CCA2 if the underlying encryption primitive is hard to *partially* invert, fills the gap between what Bellare and Rogaway proved about RSAES-OAEP and what some may have believed that they proved. Somewhat paradoxically, we are hence saved by an ostensible weakness in RSAEP (i.e., the whole inverse can be deduced from parts of it). As a consequence, it makes little sense replacing OAEP with a "more secure" encoding method, because if a CCA2 adversary is able to break RSAES-OAEP, then she will be able to break RSAEP equipped with any encoding method (if maybe slightly less efficiently). For encryption primitives different from RSAEP, however, it might be worthwhile considering a stronger encoding method such as OAEP+ suggested by Shoup [42].

---

[1] More precisely, plaintext-aware in the random oracle model.

[2] It might be fair to mention that PKCS #1 v2.0 cites [3] and claims that "*a chosen ciphertext attack is ineffective against a plaintext-aware encryption scheme such as RSAES-OAEP*" without specifying the kind of plaintext awareness or chosen ciphertext attack considered.

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

Errors: "message too long"

Assumptions: public key (n, e) is valid

Steps:

1. Apply the EME-OAEP encoding operation (Section 9.1.1.1) to the message M and the encoding parameters P to produce an encoded message EM of length k–1 octets:

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

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

Errors: "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 representative 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 (Section 9.1.1.2) 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 errors in steps 4 and 5 are indistinguishable, otherwise an adversary may be able to extract useful information from the type of error occurred. In particular, the error messages in steps 4 and 5 must be identical. Moreover, the execution time of the decryption operation must not reveal whether an error has occurred. One way of achieving this is as follows: In case of error in step 4, proceed to step 5 with *EM* set to a string of zero octets. Error message information is used to mount a chosen ciphertext attack on PKCS #1 v1.5 encrypted messages in [6].

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 [8]; [12] contains an improved attack).

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 [6]. Therefore, if RSAES-PKCS1-v1_5 is to be used, certain easily implemented countermeasures should be taken to thwart the attack found in [6]. Typical examples include 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. These can all be effective countermeasures and do not involve changes to a PKCS #1 v1.5-based protocol. See [7] for a further discussion of these and other countermeasures.

Note. 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. Håstad's results [23] 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. [8] (see [12] for an improvement of the attack) 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. Thus, as a general rule, the use of this scheme for encrypting an arbitrary message, as opposed to a randomly generated key, is not recommended. 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 RSAES-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*

Errors: "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:

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

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

Errors: "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:

$$c = \mathrm{OS2IP}\ (C)\ .$$

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

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

   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:

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

   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 from the type of error message received to mount a chosen ciphertext attack such as the one found in [6]. For further comments, see the note in Section 7.1.2.

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, the signature scheme with appendix defined here would be a suitable signature algorithm for X.509 certificates [27]. A related signature scheme could be employed in PKCS #7 [39], 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.

Two signature schemes with appendix are specified in this document: RSASSA-PKCS1-v1_5 and RSASSA-PSS. Although no attacks are known against RSASSA-PKCS1-v1_5, in the interest of increased robustness, RSASSA-PSS is recommended for eventual adoption in new applications. RSASSA-PKCS1-v1_5 is included for compatibility with existing applications, and while still appropriate for new applications, a gradual transition to RSASSA-PSS is encouraged.

The signature schemes with appendix given here follow a general model similar to that employed in IEEE Std 1363-2000 [25], 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. If the encoding method is deterministic (e.g., EMSA-PKCS1-v1_5), 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. If the method is randomized (e.g., EMSA-PSS), a verification operation is applied to the message and the octet string to determine whether they are consistent. (Note that this approach assumes that the signature and verification primitives have the message-recovery form. The signature generation and verification operations may have a different form for other primitives.)

For both signature schemes with appendix defined in this document, the signature generation and signature verification operations are readily implemented as “single-pass” operations if the signature is placed after the message.

8.1 RSASSA-PKCS1-v1_5

RSASSA-PKCS1-v1_5 combines the RSASP1 and RSAVP1 primitives with the EMSA-PKCS1-v1_5 encoding method. It is compatible with the IFSSA scheme defined in IEEE Std 1363-2000 [25], 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 IEEE Std 1363-2000, but is in the IEEE P1363a draft [26]). 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 EMSA-PKCS1-v1_5 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. More precisely, 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.

Notes.

1. As noted in PKCS #1 v1.5, the EMSA-PKCS1-v1_5 encoding method has the property that the encoded message, converted to an integer message representative, is guaranteed to be large and at least somewhat "random". This prevents attacks of the kind proposed by Desmedt and Odlyzko [16] where multiplicative relationships between message representatives are developed by factoring the message representatives into a set of small values (e.g., a set of small primes). Coron, Naccache, and Stern [13] showed that a stronger form of this type of attack could be quite effective against some instances of the ISO/IEC 9796-2 signature scheme. They also analyzed the complexity of this type of attack against the EMSA-PKCS1-v1_5 encoding method and concluded that an attack would be impractical, requiring more operations than a collision search on the underlying hash function (i.e., more than 2^{80} operations). Coron *et al.*'s attack was subsequently extended by Coppersmith, Halevi and Jutla [9] to break the ISO/IEC 9796-1 signature scheme with message recovery. The various attacks illustrate the importance of carefully constructing the input to the RSA signature primitive, particularly in a signature scheme with message recovery. Accordingly, the EMSA-PKCS-v1_5 encoding method explicitly includes a hash operation and is not intended for signature schemes with message recovery. Moreover, while no attack is known against the EMSA-PKCS-v1_5 encoding method, a gradual transition to EMSA-PSS is recommended as a precaution against future developments.

2. The signature generation and verification operation operations are readily implemented as "single-pass" operations if the signature is placed after the message. See PKCS #7 [39] for an example format.

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

Errors: "message too long"; "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:

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

   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" and stop.

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" or "invalid signature"

Errors: "message too long"; "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:

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

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" and stop.

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

3. Convert the signature S to an integer signature representative s:

$$s = \mathrm{OS2IP}\ (S)\ .$$

4. 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:

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

If RSAVP1 outputs "signature representative out of range," then output "invalid signature" and stop.

5. Convert the message representative m to a second encoded message EM of length k-1 octets:

$$EM' = \mathrm{I2OSP}\ (m, k-1).$$

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

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

Note. Another way to implement the signature verification operation is to apply a "decoding" operation (not specified in this document) to the encoded message to recover the underlying hash value, and then to compare it to a newly computed hash value. This has the advantage that it requires less intermediate storage (two hash values rather than two encoded messages), but the disadvantage that it requires additional code.

8.2 RSASSA-PSS

RSASSA-PSS combines the RSASP1 and RSAVP1 primitives with the EMSA-PSS encoding method. It is compatible with the IFSSA scheme as amended in the IEEE P1363a draft [26], where the signature and verification primitives are IFSP-RSA1 and IFVP-RSA1 as defined in IEEE Std 1363-2000 [25] and the message encoding method is EMSA4. EMSA4 is slightly more general than EMSA-PSS as it acts on bit strings rather than on octet strings. EMSA-PSS is equivalent to EMSA4 restricted to the case that the operands as well as the hash and salt values are octet strings.

The length of messages on which RSASSA-PSS can operate is either unrestricted or constrained by a very large number, depending on the hash function underlying the EMSA-PSS encoding method.

Assuming that computing e^{th} roots modulo n is infeasible and the hash and mask generation functions in EMSA-PSS have appropriate properties, RSASSA-PSS provides secure signatures. This assurance is provable in the sense that the difficulty of forging signatures can be directly related to the difficulty of inverting the RSA function, if the hash and mask generation functions are viewed as black boxes or random oracles.

In contrast to the RSASSA-PKCS1-v1_5 signature scheme, a hash function identifier is not embedded in the encoded message, so in theory it is possible for an adversary to substitute a different (and potentially weaker) hash function than the one selected by the signer. However, assuming that the mask generation function is based on the same hash function, the entire encoded message will be dependent on the selected hash function, not just the portion corresponding to the hash value as in EMSA-PKCS1-v1_5. An existing signature thus will not verify with a different hash function. Moreover, because there is a significant amount of verifiable structure within the encoded message, it will be difficult to forge new signatures as well. See [29] for further discussion of these points.

RSASSA-PSS is different than other RSA-based signature schemes in that it is probabilistic rather than deterministic, incorporating a randomly generated salt value. The salt value enhances the security of the scheme by affording a “tighter” security proof than deterministic alternatives such as Full Domain Hashing (FDH) (see [4] for discussion). However, the randomness is not critical to security. In situations where random generation is not possible, a fixed value or a sequence number could be employed instead, with the resulting provable security similar to that of FDH [10].

In the original RSA-PSS proposal [4], the message was hashed with the random salt as the first step of the signing process. This gave as a fringe benefit the property that one no longer needed to rely on the collision resistance of the underlying hash function — a property that is typically required in traditional signature schemes. However, in RSASSA-PSS the hash of the message rather than the message itself is hashed with the salt. Once again, one needs to rely on the collision resistance of the underlying hash function. Yet, this is not a significant issue since it is exactly the property of hash functions that has been required for many years in other signature schemes. Note moreover that if a signer can prevent collision attacks in general by introducing sufficient variability in the message being signed, the one-way property is all that is needed from the hash function.

8.2.1 Signature generation operation

RSASSA-PSS-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

Errors: "message too long;" "encoding error"

Steps:

1. Apply the EMSA-PSS encoding operation (Section 9.2.2.1) to the message M to produce an encoded message EM of length $\lceil(modBits-1)/8\rceil$ octets such that the bit length of the integer OS2IP(EM) is at most $modBits$–1, where $modBits$ is the length in bits of the modulus n:

$$EM = \text{EMSA-PSS-ENCODE}\ (M, modBits-1).$$

   Note that the octet length of EM will be one less than k if $modBits$–1 is divisible by 8 and equal to k otherwise. If the encoding operation outputs "message too long," then output "message too long" and stop. If the encoding operation outputs "encoding error," then output encoding error," and stop.

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.

8.2.2 Signature verification operation

RSASSA-PSS-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" or "invalid signature"

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:

$$s = \mathrm{OS2IP}\ (S).$$

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:

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

If RSAVP1 output "signature representative out of range," then output "invalid signature" and stop.

4. Convert the message representative m to an encoded message EM of length $emLen = \lceil (modBits-1)/8 \rceil$ octets, where $modBits$ is the length in bits of the modulus n:

$$EM = \mathrm{I2OSP}\ (m, emLen).$$

Note that $emLen$ will be one less than k if $modBits$–1 is divisible by 8 and equal to k otherwise. If I2OSP outputs "integer too large," then output "invalid signature" and stop.

5. Apply the EMSA-PSS verification operation (Section 9.2.2.2) to the message M and the encoded message EM to determine whether they are consistent:

$$Result = \text{EMSA-PSS-V}\textsc{erify}\ (M, EM, modBits-1).$$

If $Result$ = "consistent," output "signature verified." Otherwise, output "signature invalid."

9. Encoding methods

Encoding methods consist of operations that map between octet string messages and octet string encoded messages, which are converted to and from integer message representatives in the schemes. The integer message representatives are processed via the primitives. The encoding methods thus provide the connection between the schemes, which process messages, and the primitives.

Two types of encoding method are considered in this document: encoding methods for encryption and 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 an encoded message EM of a specified length; the decoding operation maps an encoded message EM back to a message. The encoding and decoding operations are inverses.

The encoded message *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 encoded messages.

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

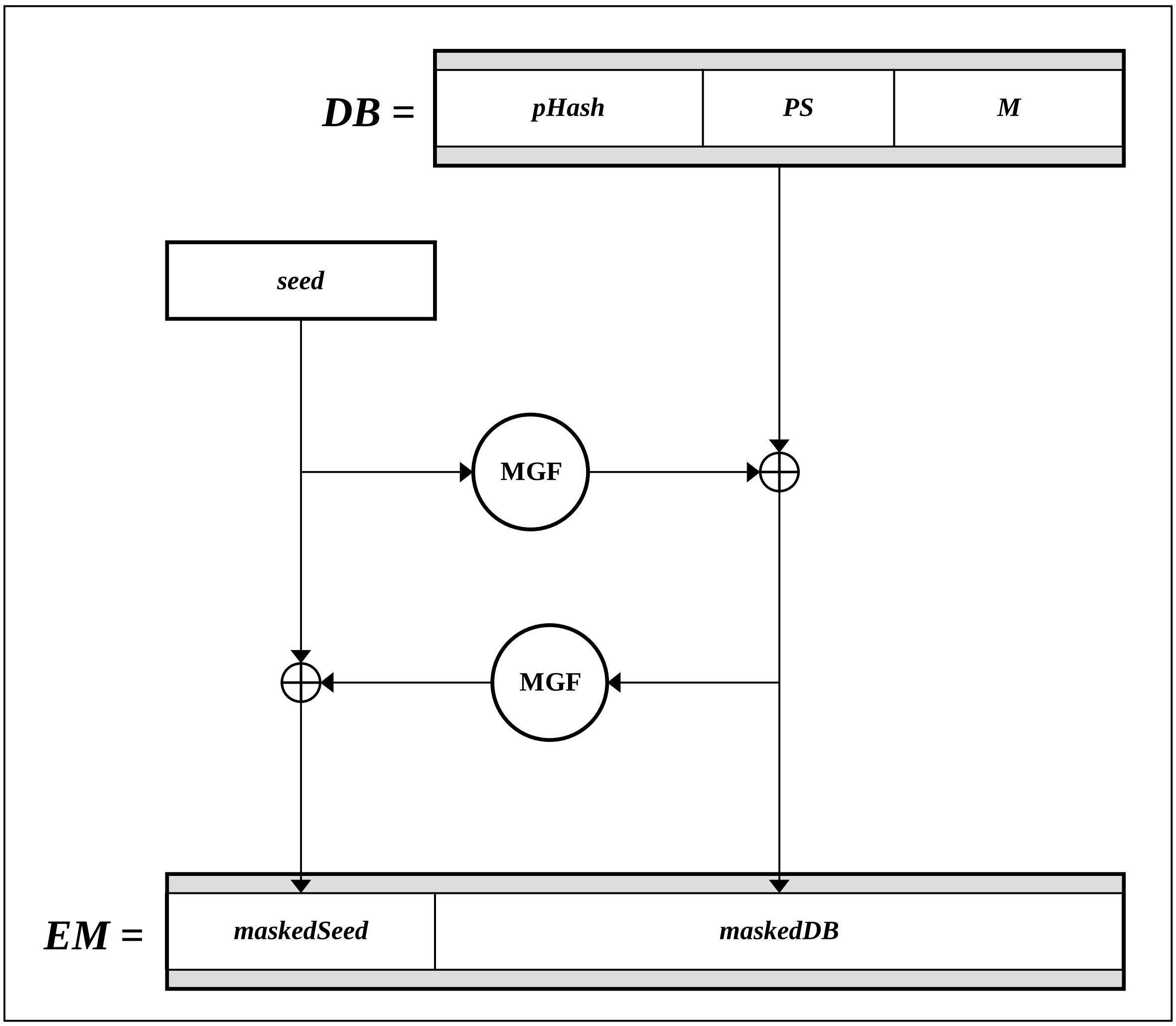

Figure 1: EME-OAEP encoding operation. *pHash* is the hash of the optional encoding parameters. Decoding operation follows reverse steps to recover *M* and verify *pHash* and *PS*.

Appendix B. The encoding method is based on Bellare and Rogaway's Optimal Asymmetric Encryption scheme [3]. (OAEP stands for "Optimal Asymmetric Encryption Padding.") It is the same as the method in IEEE Std 1363-2000 [25] except that it outputs an octet string rather than the corresponding integer, and that the length of the encoded message is expressed in octets rather than bits. Figure 1 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* (*mLen* denotes the length in octets of *M*)

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*

Errors: "message too long"; "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 *mLen* > *emLen*–2*hLen*–1, output "message too long" and stop.

3. Generate an octet string *PS* consisting of *emLen*–*mLen*–2*hLen*–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

$$DB = pHash \parallel PS \parallel 01 \parallel M \,.$$

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

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

8. Let $maskedDB = DB \oplus dbMask$.

9. Let $seedMask = MGF\ (maskedDB, hLen)$.

10. Let $maskedSeed = seed \oplus seedMask$.

11. Let $EM = maskedSeed \parallel maskedDB$.

12. Output *EM*.

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 $2hLen+1$ (*emLen* denotes the length in octets of *EM*)

P encoding parameters, an octet string

Output: *M* recovered message, an octet string of length at most $emLen-1-2hLen$

Errors: "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 $emLen < 2hLen+1$, output "decoding error" and stop.

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

4. Let $seedMask = MGF\ (maskedDB, hLen)$.

5. Let $seed = maskedSeed \oplus seedMask$.

6. Let $dbMask = MGF\ (seed, emLen-hLen)$.

7. Let $DB = maskedDB \oplus dbMask$.

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 (*mLen* denotes the length in octets of *M*)

emLen intended length in octets of the encoded message

Output: *EM* encoded message, an octet string of length *emLen*

Errors: "message too long"

Steps:

1. If *mLen* > *emLen*–10, output "message too long" and stop.

2. Generate an octet string *PS* of length *emLen*–*mLen*–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 (*emLen* denotes the length in octets of *EM*)

Output: *M* recovered message, an octet string of length at most *emLen*–10

Errors: "decoding error"

Steps:

1. If *emLen* < 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 and optionally a verification operation. An encoding operation maps a message *M* to an encoded message *EM* of a specified length. A verification operation determines whether a message *M* and an encoded message *EM* are consistent, i.e., whether the encoded message *EM* is a valid encoding of the message *M*.

The encoding operation may introduce some randomness, so that different applications of the encoding operation to the same message will produce different encoded messages, which has benefits for provable security. For such an encoding method, both an encoding and a verification operation are needed unless the verifier can reproduce the randomness (e.g., by obtaining the salt value from the signer). For a deterministic encoding method only an encoding operation is needed.

Two encoding methods for signatures with appendix are employed in the signature schemes and are specified here: EMSA-PKCS1-v1_5 and EMSA-PSS.

9.2.1 EMSA-PKCS1-v1_5

This encoding method is deterministic and 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*

Errors: "message too long"; "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" and stop.

2. Encode the algorithm ID for the hash function and the hash value into an ASN.1 value of type **DigestInfo** (see Appendix A) 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]

[3] For the six hash functions mentioned in Appendix B.1, this step is equivalent to the following:

For MD2: *T* = **30 20 30 0c 06 08 2a 86 48 86 f7 0d 02 02 05 00 04 10** || *H*.
For MD5: *T* = **30 20 30 0c 06 08 2a 86 48 86 f7 0d 02 05 05 00 04 10** || *H*.
For SHA-1: *T* = **30 21 30 09 06 05 2b 0e 03 02 1a 05 00 04 14** || *H*.
For SHA-256: *T* = **30 31 30 0d 06 09 60 86 48 01 65 03 04 02 01 05 00 04 20** || *H*.
For SHA-384: *T* = **30 41 30 0d 06 09 60 86 48 01 65 03 04 02 02 05 00 04 30** || *H*.
For SHA-512: *T* = **30 51 30 0d 06 09 60 86 48 01 65 03 04 02 03 05 00 04 40** || *H*.

3. If *emLen* < $\|T\|$+10, output "intended encoded message length too short" and stop.

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

$$EM = 01 \| PS \| 00 \| T.$$

6. Output *EM*.

9.2.2 EMSA-PSS

This encoding method is parameterized by the choice of hash function and mask generation function. Suggested hash and mask generation functions are given in Appendix B. The encoding method is based on Bellare and Rogaway's Probabilistic Signature Scheme (PSS) [4][5]. It is randomized and has an encoding operation and a verification operation. Figure 2 illustrates the encoding operation.

Notes.

1. The encoding method defined here differs from the one in Bellare and Rogaway's submission to IEEE P1363a [5] in three respects:

   - It applies a hash function to the message rather than a mask generation function. Even though the mask generation function is based on a hash function, it seems more natural to apply a hash function directly.

   - The value that is hashed together with the salt value is the string 00 00 00 00 00 00 00 00 ‖ *mHash* (where *mHash* is the hash of the actual message to be signed) rather than the message *M* itself. See Note 3 below for further discussion. (Also, the name "salt" is used instead of "seed", as it is more reflective of the value's role.)

   - The encoded message in EMSA-PSS has nine fixed bits (the first bit is 0 and the last eight bits have hexadecimal value *bc*), whereas only the first bit is fixed in the original scheme. The rationale for the octet *bc* is for compatibility with the Rabin-Williams IFSP-RW signature primitive in IEEE Std 1363-2000 [25] and the corresponding primitive in ISO/IEC 9796-2.

2. Assuming that the mask generation function is based on a hash function, it is recommended that the hash function be the same as the one that is applied to the message. In this way, the entire encoded message will be dependent on the same hash function and it will be difficult for an opponent to substitute a different hash function than the one intended by the signer (see Section 8.2 for further discussion). However, this matching of hash functions is only for the purpose of preventing hash function substitution, and is not necessary if hash function substitution is addressed by other means (e.g., the verifier accepts only a designated hash function). The provable security of RSASSA-PSS does not rely on the hash functions being the same.

3. Without compromising the security proof for EMSA-PSS, one may perform steps 1 and 2 OF EMSA-PSS-ENCODE and EMSA-PSS-VERIFY (the hash function) outside the module that computes the rest of

the signature operation, so that *mHash* rather than the message *M* itself is input to the module. In other words, the security proof for EMSA-PSS still holds even if an opponent can control the value of *mHash*. This is convenient if the module has limited I/O bandwidth, e.g., a smart card. Note that previous versions of PSS [4][5] did not have this property. Of course, it may be desirable for other security reasons to have the module process the full message. For instance, the module may need to "see" what it's signing if it doesn't trust the component that computes the hash value.

4. Typical salt lengths (in octets) are *hLen* (the length of the output of the hash function *Hash*) and 0. In both cases the security of RSASSA-PSS can be closely related to the hardness of inverting RSAVP1. Bellare and Rogaway [4] give a tight lower bound for the security of the related RSA-PSS scheme, which corresponds to the former case, while Coron [10] gives a lower bound for the related Full Domain Hashing scheme, which corresponds roughly to the latter. In [11] Coron provides a general treatment with various salt lengths ranging from 0 to *hLen*; see [26] for discussion.

5. As noted in IEEE P1363a [26], the use of randomization in signature schemes - such as the salt value in EMSA-PSS - may provide a "covert channel" for transmitting information other than the message being signed. For more on covert channels, see [44].

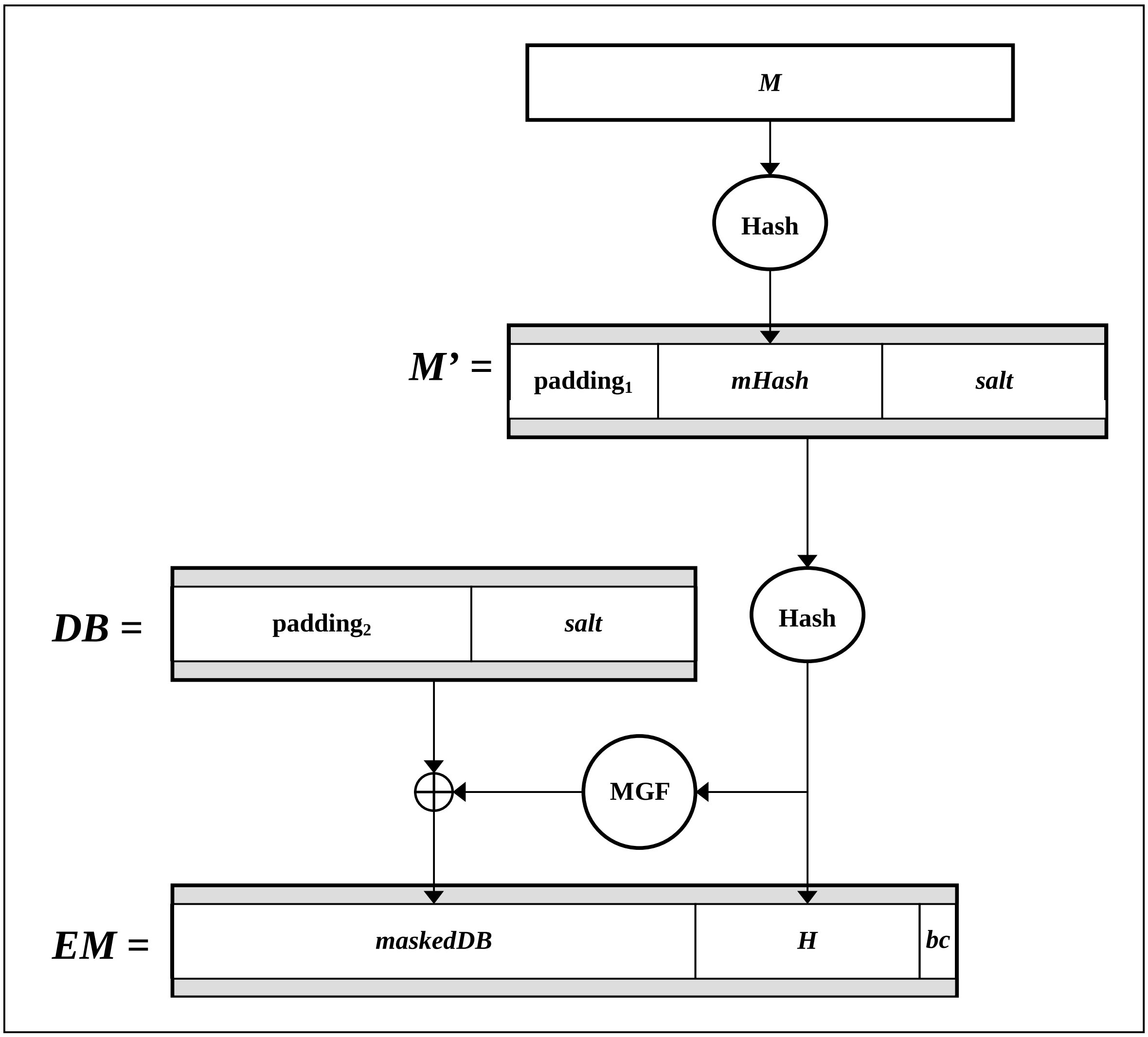

Figure 2: EMSA-PSS encoding operation. Verification operation follows reverse steps to recover *salt*, then forward steps to recompute and compare *H*.

9.2.2.1 Encoding operation

EMSA-PSS-ENCODE (*M*, *emBits*)

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

MGF mask generation function

sLen intended length in octets of the salt

Input: *M* message to be encoded, an octet string

emBits maximal bit length of the integer OS2IP(*EM*), at least 8*hLen* + 8*sLen* + 9

Output: *EM* encoded message, an octet string of length $emLen = \lceil emBits/8 \rceil$

Errors: "encoding error"; "message too long"

Steps:

1. If the length of *M* is greater than the input limitation for the hash function ($2^{61} - 1$ octets for SHA-1), then output "message too long" and stop.

2. Let *mHash* = *Hash*(*M*), an octet string of length *hLen*.

3. If *emBits* < 8*hLen* + 8*sLen* + 9, output "encoding error" and stop.

4. Generate a random octet string *salt* of length *sLen*; if *sLen* = 0, then *salt* is the empty string.

5. Let

$$M' = 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00 \parallel mHash \parallel salt;$$

M' is an octet string of length 8 + *hLen* + *sLen* with eight initial zero octets.

6. Let *H* = *Hash*(*M'*), an octet string of length *hLen*.

7. Generate an octet string *PS* consisting of *emLen* – *sLen* – *hLen* – 2 zero octets. The length of *PS* may be 0.

8. Let $DB = PS \parallel 01 \parallel salt$.

9. Let *dbMask* = *MGF*(*H*, *emLen* – *hLen* – 1).

10. Let $maskedDB = DB \oplus dbMask$.

11. Set the leftmost $8emLen - emBits$ bits of the leftmost octet in *maskedDB* to zero.

12. Let $EM = maskedDB \parallel H \parallel bc$, where *bc* is the single octet with hexadecimal value *bc*.

13. Output *EM*.

9.2.2.2 Verification operation

EMSA-PSS-VERIFY (*M*, *EM*, *emBits*)

Options:	*Hash*	hash function (*hLen* denotes the length in octets of the hash function output)
	MGF	mask generation function
	sLen	intended length in octets of the salt
Input:	*M*	message to be verified, an octet string
	EM	encoded message, an octet string of length $emLen = \lceil emBits/8 \rceil$
	emBits	maximal bit length of the integer OS2IP(*EM*), at least $8hLen + 8sLen + 9$
Output:	"consistent" or "inconsistent"	

Steps:

1. If the length of M is greater than the input limitation for the hash function ($2^{61} - 1$ octets for SHA-1), then output "inconsistent" and stop.

2. Let $mHash = Hash(M)$, an octet string of length *hLen*.

3. If $emBits < 8hLen + 8sLen + 9$, output "inconsistent" and stop.

4. If the rightmost octet of EM does not have hexadecimal value *bc*, output "inconsistent" and stop.

5. Let *maskedDB* be the leftmost $emLen - hLen - 1$ octets of *EM*, and let *H* be the next *hLen* octets.

6. If the leftmost $8emLen - emBits$ bits of the leftmost octet in *maskedDB* are not all equal to zero, output "inconsistent" and stop.

7. Let $dbMask = MGF(H, emLen - hLen - 1)$.

8. Let $DB = maskedDB \oplus dbMask$.

9. Set the leftmost $8emLen - emBits$ bits of DB to zero.

10. If the $emLen - hLen - sLen - 2$ leftmost octets of DB are not zero or if the octet at position $emLen - hLen - sLen - 1$ is not equal to 01, output "inconsistent" and stop.

11. Let *salt* be the last *sLen* octets of *DB*.

12. Let

$$M' = 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00 \parallel mHash \parallel salt;$$

    M' is an octet string of length $8 + hLen + sLen$ with eight initial zero octets.

13. Let $H' = Hash(M')$, an octet string of length *hLen*.

14. If $H = H'$, output "consistent." Otherwise, output "inconsistent."

A. ASN.1 syntax

A.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 [40], and PKCS #12 [41].

The object identifier **rsaEncryption** identifies RSA public and private keys as defined in Appendices A.1.1 and A.1.2. The **parameters** field associated with this OID in an **AlgorithmIdentifier** shall have type **NULL**.

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

The definitions in this section have been extended to support multi-prime RSA, but are backward compatible with previous versions.

A.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.

A.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
        otherPrimeInfos         OtherPrimeInfos OPTIONAL
        }

Version ::= INTEGER { two-prime(0), multi(1) }
        (CONSTRAINED BY {-- version must be multi if otherPrimeInfos are present })

OtherPrimeInfos ::= SEQUENCE SIZE(1..MAX) OF OtherPrimeInfo

OtherPrimeInfo ::= SEQUENCE {
        prime                   INTEGER, -- ri
        exponent                INTEGER, -- di
        coefficient             INTEGER  -- ti
        }
```

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, unless multi-prime is used, in which case it shall be 1.
- **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 CRT coefficient q^{-1} mod p.
- **otherPrimeInfos** contains the information for the additional primes $r_3, \ldots, r_f$, in order. It shall be omitted if **version** is 0 and shall contain at least one instance of **OtherPrimeInfo** if **version** is 1.

The fields of type **OtherPrimeInfo** have the following meanings:

- **prime** is a prime factor r_i of n, where $i \geq 3$.
- **exponent** is $d_i = d \bmod (r_i - 1)$.
- **coefficient** is the CRT coefficient $t_i = (r_1 \cdot r_2 \cdot \ldots \cdot r_{(i-1)})^{-1} \bmod r_i$.

Note. It is important to protect the private key against both disclosure and modification. Techniques for such protection are outside the scope of this document. Method for storing and distributing private keys and other cryptographic data are described in PKCS #12 and #15.

A.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.

A.2.1 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 **RSAES-OAEP-params**:

```
RSAES-OAEP-params ::= SEQUENCE {
        hashFunc        [0] AlgorithmIdentifier {{OAEP-PSSDigestAlgorithms}}
                            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 **OAEP-PSSDigestAlgorithms**. For a discussion of supported hash functions, see Appendix B.1.

```
OAEP-PSSDigestAlgorithms ALGORITHM-IDENTIFIER ::= {
        { SHAParameters IDENTIFIED BY id-sha1 } |
        { SHAParameters IDENTIFIED BY id-sha256 } |
        { SHAParameters IDENTIFIED BY id-sha384 } |
        { SHAParameters IDENTIFIED BY id-sha512 },
         ... --Allows for future expansion
        }
```

  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 Appendix B.2.1). The **parameters** field for **id-mgf1** shall be an algorithm ID with an OID in the set **OAEP-PSSDigestAlgorithms**, identifying the hash function on which MGF1 is based.

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

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}
        }
```

A.2.2 RSAES-PKCS1-v1_5

The object identifier **rsaEncryption** (Appendix A.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}
```

A.2.3 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, SHA-1, SHA-256, SHA-384, or SHA-512. 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**. The OID should be chosen in accordance with the following table:

Hash algorithm	OID
MD2	md2WithRSAEncryption ::= {pkcs-1 2}
MD5	md5WithRSAEncryption ::= {pkcs-1 4}
SHA-1	sha1WithRSAEncryption ::= {pkcs-1 5}
SHA-256	sha256WithRSAEncryption ::= {pkcs-1 11}
SHA-384	sha384WithRSAEncryption ::= {pkcs-1 12}
SHA-512	sha512WithRSAEncryption ::= {pkcs-1 13}

A.2.4 RSASSA-PSS

The object identifier **id-RSASSA-PSS** identifies the RSASSA-PSS encryption scheme.

```
id-RSASSA-PSS OBJECT IDENTIFIER ::= {pkcs-1 10}
```

The **parameters** field associated with this OID in an **AlgorithmIdentifier** shall have type **RSASSA-PSS-params:**

```
RSASSA-PSS-params ::=  SEQUENCE {
        hashFunc          [0] AlgorithmIdentifier {{OAEP-PSSDigestAlgorithms}}
                              DEFAULT sha1Identifier,
        maskGenFunc       [1] AlgorithmIdentifier {{pkcs1MGFAlgorithms}}
                              DEFAULT mgf1SHA1Identifier
        }
```

The fields of type **RSASSA-PSS-params** have the following meanings:

- **hashFunc** identifies the hash function. It shall be an algorithm ID with an OID in the set **OAEP-PSSDigestAlgorithms** (see Appendix A.2.1). The default hash function is SHA-1.

- **maskGenFunc** identifies the mask generation function. It shall be an algorithm ID with an OID in the set **pkcs1MGFAlgorithms** (see Appendix A.2.1). The default mask generation function is MGF1 with SHA-1. For MGF1 (and more generally, for other mask generation functions based on a hash function), it is recommended that the underlying hash function be the same as the one identified by **hashFunc**; see Note 2 in Section 9.2.2 for further comments.

If the default values of the **hashFunc** and **maskGenFunc** fields of **RSASSA-PSS-params** are used, then the algorithm identifier will have the following value:

```
RSASSA-PSS-Default-Identifier ::= AlgorithmIdentifier {
        id-RSASSA-PSS,
        { sha1Identifier, mgf1SHA1Identifier }
        }
```

Note. In some applications, the hash function underlying a signature scheme is identified separately from the rest of the operations in the signature scheme. For instance, in PKCS #7 [39], a hash function identifier is placed before the message and a "digest encryption" algorithm identifier (indicating the rest of the operations) is carried with the signature. In order for applications such as PKCS #7 to support the RSASSA-PSS signature scheme, an object identifier would need to be defined for the operations in RSASSA-PSS after the hash function (analogous to the **RSAEncryption** OID for the RSASSA-PKCS1-v1_5 scheme). S/MIME CMS [24] takes a different approach. Although a hash function identifier is placed before the message, an algorithm identifier for the full signature scheme may be carried with a CMS signature (this is done for DSA signatures). Following this convention, the **id-RSASSA-PSS** OID can be used to identify RSASSA-PSS signatures in CMS. Since CMS is considered the successor to PKCS #7 and new developments such as the addition of support for RSASSA-PSS will be pursued with respect to CMS rather than PKCS #7, an OID for the "rest of" RSASSA-PSS is not defined in this version of PKCS #1.

B. Supporting techniques

This section gives several examples of underlying functions supporting the encoding methods in Section 9. While these supporting techniques are appropriate for applications to implement, none of them is required to be implemented. It is expected, however, that profiles for PKCS #1 v2.1 will be developed that specify particular supporting techniques.

This section also gives object identifiers for the supporting techniques.

B.1 Hash functions

Hash functions are used in the operations contained in Sections 7 and 8. 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 and 8 should generally 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. In addition to the requirements, the hash function should yield a mask generation function (Appendix B.2) with pseudorandom output.

Six hash functions are recommended for the encoding methods in this document: MD2 [28], MD5 [35], SHA-1 [32], and the proposed algorithms SHA-256, SHA-384, and SHA-512 [33]. For the EME-OAEP and EMSA-PSS encoding methods, only SHA-1 and SHA-256/384/512 are recommended. For the EMSA-PKCS1-v1_5 encoding method, SHA-1 or SHA-256/384/512 are recommended for new applications. MD2 and MD5 are recommended only for compatibility with existing applications based on PKCS #1 v1.5.

The object identifiers **md2**, **md5**, **id-SHA1**, **id-SHA256**, **id-SHA384**, and **id-SHA512**, identify the respective hash functions:

```
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}

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

id-SHA256  OBJECT IDENTIFIER ::=
    {joint-iso-itu-t(2) country(16) us(840) organization(1) gov(101) csor(3) nistalgorithm(4)
    hashalgs(2) 1 }

id-SHA384  OBJECT IDENTIFIER ::=
```

```
        {joint-iso-itu-t(2) country(16) us(840) organization(1) gov(101) csor(3) nistalgorithm(4)
        hashalgs(2) 2 }

id-SHA512     OBJECT IDENTIFIER ::=
        {joint-iso-itu-t(2) country(16) us(840) organization(1) gov(101) csor(3) nistalgorithm(4)
        hashalgs(2) 3 }
```

The **parameters** field associated with these OIDs in an **AlgorithmIdentifier** shall have type **NULL**.

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 [17] demonstrated how to find collisions for MD4 and that the first two rounds of MD4 are not one-way [19]. Because of these results and others (e.g. [14]), 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 [37] 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. [15]).

To address these concerns, SHA-1 is recommended for new applications. Yet, while a hash function with a 160-bit output (as provided by SHA-1) is considered sufficient when the size of the RSA modulus is 1024, a hash function with a larger output (e.g., SHA-256) may be more appropriate for larger RSA keys. This is particularly true for the signature schemes defined in this document, as the complexity of collision attacks is $2^{L/2}$, where L is the bit length of the hash output. In particular, the security of RSASSA-PKCS-v1_5 or RSASSA-PSS based on SHA-1 against collision attacks will remain at the 80-bit level no matter how large the RSA modulus might be. The situation is quite different for RSA-OAEP, but it seems reasonable to base RSA-OAEP and RSASSA-PSS on the same hash function for similar RSA key sizes (if only for alignment). In addition, the time-complexity of an attack is not the only issue to consider when selecting key sizes and hash sizes; memory and cost must be considered as well. See [43] for further discussion.

B.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: Given one part of the output but not the input, it should be infeasible to predict another part of the output. The provable security of RSAES-OAEP and RSASSA-PSS 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. MGF1 coincides with the mask generation functions defined in IEEE Std 1363-2000 [25] and the draft ANSI X9.44 [1]. Future versions of this document may define other mask generation functions.

B.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
Error:	"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*.

The object identifier **id-mgf1** identifies the MGF1 mask generation function:

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

The **parameters** field associated with this OID in an **AlgorithmIdentifier** shall have type **AlgorithmIdentifier**, identifying the hash function on which MGF1 is based.

C. ASN.1 module

```
PKCS-1 {iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1) pkcs-1(1)  modules(0) pkcs-1(1)}

-- $Revision: 1.0 $

-- This module has been checked for conformance with the ASN.1 standard by the OSS ASN.1 Tools

DEFINITIONS EXPLICIT TAGS ::=

BEGIN

-- EXPORTS ALL --
-- All types and values defined in this module is exported for use in other ASN.1 modules.

IMPORTS

id-sha256, id-sha384, id-sha512
        FROM NIST-SHA2 {joint-iso-itu-t(2) country(16) us(840) organization(1) gov(101) csor(3)
        nistalgorithm(4) modules (0) sha2 (1)};

-- Basic object identifiers

-- The DER for this in hexadecimal is:
-- 06 08
--     2A 86 48 86 F7 0D 01 01
--
pkcs-1 OBJECT IDENTIFIER ::= {iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1) 1}

--
-- When the following OIDs are used in an AlgorithmIdentifier the parameters MUST be present and
-- MUST be NULL.
--
rsaEncryption               OBJECT IDENTIFIER ::= { pkcs-1 1 }
md2WithRSAEncryption        OBJECT IDENTIFIER ::= { pkcs-1 2 }
md4WithRSAEncryption        OBJECT IDENTIFIER ::= { pkcs-1 3 }
md5WithRSAEncryption        OBJECT IDENTIFIER ::= { pkcs-1 4 }
sha1WithRSAEncryption       OBJECT IDENTIFIER ::= { pkcs-1 5 }
sha256WithRSAEncryption     OBJECT IDENTIFIER ::= { pkcs-1 11 }
sha384WithRSAEncryption     OBJECT IDENTIFIER ::= { pkcs-1 12 }
sha512WithRSAEncryption     OBJECT IDENTIFIER ::= { pkcs-1 13 }

rsaOAEPEncryptionSET        OBJECT IDENTIFIER ::= { pkcs-1 6 }

--
-- When id-RSAES-OAEP is used in an AlgorithmIdentifier the parameters MUST be present and
-- MUST be RSAES-OAEP-params.
--
id-RSAES-OAEP               OBJECT IDENTIFIER ::= { pkcs-1 7 }

--
-- When id-mgf1 is used in an AlgorithmIdentifier the parameters MUST be present and MUST be
-- a DigestAlgorithmIdentifier, for example SHA1Identifier.
--
id-mgf1                     OBJECT IDENTIFIER ::= { pkcs-1 8 }

--
```

```
-- When id-pSpecified is used in an AlgorithmIdentifier the parameters MUST be an OCTET STRING.
--
id-pSpecified              OBJECT IDENTIFIER ::= { pkcs-1 9 }

--
-- When id-RSASSA-PSS is used in an AlgorithmIdentifier the parameters MUST be present and
-- MUST be RSASSA-PSS-params.
--
id-RSASSA-PSS              OBJECT IDENTIFIER ::= { pkcs-1 10 }

--
-- This OID really belongs in a module with the secsig OIDs.
--
id-sha1                    OBJECT IDENTIFIER ::=
        {iso(1) identified-organization(3) oiw(14) secsig(3) algorithms(2) 26 }

-- Useful types

ALGORITHM-IDENTIFIER ::= TYPE-IDENTIFIER

-- Note: the parameter InfoObjectSet in the following definitions allows a distinct information object
-- set to be specified for sets of algorithms such as:
-- DigestAlgorithms         ALGORITHM-IDENTIFIER ::= {
--      { NULL IDENTIFIED BY id-md2 },
--      { NULL IDENTIFIED BY id-md5 },
--      { NULL IDENTIFIED BY id-sha1 }
--      }
--

AlgorithmIdentifier { ALGORITHM-IDENTIFIER:InfoObjectSet } ::= SEQUENCE {
        algorithm       ALGORITHM-IDENTIFIER.&id({InfoObjectSet}),
        parameters      ALGORITHM-IDENTIFIER.&Type({InfoObjectSet} {@algorithm}) OPTIONAL
        }

-- Algorithms

--
-- Allowed OAEP digest algorithms.
--
OAEP-PSSDigestAlgorithms  ALGORITHM-IDENTIFIER ::= {
        { SHAParameters IDENTIFIED BY id-sha1 }      |
        { SHAParameters IDENTIFIED BY id-sha256 }    |
        { SHAParameters IDENTIFIED BY id-sha384 }    |
        { SHAParameters IDENTIFIED BY id-sha512 },
         ... -- Allows for future expansion
        }

sha1Identifier AlgorithmIdentifier {{ OAEP-PSSDigestAlgorithms }} ::=
        {algorithm id-sha1, parameters SHAParameters : NULL}

SHAParameters ::= NULL

--
-- Allowed Mask Generation Function algorithms.
-- If the identifier is id-mgf1, the parameters
-- are a single digest algorithm identifier.
--
PKCS1MGFAlgorithms         ALGORITHM-IDENTIFIER ::= {
```

```
        { MGF1Parameters IDENTIFIED BY id-mgf1 },
        ...-- Allows for future expansion
        }

MGF1Parameters ::= AlgorithmIdentifier { {OAEP-PSSDigestAlgorithms} }

--
-- Allowed algorithms for pSourceFunc.
--
PKCS1PSourceAlgorithms    ALGORITHM-IDENTIFIER ::= {
        { PEmptyString IDENTIFIED BY id-pSpecified },
        ...-- Allows for future expansion
        }

PEmptyString ::= OCTET STRING (SIZE(0))

--
-- This identifier means that P is an empty string, so the digest
-- of the empty string appears in the RSA block before masking.
--
pSpecifiedEmptyIdentifier AlgorithmIdentifier {{ PKCS1PSourceAlgorithms }} ::=
        { algorithm id-pSpecified, parameters PEmptyString : ''H }

--
-- Default AlgorithmIdentifier for id-RSAES-OAEP.maskGenFunc.
--
mgf1SHA1Identifier AlgorithmIdentifier {{ PKCS1MGFAlgorithms }} ::=
        {algorithm id-mgf1,
        parameters AlgorithmIdentifier{{OAEP-PSSDigestAlgorithms}} : sha1Identifier}

--
-- Type identifier definitions for the PKCS #1 OIDs.
--
PKCS1Algorithms               ALGORITHM-IDENTIFIER ::= {
        { NULL IDENTIFIED BY rsaEncryption }                   |
        { NULL IDENTIFIED BY md2WithRSAEncryption }            |
        { NULL IDENTIFIED BY md4WithRSAEncryption }            |
        { NULL IDENTIFIED BY md5WithRSAEncryption }            |
        { NULL IDENTIFIED BY sha1WithRSAEncryption }           |
        { NULL IDENTIFIED BY sha256WithRSAEncryption }         |
        { NULL IDENTIFIED BY sha384WithRSAEncryption }         |
        { NULL IDENTIFIED BY sha512WithRSAEncryption }         |
        { NULL IDENTIFIED BY rsaOAEPEncryptionSET }            |
        { RSAES-OAEP-params IDENTIFIED BY id-RSAES-OAEP }      |
        PKCS1PSourceAlgorithms                                 |
        { RSASSA-PSS-params IDENTIFIED BY id-RSASSA-PSS },
        ... -- Allows for future expansion
        }

-- Main structures

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

--
-- Representation of RSA private key with
```

```
-- information for the CRT algorithm.
--
RSAPrivateKey ::= SEQUENCE {
        version                 Version,
        modulus                 INTEGER, -- (Usually large) n
        publicExponent          INTEGER, -- (Usually small) e
        privateExponent         INTEGER, -- (Usually large) d
        prime1                  INTEGER, -- (Usually large) p
        prime2                  INTEGER, -- (Usually large) q
        exponent1               INTEGER, -- (Usually large) d mod (p-1)
        exponent2               INTEGER, -- (Usually large) d mod (q-1)
        coefficient             INTEGER, -- (Usually large) (inverse of q) mod p
        otherPrimeInfos         OtherPrimeInfos OPTIONAL
        }

Version ::= INTEGER { two-prime(0), multi(1) }
        (CONSTRAINED BY {-- version must be multi if otherPrimeInfos present --})

OtherPrimeInfos ::= SEQUENCE SIZE(1..MAX) OF OtherPrimeInfo

OtherPrimeInfo ::= SEQUENCE {
        prime INTEGER,  -- ri
        exponent INTEGER, -- di
        coefficient INTEGER -- ti
        }

--
-- AlgorithmIdentifier.parameters for id-RSAES-OAEP.
-- Note that the tags in this Sequence are explicit.
--
RSAES-OAEP-params ::=  SEQUENCE {
        hashFunc        [0] AlgorithmIdentifier { {OAEP-PSSDigestAlgorithms} }
                            DEFAULT sha1Identifier,
        maskGenFunc     [1] AlgorithmIdentifier { {PKCS1MGFAlgorithms} }
                            DEFAULT mgf1SHA1Identifier,
        pSourceFunc     [2] AlgorithmIdentifier { {PKCS1PSourceAlgorithms} }
                            DEFAULT pSpecifiedEmptyIdentifier
        }

--
-- Identifier for default RSAES-OAEP algorithm identifier
-- The DER Encoding of this is in hexadecimal:
-- 30 0D
--    06 09
--       2A 86 48 86 F7 0D 01 01 07
--    30 00
--       Notice that the DER encoding of default values is "empty".
--

rSAES-OAEP-Default-Identifier AlgorithmIdentifier{ {PKCS1Algorithms} } ::=
        {algorithm id-RSAES-OAEP,
        parameters RSAES-OAEP-params : {
                hashFunc        sha1Identifier,
                maskGenFunc     mgf1SHA1Identifier,
                pSourceFunc     pSpecifiedEmptyIdentifier
                }
        }
```

```
--
-- AlgorithmIdentifier.parameters for id-RSASSA-PSS.
-- Note that the tags in this Sequence are explicit.
--
RSASSA-PSS-params ::=  SEQUENCE {
        hashFunc        [0] AlgorithmIdentifier {{OAEP-PSSDigestAlgorithms}}
                            DEFAULT sha1Identifier,
        maskGenFunc     [1] AlgorithmIdentifier {{PKCS1MGFAlgorithms}}
                            DEFAULT mgf1SHA1Identifier
        }

--
-- Identifier for default RSASSA-PSS algorithm identifier
-- The DER Encoding of this is in hexadecimal:
-- 30 0D
--    06 09
--       2A 86 48 86 F7 0D 01 01 0A
--    30 00
--       Notice that the DER encoding of default values is "empty".
--
rSASSA-PSS-Default-Identifier AlgorithmIdentifier{ {PKCS1Algorithms} } ::=
        {algorithm id-RSASSA-PSS,
        parameters RSASSA-PSS-params : {
                hashFunc        sha1Identifier,
                maskGenFunc     mgf1SHA1Identifier
                }
        }

END -- PKCS1Definitions
```

D. Intellectual property considerations

The RSA public-key cryptosystem is described in U.S. Patent 4,405,829, which expired on September 20, 2000. RSA Security Inc. makes no other patent claims on the constructions described in this document, although specific underlying techniques may be covered.

Multi-prime RSA is described in U.S. Patent 5,848,159.

The University of California has indicated that it has a patent pending on the PSS signature scheme [5]. It has also provided a letter to the IEEE P1363 working group stating that if the PSS signature scheme is included in an IEEE standard, "*the University of California will, when that standard is adopted, FREELY license any conforming implementation of PSS as a technique for achieving a digital signature with appendix.*" [22].

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.

E. 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 1.5 was republished as IETF RFC 2313.

Version 2.0

Version 2.0 incorporated major editorial changes in terms of the document structure and introduced the RSAES-OAEP encryption scheme. This version continued to support the encryption and signature processes in version 1.5, although the hash algorithm MD4 was no longer allowed due to cryptanalytic advances in the intervening years. Version 2.0 was republished as IETF RFC 2437 [30].

Version 2.1

Version 2.1 introduces multi-prime RSA and the RSASSA-PSS signature scheme with appendix along with several editorial improvements. This version continues to support the schemes in version 2.0.

F. References

[1] ANSI X9F1 Working Group. *ANSI X9.44: Key Establishment Using Factoring-Based Public Key Cryptography for the Financial Services Industry*. Working Draft, June 2000.

[2] M. Bellare, A. Desai, D. Pointcheval and P. Rogaway. Relations among Notions of Security for Public-Key Encryption Schemes. In *Advances in Cryptology – Crypto'98, pp. 26-45*. Springer Verlag, 1998.

[3] 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.

[4] 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.

[5] M. Bellare and P. Rogaway. *PSS: Provably Secure Encoding Method for Digital Signatures*. Submission to IEEE P1363 working group, August 1998. Available from http://grouper.ieee.org/groups/1363/.

[6] D. Bleichenbacher. Chosen Ciphertext Attacks against Protocols Based on the RSA Encryption Standard PKCS #1. In *Advances in Cryptology – Crypto '98*, pp. 1-12. Springer Verlag, 1998.

[7] D. Bleichenbacher, B. Kaliski and J. Staddon. *Recent Results on PKCS #1: RSA Encryption Standard*. RSA Laboratories' Bulletin No. 7, June 24, 1998.

[8] 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.

[9] D. Coppersmith, S. Halevi and C. Jutla. *ISO 9796-1 and the New Forgery Strategy*. Presented at the rump session of Crypto '99, August 17, 1999.

[10] J.-S. Coron. On the Exact Security of Full Domain Hashing. In *Advances in Cryptology – Crypto 2000*, pp. 229–235. Springer Verlag, 2000.

[11] J.-S. Coron. *Security of PSS for Short Random Size*. Manuscript, July 2000.

[12] J.-S. Coron, M. Joye, D. Naccache and P. Paillier. New Attacks on PKCS #1 v1.5 Encryption. In *Advances in Cryptology – Eurocrypt 2000*, pp. 369-379. Springer Verlag, 2000.

[13] J.-S. Coron, D. Naccache and J. P. Stern. On the Security of RSA Padding. In *Advances in Cryptology – Crypto '99*, pp. 1-18. Springer Verlag, 1999.

[14] 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.

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

[16] Y. Desmedt and A.M. Odlyzko. A Chosen Text Attack on the RSA Cryptosystem and Some Discrete Logarithm Schemes. In *Advances in Cryptology – Crypto '85*, pp. 516-521. Springer Verlag, 1986.

[17] H. Dobbertin. Cryptanalysis of MD4. In *Fast Software Encryption '96*, pp. 55-72. Springer Verlag, 1996.

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

[19] H. Dobbertin. The First Two Rounds of MD4 are Not One-Way. In *Fast Software Encryption '98*, pp. 284-292. Springer Verlag, 1998.

[20] E. Fujisaki, T. Okamoto, D. Pointcheval and J. Stern. *RSA-OAEP Is Still Alive!* Preprint, November 2000. Available from http://eprint.iacr.org/.

[21] H. Garner. The Residue Number System. *IRE Transactions on Electronic Computers*, EC-8 (6), pp. 140-147, June 1959.

[22] M.L. Grell. *Re: Encoding Methods PSS/PSS-R.* Letter to IEEE P1363 working group, University of California, June 15, 1999. Available from http://grouper.ieee.org/groups/1363/P1363/patents.html.

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

[24] R. Housley. *IETF RFC 2630: Cryptographic Message Syntax.* June 1999.

[25] *IEEE Std 1363-2000: Standard Specifications for Public Key Cryptography*. IEEE, August 2000.

[26] IEEE P1363 working group. *IEEE P1363a D6: Standard Specifications for Public Key Cryptography: Additional Techniques.* November 2000. Available from http://grouper.ieee.org/groups/1363/P1363a/draft.html.

[27] *ISO/IEC 9594-8:1997: Information technology – Open Systems Interconnection – The Directory: Authentication framework*. 1997.

[28] B. Kaliski. *IETF RFC 1319: The MD2 Message-Digest Algorithm*. April 1992.

[29] B. Kaliski. *Hash Function Firewalls in Signature Schemes*. Presentation to IEEE P1363 working group, June 2000. Available from http://grouper.ieee.org/groups/1363/Research/Presentations.html.

[30] B. Kaliski and J. Staddon. *IETF RFC 2437: PKCS #1: RSA Cryptography Specifications Version 2.0.* October 1998.

[31] A. Menezes, P. van Oorschot and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

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

[33] National Institute of Standards and Technology (NIST). *Descriptions of SHA-256, SHA-384, and SHA-512*. October 2000. Available from http://www.nist.gov/sha/.

[34] J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for RSA public-key cryptosystem. *Electronics Letters*, 18(21), pp. 905–907, October 14, 1982

[35] R. Rivest. *IETF RFC 1321: The MD5 Message-Digest Algorithm*. April 1992.

[36] 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.

[37] 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.

[38] RSA Laboratories. *PKCS #1 v2.0: RSA Encryption Standard*. October 1, 1998.

[39] RSA Laboratories. *PKCS #7 v1.5: Cryptographic Message Syntax Standard*. November 1993.

[40] RSA Laboratories. *PKCS #8 v1.2: Private-Key Information Syntax Standard*. November 1993.

[41] RSA Laboratories. *PKCS #12 v1.0: Personal Information Exchange Syntax Standard*. June 24, 1999.

[42] V. Shoup. *OAEP Reconsidered.* Preprint, November 2000. Available from http://eprint.iacr.org/.

[43] R. D. Silverman. *A Cost-Based Security Analysis of Symmetric and Asymmetric Key Lengths.* RSA Laboratories Bulletin No. 13, April 2000. Available from http://www.rsasecurity.com.rsalabs/bulletins/.

[44] G. J. Simmons. Subliminal communication is easy using the DSA. In *Advances in Cryptology – Eurocrypt '93*, pp. 218-232. Springer-Verlag, 1993.

G. About PKCS

The *Public-Key Cryptography Standards* are specifications produced by RSA Laboratories in cooperation with secure systems developers worldwide for the purpose of accelerating the deployment of public-key cryptography. First published in 1991 as a result of meetings with a small group of early adopters of public-key technology, the PKCS documents have become widely referenced and implemented. Contributions from the PKCS series have become part of many formal and *de facto* standards, including ANSI X9 and IEEE P1363 documents, PKIX, SET, S/MIME, SSL/TLS, and WAP/WTLS.

Further development of PKCS occurs through mailing list discussions and occasional workshops, and suggestions for improvement are welcome. For more information, contact:

PKCS Editor
RSA Laboratories
20 Crosby Drive
Bedford, MA 01730 USA
pkcs-editor@rsasecurity.com
http://www.rsasecurity.com/rsalabs/pkcs