RSA LABORATORIES

PKCS #1 v2.1: RSA Cryptography Standard

RSA Laboratories

June 14, 2002

Table of Contents

003-100000-210-001-000

1 Introduction

This document provides recommendations for the implementation of public-key cryptography based on the RSA algorithm [42], 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 the standard IEEE-1363-2000 [26] and draft standards currently being developed by the ANSI X9F1 [1] and IEEE P1363 [27] working groups.

This document supersedes PKCS #1 version 2.0 [44] but includes compatible techniques.

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 signature schemes in Section 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

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

C ciphertext, an octet string

d RSA private exponent

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

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

dP	*p*'s CRT exponent, a positive integer such that $e \cdot dP \equiv 1 \pmod{(p-1)}$
dQ	*q*'s CRT exponent, a positive integer such that $e \cdot dQ \equiv 1 \pmod{(q-1)}$
e	RSA public exponent
EM	encoded message, an octet string
emBits	(intended) length in bits of an encoded message *EM*
emLen	(intended) length in octets of an encoded message *EM*
GCD (. , .)	greatest common divisor of two nonnegative integers
Hash	hash function
hLen	output length in octets of hash function Hash
k	length in octets of the RSA modulus *n*
K	RSA private key
L	optional RSAES-OAEP label, an 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	MGF output, an octet string
maskLen	(intended) length of the octet string *mask*
MGF	mask generation function
mgfSeed	seed from which *mask* is generated, an octet string
mLen	length in octets of a message *M*
n	RSA modulus, $n = r_1 \cdot r_2 \cdot \ldots \cdot r_u$, $u \geq 2$
(*n*, *e*)	RSA public key
p, *q*	first two prime factors of the RSA modulus *n*

qInv	CRT coefficient, a positive integer less than p such that $q \cdot qInv \equiv 1 \pmod{p}$
r_i	prime factors of the RSA modulus n, 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
sLen	length in octets of the EMSA-PSS salt
t_i	additional prime factor r_i's CRT coefficient, a positive integer less than r_i such that $r_1 \cdot r_2 \cdot \ldots \cdot r_{i-1} \cdot t_i \equiv 1 \pmod{r_i}$, $i = 3, \ldots, u$
u	number of prime factors of the RSA modulus, $u \geq 2$
x	a nonnegative integer
X	an octet string corresponding to x
xLen	(intended) length of the octet string X
0x	indicator of hexadecimal representation of an octet or an octet string; "0x48" denotes the octet with hexadecimal value 48; "(0x)48 09 0e" denotes the string of three consecutive octets with hexadecimal value 48, 09, and 0e, respectively
$\lambda(n)$	LCM $(r_1 - 1, r_2 - 1, \ldots, r_u - 1)$
$\oplus$	bit-wise exclusive-or of two octet strings
$\lceil \cdot \rceil$	ceiling function; $\lceil x \rceil$ is the smallest integer larger than or equal to the real number x
$\|$	concatenation operator
$\equiv$	congruence symbol; $a \equiv b \pmod{n}$ means that the integer n divides the integer $a - b$

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 [22] 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 [49].

3.1 RSA public key

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

n the RSA modulus, a positive integer

e the RSA public exponent, a positive integer

In a *valid RSA public key*, the RSA modulus *n* is a product of *u* distinct odd primes r_i, $i = 1, 2, \ldots, u$, where $u \geq 2$, and the RSA public exponent *e* is an integer between 3 and $n - 1$ satisfying GCD $(e, \lambda(n)) = 1$, where $\lambda(n) = \text{LCM}(r_1 - 1, \ldots, r_u - 1)$. 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 RSA modulus, a positive integer

d the RSA private exponent, a positive 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), $i = 3, \ldots, u$, one for each prime not in the quintuple, where the components have the following meanings:

p	the first factor, a positive integer
q	the second factor, a positive integer
dP	the first factor's CRT exponent, a positive integer
dQ	the second factor's CRT exponent, a positive integer
$qInv$	the (first) CRT coefficient, a positive integer
r_i	the i^{th} factor, a positive integer
d_i	the i^{th} factor's CRT exponent, a positive integer
t_i	the i^{th} factor's CRT coefficient, a positive integer

In a *valid RSA private key* with the first representation, the RSA modulus n is the same as in the corresponding RSA public key and is the product of u distinct odd primes r_i, $i = 1, 2, \ldots, u$, where $u \geq 2$. The RSA private exponent d is a positive integer less than n satisfying

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

where e is the corresponding RSA public exponent and $\lambda(n)$ is defined as in Section 3.1.

In a valid RSA private key with the second representation, the two factors p and q are the *first two* prime factors of the RSA modulus n (i.e., r_1 and r_2), the CRT 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 $u > 2$, the representation will include one or more triplets (r_i, d_i, t_i), $i = 3, \ldots, u$. The factors r_i are the additional prime factors of the RSA modulus n. Each CRT exponent d_i ($i = 3, \ldots, u$) satisfies

$$e \cdot d_i \equiv 1 \pmod{(r_i - 1)} .$$

Each CRT coefficient t_i ($i = 3, \ldots, u$) 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 follow Garner's algorithm [22] (see also Algorithm 14.71 in [37]). 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 [40] 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*, *xLen*)

Input: *x* nonnegative integer to be converted

xLen intended length of the resulting octet string

Output: *X* corresponding octet string of length *xLen*

Error: "integer too large"

Steps:

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

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

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

   where $0 \leq x_i < 256$ (note that one or more leading digits will be zero if x is less than 256^{xLen-1}).

3. Let the octet X_i have the integer value x_{xLen-i} for $1 \leq i \leq xLen$. Output the octet string

$$X = X_1\, X_2\, \ldots\, X_{xLen}.$$

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 $X_1\, X_2\, \ldots\, X_{xLen}$ be the octets of X from first to last, and let x_{xLen-i} be the integer value of the octet X_i for $1 \leq i \leq xLen$.

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

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 RSA 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 [26] (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 $n - 1$

Error: "message representative out of range"

Assumption: RSA 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), $i = 3, \ldots, u$

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

Output: *m* message representative, an integer between 0 and $n - 1$

Error: "ciphertext representative out of range"

Assumption: RSA 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. The message representative *m* is computed as follows.
   a. If the first form (n, d) of *K* is used, let $m = c^d \bmod n$.
   b. If the second form $(p, q, dP, dQ, qInv)$ and (r_i, d_i, t_i) of *K* is used, proceed as follows:
      i. Let $m_1 = c^{dP} \bmod p$ and $m_2 = c^{dQ} \bmod q$.
      ii. If $u > 2$, let $m_i = c^{d_i} \bmod r_i$, $i = 3, \ldots, u$.
      iii. Let $h = (m_1 - m_2) \cdot qInv \bmod p$.
      iv. Let $m = m_2 + q \cdot h$.
      v. If $u > 2$, let $R = r_1$ and for $i = 3$ to u do
         1. Let $R = R \cdot r_{i-1}$.
         2. Let $h = (m_i - m) \cdot t_i \pmod{r_i}$.
         3. Let $m = m + R \cdot h$.
3. Output *m*.

Note. Step 2.a 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 [26] (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), $i = 3, \ldots, u$

m message representative, an integer between 0 and $n - 1$

Output: *s* signature representative, an integer between 0 and $n - 1$

Error: "message representative out of range"

Assumption: RSA 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. The signature representative *s* is computed as follows.

    a. If the first form (*n*, *d*) of *K* is used, let $s = m^d \bmod n$.

    b. If the second form (*p*, *q*, *dP*, *dQ*, *qInv*) and (r_i, d_i, t_i) of *K* is used, proceed as follows:

i. Let $s_1 = m^{dP} \bmod p$ and $s_2 = m^{dQ} \bmod q$.

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

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

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

v. If $u > 2$, let $R = r_1$ and for $i = 3$ to u do

1. Let $R = R \cdot r_{i-1}$.
2. Let $h = (s_i - s) \cdot t_i \pmod{r_i}$.
3. Let $s = s + R \cdot h$.

3. Output s.

Note. Step 2.a 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.

5.2.2 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 $n - 1$

Error: "signature representative out of range"

Assumption: RSA 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 an RSA public or private 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 RSA 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 RSA 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 RSA 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. While RSAES-PKCS1-v1_5 (Section 7.2) and RSASSA-PKCS1-v1_5 (Section 8.2) have traditionally been employed together without any known bad interactions (indeed, this is the model introduced by PKCS #1 v1.5), such a combined use of an RSA key pair is not recommended for new applications.

To illustrate the risks related to the employment of an RSA key pair in more than one scheme, suppose an RSA key pair is employed in both RSAES-OAEP (Section 7.1) and RSAES-PKCS1-v1_5. Although RSAES-OAEP by itself would resist attack, an opponent might be able to exploit a weakness in the implementation of RSAES-PKCS1-v1_5 to recover messages encrypted with either scheme. As another example, suppose an RSA key pair is employed in both RSASSA-PSS (Section 8.1) and RSASSA-PKCS1-v1_5. Then the security proof for RSASSA-PSS would no longer be sufficient since the proof does not account for the possibility that signatures might be generated with a second scheme. Similar considerations may apply if an RSA key pair is employed in one of the schemes defined here and in a variant defined elsewhere.

7 Encryption schemes

For the purposes of this document, 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 RSA public key, and the decryption

operation recovers the message from the ciphertext with the recipient's corresponding RSA 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 [45] 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 [26], 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.

To avoid implementation weaknesses related to the way errors are handled within the decoding operation (see [6] and [36]), the encoding and decoding operations for RSAES-OAEP and RSAES-PKCS1-v1_5 are embedded in the specifications of the respective encryption schemes rather than defined in separate specifications. Both encryption schemes are compatible with the corresponding schemes in PKCS #1 v2.0.

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 (step 1.b in Section 7.1.1 and step 3 in Section 7.1.2). EME-OAEP is based on Bellare and Rogaway's Optimal Asymmetric Encryption scheme [3]. (OAEP stands for "Optimal Asymmetric Encryption Padding."). It is compatible with the IFES scheme defined in IEEE Std 1363-2000 [26], 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 $k - 2hLen - 2$ octets, where *hLen* is the length of the output from the underlying hash function and k is the length in octets of the recipient's RSA modulus.

Assuming that computing e^{th} roots modulo n is infeasible and the mask generation function in RSAES-OAEP has appropriate properties, RSAES-OAEP is semantically secure against adaptive chosen-ciphertext attacks. This assurance is provable in the sense

that the difficulty of breaking RSAES-OAEP can be directly related to the difficulty of inverting the RSA function, provided that the mask generation function is viewed as a black box or random oracle; see [21] and the note below for further discussion.

Both the encryption and the decryption operations of RSAES-OAEP take the value of a label *L* as input. In this version of PKCS #1, *L* is the empty string; other uses of the label are outside the scope of this document. See Appendix A.2.1 for the relevant ASN.1 syntax.

RSAES-OAEP is parameterized by the choice of hash function and mask generation function. This choice should be fixed for a given RSA key. Suggested hash and mask generation functions are given in Appendix B.

Note. Recent results have helpfully clarified the security properties of the OAEP encoding method [3] (roughly the procedure described in step 1.b in Section 7.1.1). 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 a deterministic public-key encryption primitive (e.g., RSAEP) is hard to invert without the private key, then the corresponding OAEP-based encryption scheme is plaintext-aware (in the random oracle model), 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, which it does not.[1] OAEP has never been proven secure against CCA2; in fact, Victor Shoup [48] has demonstrated 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 knows how to recover all but the first 20 bytes of a random 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 [21] 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

[1] 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.

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).

Unfortunately however, the security reduction is not efficient for concrete parameters. While the proof successfully relates an adversary *A* against the CCA2 security of RSAES-OAEP to an algorithm *I* inverting RSA, the probability of success for *I* is only approximately $\varepsilon^2 / 2^{18}$, where ε is the probability of success for *A*.[1] In addition, the running time for *I* is approximately t^2, where *t* is the running time of the adversary. The consequence is that we cannot exclude the possibility that attacking RSAES-OAEP is considerably easier than inverting RSA for concrete parameters. Still, the existence of a security proof provides some assurance that the RSAES-OAEP construction is sounder than *ad hoc* constructions such as RSAES-PKCS1-v1_5.

Hybrid encryption schemes based on the RSA-KEM key encapsulation paradigm offer tight proofs of security directly applicable to concrete parameters; see [30] for discussion. Future versions of PKCS #1 may specify schemes based on this paradigm.

7.1.1 Encryption operation

RSAES-OAEP-ENCRYPT ((*n*, *e*), *M*, *L*)

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

Input:
- (*n*, *e*) — recipient's RSA public key (*k* denotes the length in octets of the RSA modulus *n*)
- *M* — message to be encrypted, an octet string of length *mLen*, where $mLen \le k - 2hLen - 2$
- *L* — optional label to be associated with the message; the default value for *L*, if *L* is not provided, is the empty string

Output:
- *C* — ciphertext, an octet string of length *k*

Errors: "message too long"; "label too long"

Assumption: RSA public key (*n*, *e*) is valid

Steps:

---

[1] In [21] the probability of success for the inverter was $\varepsilon^2 / 4$. The additional factor $1 / 2^{16}$ is due to the eight fixed zero bits at the beginning of the encoded message *EM*, which are not present in the variant of OAEP considered in [21] (*I* must apply *A* twice to invert RSA, and each application corresponds to a factor $1 / 2^8$).

1. *Length checking*:

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

    b. If $mLen > k - 2hLen - 2$, output "message too long" and stop.

2. *EME-OAEP encoding* (see Figure 1 below):

    a. If the label L is not provided, let L be the empty string. Let $lHash$ = Hash (L), an octet string of length $hLen$ (see the note below).

    b. Generate an octet string PS consisting of $k - mLen - 2hLen - 2$ zero octets. The length of PS may be zero.

    c. Concatenate $lHash$, PS, a single octet with hexadecimal value 0x01, and the message M to form a data block DB of length $k - hLen - 1$ octets as

$$DB = lHash \| PS \| 0x01 \| M .$$

    d. Generate a random octet string *seed* of length $hLen$.

    e. Let $dbMask$ = MGF ($seed$, $k - hLen - 1$)

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

    g. Let $seedMask$ = MGF ($maskedDB$, $hLen$).

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

    i. Concatenate a single octet with hexadecimal value 0x00, $maskedSeed$, and $maskedDB$ to form an encoded message EM of length k octets as

$$EM = 0x00 \| maskedSeed \| maskedDB.$$

3. *RSA encryption*:

    a. Convert the encoded message EM to an integer message representative m (see Section 4.2):

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

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

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

c. Convert the ciphertext representative c to a ciphertext C of length k octets (see Section 4.1):

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

4. Output the ciphertext C.

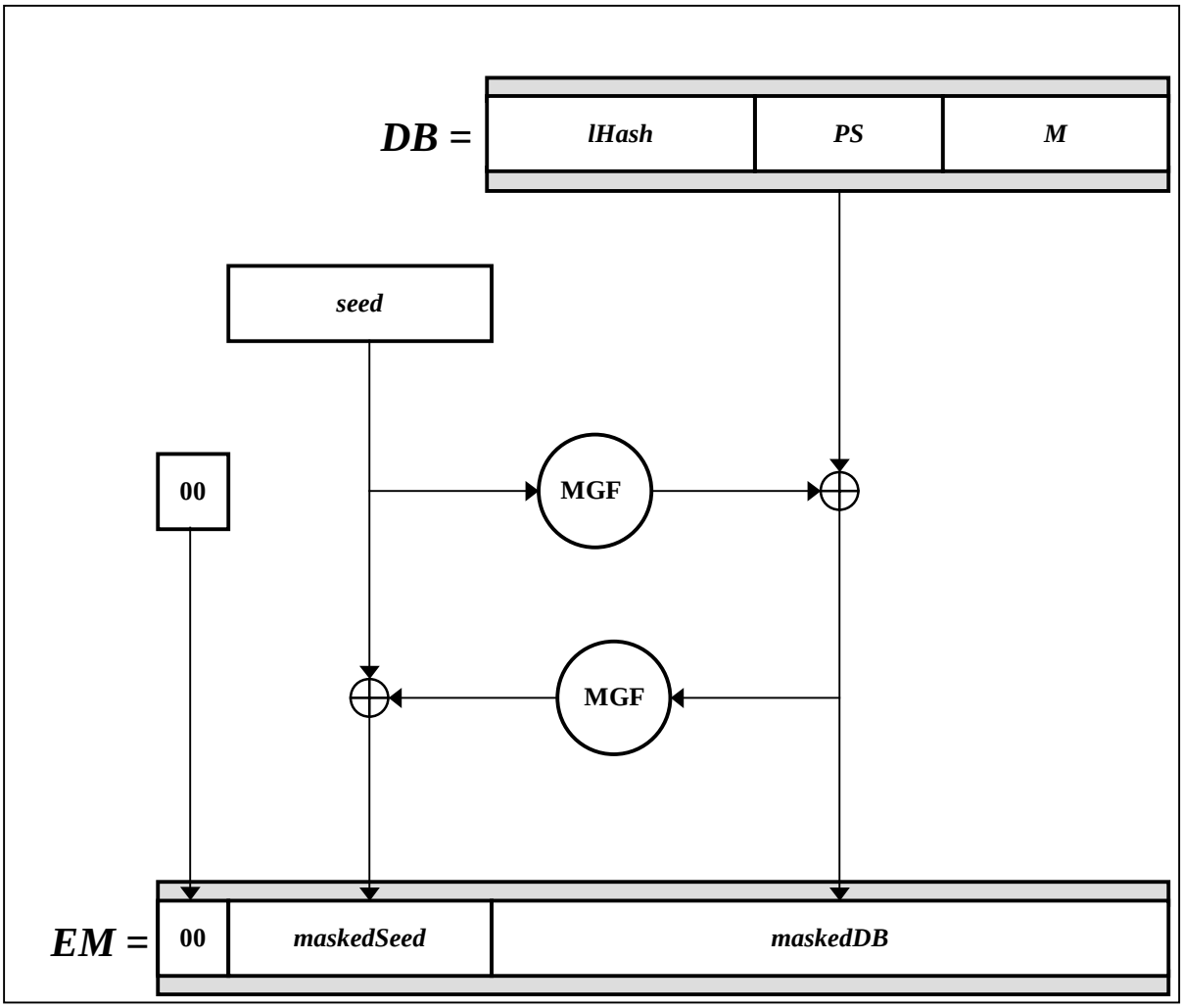

Figure 1: EME-OAEP encoding operation. *lHash* is the hash of the optional label *L*. Decoding operation follows reverse steps to recover *M* and verify *lHash* and *PS*.

Note. If *L* is the empty string, the corresponding hash value *lHash* has the following hexadecimal representation for different choices of Hash:

SHA-1: `(0x)da39a3ee 5e6b4b0d 3255bfef 95601890 afd80709`

SHA-256: `(0x)e3b0c442 98fc1c14 9afbf4c8 996fb924 27ae41e4 649b934c a495991b 7852b855`

SHA-384: `(0x)38b060a7 51ac9638 4cd9327e b1b1e36a 21fdb711 14be0743 4c0cc7bf 63f6e1da 274edebf e76f65fb d51ad2f1 4898b95b`

SHA-512: `(0x)cf83e135 7eefb8bd f1542850 d66d8007 d620e405 0b5715dc 83f4a921 d36ce9ce 47d0d13c 5d85f2b0 ff8318d2 877eec2f 63b931bd 47417a81 a538327a f927da3e`

7.1.2 Decryption operation

RSAES-OAEP-DECRYPT (*K*, *C*, *L*)

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

MGF mask generation function

Input: *K* recipient's RSA private key (*k* denotes the length in octets of the RSA modulus *n*)

C ciphertext to be decrypted, an octet string of length *k*, where $k \geq 2hLen + 2$

L optional label whose association with the message is to be verified; the default value for *L*, if *L* is not provided, is the empty string

Output: *M* message, an octet string of length *mLen*, where $mLen \leq k - 2hLen - 2$

Error: "decryption error"

Steps:

1. *Length checking*:

   a. If the length of *L* is greater than the input limitation for the hash function ($2^{61} - 1$ octets for SHA-1), output "decryption error" and stop.

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

   c. If $k < 2hLen + 2$, output "decryption error" and stop.

2. *RSA decryption*:

   a. Convert the ciphertext *C* to an integer ciphertext representative *c* (see Section 4.2):

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

   b. Apply the RSADP decryption primitive (Section 5.1.2) to the RSA 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" (meaning that $c \geq n$), output "decryption error" and stop.

c. Convert the message representative m to an encoded message EM of length k octets (see Section 4.1):

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

3. *EME-OAEP decoding*:

a. If the label L is not provided, let L be the empty string. Let $lHash = \text{Hash}(L)$, an octet string of length $hLen$ (see the note in Section 7.1.1).

b. Separate the encoded message EM into a single octet Y, an octet string $maskedSeed$ of length $hLen$, and an octet string $maskedDB$ of length $k - hLen - 1$ as

$$EM = Y \| maskedSeed \| maskedDB .$$

c. Let $seedMask = \text{MGF}(maskedDB, hLen)$.

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

e. Let $dbMask = \text{MGF}(seed, k - hLen - 1)$.

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

g. Separate DB into an octet string $lHash'$ of length $hLen$, a (possibly empty) padding string PS consisting of octets with hexadecimal value 0x00, and a message M as

$$DB = lHash' \| PS \| 0x01 \| M .$$

If there is no octet with hexadecimal value 0x01 to separate PS from M, if $lHash$ does not equal $lHash'$, or if Y is nonzero, output "decryption error" and stop. (See the note below.)

4. Output the message M.

Note. Care must be taken to ensure that an opponent cannot distinguish the different error conditions in Step 3.f, whether by error message or timing, or, more generally, learn partial information about the encoded message EM. Otherwise an opponent may be able to obtain useful information about the decryption of the ciphertext C, leading to a chosen-ciphertext attack such as the one observed by Manger [36].

7.2 RSAES-PKCS1-v1_5

RSAES-PKCS1-v1_5 combines the RSAEP and RSADP primitives (Sections 5.1.1 and 5.1.2) with the EME-PKCS1-v1_5 encoding method (step 1 in Section 7.2.1 and step 3 in Section 7.2.2). It is mathematically equivalent to the encryption scheme in PKCS #1 v1.5. RSAES-PKCS1-v1_5 can operate on messages of length up to $k - 11$ octets (k is the octet length of the RSA modulus), although care should be taken to avoid certain attacks on low-exponent RSA due to Coppersmith, Franklin, Patarin, and Reiter when long messages are encrypted (see the third bullet in the notes below and [10]; [14] contains an improved attack). As a general rule, the use of this scheme for encrypting an arbitrary message, as opposed to a randomly generated key, is not recommended.

It is possible to generate valid RSAES-PKCS1-v1_5 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. It has recently been shown that the security of the SSL/TLS handshake protocol [17], which uses RSAES-PKCS1-v1_5 and certain countermeasures, can be related to a variant of the RSA problem; see [32] for discussion.

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 step 2 in Section 7.2.1 be generated independently for each encryption process, especially if the same data is input to more than one encryption process. Håstad's results [24] are one motivation for this recommendation.

- The padding string *PS* in step 2 in Section 7.2.1 is at least eight octets long, which is a security condition for public-key operations that makes it difficult for an attacker to recover data by trying all possible encryption blocks.

- The pseudorandom octets can also help thwart an attack due to Coppersmith et al. [10] (see [14] 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 RSA public key. More specifically, in one flavor of the attack, when two inputs to RSAEP agree on a large fraction of bits (8/9) and low-exponent RSA ($e = 3$) is used to encrypt both of them, it may be possible to recover both inputs with the attack. Another flavor of the attack is successful in decrypting a single ciphertext when a large fraction (2/3) of the input to RSAEP is already known. For typical applications, the message to be encrypted is short (e.g., a 128-bit symmetric key) so not enough information will be known or common between two messages to enable the attack. However, if a long message is encrypted, or if part of a message is known, then the attack may be a concern. In any case, the 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 (*k* denotes the length in octets of the modulus *n*)

M message to be encrypted, an octet string of length *mLen*, where $mLen \le k - 11$

Output: *C* ciphertext, an octet string of length *k*

Error: "message too long"

Steps:

1. *Length checking*: If $mLen > k - 11$, output "message too long" and stop.
2. *EME-PKCS1-v1_5 encoding*:
   a. Generate an octet string *PS* of length $k - mLen - 3$ consisting of pseudo-randomly generated nonzero octets. The length of *PS* will be at least eight octets.
   b. Concatenate *PS*, the message *M*, and other padding to form an encoded message *EM* of length *k* octets as
   $$EM = 0x00 \,||\, 0x02 \,||\, PS \,||\, 0x00 \,||\, M .$$
3. *RSA encryption*:
   a. Convert the encoded message *EM* to an integer message representative *m* (see Section 4.2):
   $$m = \text{OS2IP}\ (EM) .$$
   b. Apply the RSAEP encryption primitive (Section 5.1.1) to the RSA public key (*n*, *e*) and the message representative *m* to produce an integer ciphertext representative *c*:
   $$c = \text{RSAEP}\ ((n, e), m) .$$
   c. Convert the ciphertext representative *c* to a ciphertext *C* of length *k* octets (see Section 4.1):

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

4. 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 RSA modulus n

Output: M message, an octet string of length at most $k - 11$

Error: "decryption error"

Steps:

1. *Length checking*: If the length of the ciphertext C is not k octets (or if $k < 11$), output "decryption error" and stop.

2. *RSA decryption*:

   a. Convert the ciphertext C to an integer ciphertext representative c (see Section 4.2):

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

   b. Apply the RSADP decryption primitive (Section 5.1.2) to the RSA private key (n, d) and the ciphertext representative c to produce an integer message representative m:

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

   If RSADP outputs "ciphertext representative out of range" (meaning that $c \geq n$), output "decryption error" and stop.

   c. Convert the message representative m to an encoded message EM of length k octets (see Section 4.1):

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

3. *EME-PKCS1-v1_5 decoding*: Separate the encoded message EM into an octet string PS consisting of nonzero octets and a message M as

   $$EM = 0\text{x}00 \parallel 0\text{x}02 \parallel PS \parallel 0\text{x}00 \parallel M\ .$$

If the first octet of *EM* does not have hexadecimal value 0x00, if the second octet of *EM* does not have hexadecimal value 0x02, if there is no octet with hexadecimal value 0x00 to separate *PS* from *M*, or if the length of *PS* is less than 8 octets, output "decryption error" and stop. (See the note below.)

4. Output *M*.

Note. Care shall be taken to ensure that an opponent cannot distinguish the different error conditions in Step 3, whether by error message or timing. Otherwise an opponent may be able to obtain useful information about the decryption of the ciphertext *C*, leading to a strengthened version of Bleichenbacher's attack [6]; compare to Manger's attack [36].

8 Signature schemes with appendix

For the purposes of this document, 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 RSA private key, and the signature verification operation verifies the signature on the message with the signer's corresponding RSA 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 schemes with appendix defined here would be suitable signature algorithms for X.509 certificates [28]. Related signature schemes could be employed in PKCS #7 [45], 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; see the note in Appendix A.2.3 for more discussion.

Two signature schemes with appendix are specified in this document: RSASSA-PSS and RSASSA-PKCS1-v1_5. 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 [26], 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 applied to the message representative to produce the signature. Reversing this, 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 encoded message. A verification

operation is applied to the message and the encoded message to determine whether they are consistent.

If the encoding method is deterministic (e.g., EMSA-PKCS1-v1_5), the verification operation may apply the message encoding operation to the message and compare the resulting encoded message to the previously derived encoded message. If there is a match, the signature is considered valid. If the method is randomized (e.g., EMSA-PSS), the verification operation is typically more complicated. For example, the verification operation in EMSA-PSS extracts the random salt and a hash output from the encoded message and checks whether the hash output, the salt, and the message are consistent; the hash output is a deterministic function in terms of the message and the salt.

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. See PKCS #7 [45] for an example format in the case of RSASSA-PKCS1-v1_5.

8.1 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 [27], where the signature and verification primitives are IFSP-RSA1 and IFVP-RSA1 as defined in IEEE Std 1363-2000 [26] 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, provided that the hash and mask generation functions are viewed as black boxes or random oracles. The bounds in the security proof are essentially "tight", meaning that the success probability and running time for the best forger against RSASSA-PSS are very close to the corresponding parameters for the best RSA inversion algorithm; see [4][13][31] for further discussion.

In contrast to the RSASSA-PKCS1-v1_5 signature scheme, a hash function identifier is not embedded in the EMSA-PSS 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. Therefore, it is recommended that the EMSA-PSS mask generation

function be based on the same hash function. In this manner the entire encoded message will be dependent on the hash function and it will be difficult for an opponent to substitute a different hash function than the one intended by the signer. 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). See [34] for further discussion of these points. The provable security of RSASSA-PSS does not rely on the hash function in the mask generation function being the same as the hash function applied to the message.

RSASSA-PSS is different from 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 [12].

8.1.1 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 RSA modulus *n*

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

Steps:

1. *EMSA-PSS encoding*: Apply the EMSA-PSS encoding operation (Section 9.1.1) 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*) (see Section 4.2) is at most $modBits - 1$, where *modBits* is the length in bits of the RSA 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," output "message too long" and stop. If the encoding operation outputs "encoding error," output "encoding error" and stop.

2. *RSA signature*:

a. Convert the encoded message *EM* to an integer message representative *m* (see Section 4.2):

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

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

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

c. Convert the signature representative *s* to a signature *S* of length *k* octets (see Section 4.1):

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

3. Output the signature *S*.

8.1.2 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 RSA modulus *n*

Output: "valid signature" or "invalid signature"

Steps:

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

2. *RSA verification*:

   a. Convert the signature *S* to an integer signature representative *s* (see Section 4.2):

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

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

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

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

c. 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 RSA modulus n (see Section 4.1):

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

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," output "invalid signature" and stop.

3. *EMSA-PSS verification*: Apply the EMSA-PSS verification operation (Section 9.1.2) to the message M and the encoded message EM to determine whether they are consistent:

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

4. If *Result* = "consistent," output "valid signature." Otherwise, output "invalid signature."

8.2 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 [26], 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 [27]).

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 computing e^{th} roots modulo n is infeasible and the hash function in EMSA-PKCS1-v1_5 has appropriate properties, RSASSA-PKCS1-v1_5 is conjectured to provide secure signatures. More precisely, forging signatures without knowing the RSA private key is conjectured to be computationally infeasible. Also, in the encoding method EMSA-PKCS1-v1_5, a hash function identifier is embedded in the encoding. Because of this feature, an adversary trying to find a message with the same signature as a previously signed message must 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. See [34] for further discussion.

Note. 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 [15] 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). Coppersmith, Halevi, and Jutla [11] subsequently extended Coron *et al.*'s attack 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.

8.2.1 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 RSA modulus *n*

Errors: "message too long"; "RSA modulus too short"

Steps:

1. *EMSA-PKCS1-v1_5 encoding*: Apply the EMSA-PKCS1-v1_5 encoding operation (Section 9.2) to the message *M* to produce an encoded message *EM* of length *k* octets:

   $$EM = \text{EMSA-PKCS1-V1\_5-ENCODE}\ (M, k)\ .$$

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

2. *RSA signature*:

   a. Convert the encoded message *EM* to an integer message representative *m* (see Section 4.2):

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

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

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

c. Convert the signature representative *s* to a signature *S* of length *k* octets (see Section 4.1):

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

3. Output the signature *S*.

8.2.2 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 RSA modulus *n*

Output: "valid signature" or "invalid signature"

Errors: "message too long"; "RSA modulus too short"

Steps:

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

2. *RSA verification*:

   a. Convert the signature *S* to an integer signature representative *s* (see Section 4.2):

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

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

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

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

c. Convert the message representative *m* to an encoded message *EM* of length *k* octets (see Section 4.1):

$$EM' = \text{I2OSP}\ (m, k)\ .$$

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

3. *EMSA-PKCS1-v1_5 encoding*: Apply the EMSA-PKCS1-v1_5 encoding operation (Section 9.2) to the message *M* to produce a second encoded message *EM'* of length *k* octets:

$$EM' = \text{EMSA-PKCS1-V1\_5-ENCODE}\ (M, k)\ .$$

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

4. 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.

9 Encoding methods for signatures with appendix

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.

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-PSS and EMSA-PKCS1-v1_5.

9.1 EMSA-PSS

This encoding method is parameterized by the choice of hash function, mask generation function, and salt length. These options should be fixed for a given RSA key, except that the salt length can be variable (see [31] for discussion). 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.

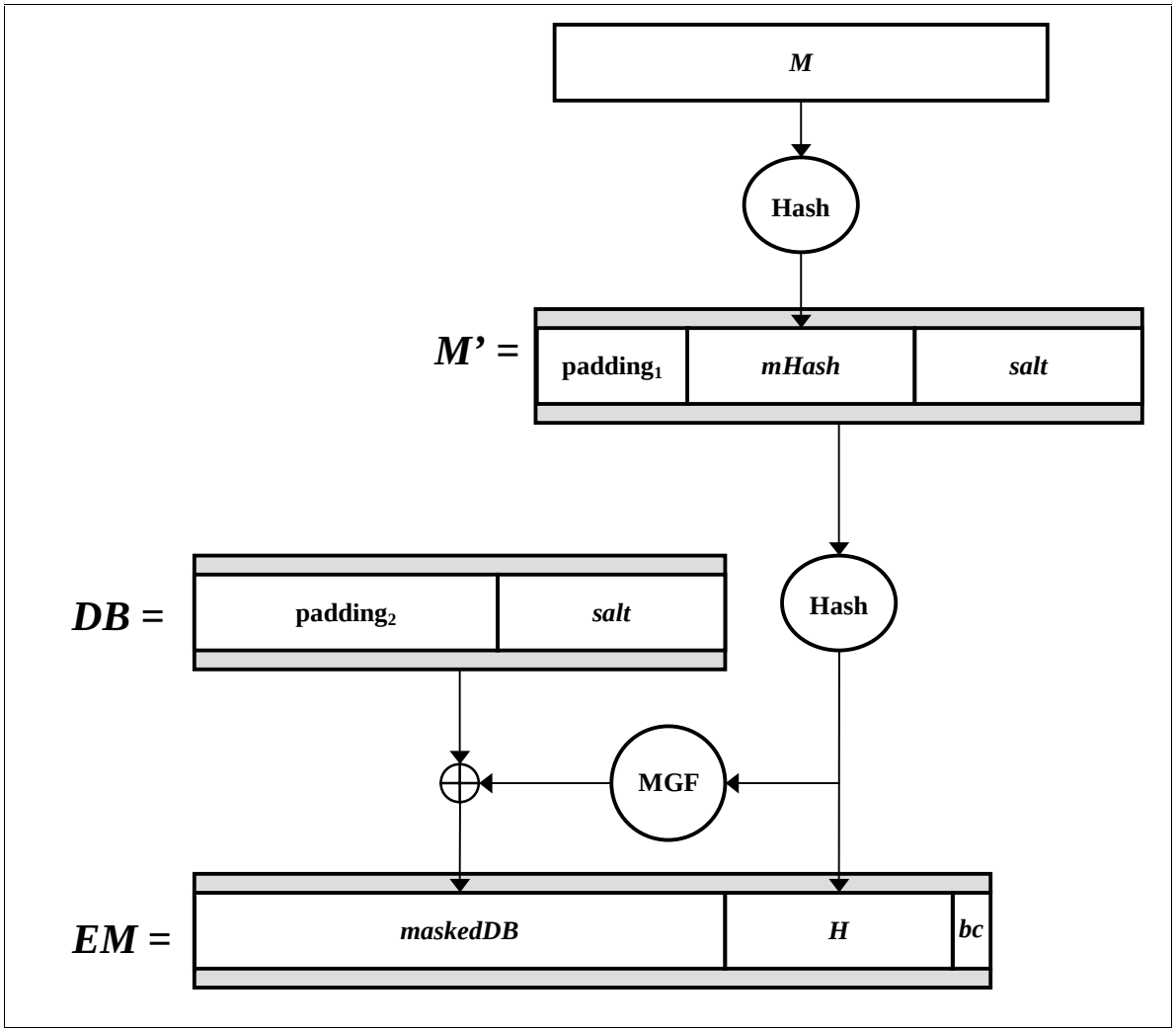

Figure 2: EMSA-PSS encoding operation. Verification operation follows reverse steps to recover *salt*, then forward steps to recompute and compare *H*.

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 rather than a mask generation function to the message. 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 (0x)00 00 00 00 00 00 00 00 || *mHash* rather than the message *M* itself. Here, *mHash* is the hash of *M*. Note that the hash function is the same in both steps. 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 form a "trailer field", the octet 0xbc. In the original scheme, only the first bit is fixed. The rationale for the trailer field is for compatibility with the Rabin-Williams IFSP-RW signature primitive in IEEE Std 1363-2000 [26] and the corresponding primitive in the draft ISO/IEC 9796-2 [29].

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; see Section 8.1 for further discussion.

3. Without compromising the security proof for RSASSA-PSS, one may perform steps 1 and 2 of EMSA-PSS-ENCODE and EMSA-PSS-VERIFY (the application of the hash function to the message) 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 RSASSA-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 is signing if it does not 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 original RSA-PSS scheme, which corresponds roughly to the former case, while Coron [12] gives a lower bound for the related Full Domain Hashing scheme, which corresponds roughly to the latter case. In [13] Coron provides a general treatment with various salt lengths ranging from 0 to *hLen*; see [27] for discussion. See also [31], which adapts the security proofs in [4][13] to address the differences between the original and the present version of RSA-PSS as listed in Note 1 above.

5. As noted in IEEE P1363a [27], 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 [50].

9.1.1 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*) (see Section 4.2), at least $8hLen + 8sLen + 9$

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), output "message too long" and stop.
2. Let *mHash* = Hash (*M*), an octet string of length *hLen*.
3. If $emLen < hLen + sLen + 2$, 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

   $$M' = (0x)00\ 00\ 00\ 00\ 00\ 00\ 00\ 00 \| mHash \| salt;$$

   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 $DB = PS \| 0x01 \| salt$; *DB* is an octet string of length $emLen - hLen - 1$.
9. Let *dbMask* = MGF (*H*, $emLen - hLen - 1$).
10. Let $maskedDB = DB \oplus dbMask$.

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

12. Let *EM* = *maskedDB* || *H* || 0xbc.

13. Output *EM*.

9.1.2 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*) (see Section 4.2), at least 8*hLen* + 8*sLen* + 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), output "inconsistent" and stop.

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

3. If *emLen* < *hLen* + *sLen* + 2, output "inconsistent" and stop.

4. If the rightmost octet of *EM* does not have hexadecimal value 0xbc, 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 8*emLen* – *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* ⊕ *dbMask*.

9. Set the leftmost 8*emLen – emBits* bits of the leftmost octet in *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 (the leftmost position is "position 1") does not have hexadecimal value 0x01, output "inconsistent" and stop.

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

12. Let

$$M' = (0x)00\ 00\ 00\ 00\ 00\ 00\ 00\ 00 \| mHash \| salt ;$$

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."

9.2 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 *tLen* + 11, where *tLen* is the octet length of the DER encoding *T* 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*:

$$H = \text{Hash}(M) .$$

If the hash function outputs "message too long," 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.2.4) 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 of the **DigestInfo** value (see the notes below) and let *tLen* be the length in octets of *T*.

3. If *emLen* < *tLen* + 11, output "intended encoded message length too short" and stop.

4. Generate an octet string *PS* consisting of *emLen* – *tLen* – 3 octets with hexadecimal value 0xff. 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

$$EM = 0x00 \,||\, 0x01 \,||\, PS \,||\, 0x00 \,||\, T\ .$$

6. Output *EM*.

Notes.

1. For the six hash functions mentioned in Appendix B.1, the DER encoding *T* of the **DigestInfo** value is equal to the following:

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

2. In version 1.5 of this document, *T* was defined as the BER encoding, rather than the DER encoding, of the **DigestInfo** value. In particular, it is possible – at least in theory – that the verification operation defined in this document (as well as in version 2.0) rejects a signature that is valid with respect to the specification given in PKCS #1 v1.5. This occurs if other rules than DER are applied to **DigestInfo** (e.g., an indefinite length encoding of the underlying **SEQUENCE** type). While this is unlikely to be a concern in practice, a cautious implementer may choose to employ a verification operation based on a BER decoding operation as specified in PKCS #1 v1.5. In this manner, compatibility with any valid implementation based on PKCS #1 v1.5 is obtained. Such a verification operation should indicate whether the underlying BER encoding is a DER encoding and hence whether the signature is valid with respect to the specification given in this document.

A. ASN.1 syntax

A.1 RSA 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 [46], and PKCS #12 [47].

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 a value of type **AlgorithmIdentifier** shall have a value of 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 RSA public key syntax

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

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

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

- **modulus** is the RSA modulus *n*.
- **publicExponent** is the RSA public exponent *e*.

A.1.2 RSA private key syntax

An RSA private key should be represented with the 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
}
```

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.

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

- **modulus** is the RSA modulus n.
- **publicExponent** is the RSA public exponent e.
- **privateExponent** is the RSA private exponent d.
- **prime1** is the prime factor p of n.
- **prime2** is the prime factor q of n.
- **exponent1** is $d \bmod (p - 1)$.
- **exponent2** is $d \bmod (q - 1)$.
- **coefficient** is the CRT coefficient $q^{-1} \bmod p$.
- **otherPrimeInfos** contains the information for the additional primes $r_3, \ldots, r_u$, in order. It shall be omitted if **version** is 0 and shall contain at least one instance of **OtherPrimeInfo** if **version** is 1.

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

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

  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 RSA private key against both disclosure and modification. Techniques for such protection are outside the scope of this document. Methods 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.

Here are type identifier definitions for the PKCS #1 OIDs:

```
PKCS1Algorithms    ALGORITHM-IDENTIFIER ::= {
    { OID rsaEncryption              PARAMETERS NULL } |
    { OID md2WithRSAEncryption       PARAMETERS NULL } |
    { OID md5WithRSAEncryption       PARAMETERS NULL } |
    { OID sha1WithRSAEncryption      PARAMETERS NULL } |
    { OID sha256WithRSAEncryption    PARAMETERS NULL } |
    { OID sha384WithRSAEncryption    PARAMETERS NULL } |
    { OID sha512WithRSAEncryption    PARAMETERS NULL } |
    { OID id-RSAES-OAEP PARAMETERS RSAES-OAEP-params } |
    PKCS1PSourceAlgorithms                           |
    { OID id-RSASSA-PSS PARAMETERS RSASSA-PSS-params } ,
    ...  -- Allows for future expansion --
}
```

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 a value of type **AlgorithmIdentifier** shall have a value of type **RSAES-OAEP-params**:

```
RSAES-OAEP-params ::= SEQUENCE {
    hashAlgorithm      [0] HashAlgorithm     DEFAULT sha1,
    maskGenAlgorithm   [1] MaskGenAlgorithm  DEFAULT mgf1SHA1,
    pSourceAlgorithm   [2] PSourceAlgorithm  DEFAULT pSpecifiedEmpty
}
```

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

- **hashAlgorithm** 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.

  ```
  HashAlgorithm ::= AlgorithmIdentifier { {OAEP-PSSDigestAlgorithms} }

  OAEP-PSSDigestAlgorithms    ALGORITHM-IDENTIFIER ::= {
      { OID id-sha1 PARAMETERS NULL   }|
      { OID id-sha256 PARAMETERS NULL }|
      { OID id-sha384 PARAMETERS NULL }|
      { OID id-sha512 PARAMETERS NULL },
      ...  -- Allows for future expansion --
  }
  ```

  The default hash function is SHA-1:

```
sha1    HashAlgorithm ::= {
    algorithm   id-sha1,
    parameters  SHA1Parameters : NULL
}

SHA1Parameters ::= NULL
```

- **maskGenAlgorithm** 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 associated with **id-mgf1** shall be an algorithm ID with an OID in the set **OAEP-PSSDigestAlgorithms**, identifying the hash function on which MGF1 is based.

```
MaskGenAlgorithm ::= AlgorithmIdentifier { {PKCS1MGFAlgorithms} }

PKCS1MGFAlgorithms    ALGORITHM-IDENTIFIER ::= {
    { OID id-mgf1 PARAMETERS HashAlgorithm },
    ...  -- Allows for future expansion --
}
```

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

```
mgf1SHA1    MaskGenAlgorithm ::= {
    algorithm   id-mgf1,
    parameters  HashAlgorithm : sha1
}
```

- **pSourceAlgorithm** identifies the source (and possibly the value) of the label *L*. 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 label is specified explicitly. The **parameters** field associated with **id-pSpecified** shall have a value of type **OCTET STRING**, containing the label. In previous versions of this specification, the term "encoding parameters" was used rather than "label", hence the name of the type below.

```
PSourceAlgorithm ::= AlgorithmIdentifier { {PKCS1PSourceAlgorithms} }

PKCS1PSourceAlgorithms    ALGORITHM-IDENTIFIER ::= {
    { OID id-pSpecified PARAMETERS EncodingParameters },
    ...  -- Allows for future expansion --
}

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

EncodingParameters ::= OCTET STRING(SIZE(0..MAX))
```

  The default label is an empty string (so that *lHash* will contain the hash of the empty string):

```
pSpecifiedEmpty    PSourceAlgorithm ::= {
    algorithm   id-pSpecified,
    parameters  EncodingParameters : emptyString
}

emptyString    EncodingParameters ::= ''H
```

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    RSAES-AlgorithmIdentifier ::= {
    algorithm   id-RSAES-OAEP,
    parameters  RSAES-OAEP-params : {
        hashAlgorithm       sha1,
        maskGenAlgorithm    mgf1SHA1,
        pSourceAlgorithm    pSpecifiedEmpty
    }
}

RSAES-AlgorithmIdentifier ::= AlgorithmIdentifier { {PKCS1Algorithms} }
```

A.2.2 RSAES-PKCS1-v1_5

The object identifier **rsaEncryption** (see Appendix A.1) identifies the RSAES-PKCS1-v1_5 encryption scheme. The **parameters** field associated with this OID in a value of type **AlgorithmIdentifier** shall have a value of type **NULL**. This is the same as in PKCS #1 v1.5.

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

A.2.3 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 a value of type **AlgorithmIdentifier** shall have a value of type **RSASSA-PSS-params:**

```
RSASSA-PSS-params ::= SEQUENCE {
    hashAlgorithm      [0] HashAlgorithm      DEFAULT sha1,
    maskGenAlgorithm   [1] MaskGenAlgorithm   DEFAULT mgf1SHA1,
    saltLength         [2] INTEGER            DEFAULT 20,
    trailerField       [3] TrailerField       DEFAULT trailerFieldBC
}
```

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

- **hashAlgorithm** 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.

- **maskGenAlgorithm** 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 **hashAlgorithm**; see Note 2 in Section 9.1 for further comments.

- **saltLength** is the octet length of the salt. It shall be an integer. For a given **hashAlgorithm**, the default value of **saltLength** is the octet length of the hash value. Unlike the other fields of type **RSASSA-PSS-params**, **saltLength** does not need to be fixed for a given RSA key pair.

- **trailerField** is the trailer field number, for compatibility with the draft IEEE P1363a [27]. It shall be 1 for this version of the document, which represents the trailer field with hexadecimal value 0xbc. Other trailer fields (including the trailer field *HashID* || 0xcc in IEEE P1363a) are not supported in this document.

```
TrailerField ::= INTEGER { trailerFieldBC(1) }
```

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

```
rSASSA-PSS-Default-Identifier    RSASSA-AlgorithmIdentifier ::= {
    algorithm   id-RSASSA-PSS,
    parameters  RSASSA-PSS-params : {
        hashAlgorithm       sha1,
        maskGenAlgorithm    mgf1SHA1,
        saltLength          20,
        trailerField        trailerFieldBC
    }
}

RSASSA-AlgorithmIdentifier ::= AlgorithmIdentifier { {PKCS1Algorithms} }
```

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 [45], 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 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 [25] 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.

A.2.4 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 a value of type **AlgorithmIdentifier** shall have a value of 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}`

The EMSA-PKCS1-v1_5 encoding method includes an ASN.1 value of type **DigestInfo**, where the type **DigestInfo** has the syntax

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

digestAlgorithm identifies the hash function and shall be an algorithm ID with an OID in the set **PKCS1-v1-5DigestAlgorithms**. For a discussion of supported hash functions, see Appendix B.1.

```
DigestAlgorithm ::= AlgorithmIdentifier { {PKCS1-v1-5DigestAlgorithms} }

PKCS1-v1-5DigestAlgorithms    ALGORITHM-IDENTIFIER ::= {
    { OID id-md2 PARAMETERS NULL    }|
    { OID id-md5 PARAMETERS NULL    }|
    { OID id-sha1 PARAMETERS NULL   }|
    { OID id-sha256 PARAMETERS NULL }|
    { OID id-sha384 PARAMETERS NULL }|
    { OID id-sha512 PARAMETERS NULL }
}
```

B. Supporting techniques

This section gives several examples of underlying functions supporting the encryption schemes in Section 7 and the encoding methods in Section 9. A range of techniques is given here to allow compatibility with existing applications as well as migration to new techniques. While these supporting techniques are appropriate for applications to implement, none of them is required to be implemented. It is expected 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 9. Hash functions are deterministic, meaning that the output is completely determined by the input. Hash functions take octet strings of variable length, and generate fixed length octet strings. The hash functions used in the operations contained in Sections 7 and 9 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 given as examples for the encoding methods in this document: MD2 [33], MD5 [41], SHA-1 [38], and the proposed algorithms SHA-256, SHA-384, and SHA-512 [39]. For the RSAES-OAEP encryption scheme and EMSA-PSS encoding method, only SHA-1 and SHA-256/384/512 are recommended. For the EMSA-PKCS1-v1_5 encoding method, SHA-1 and 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 **id-md2**, **id-md5**, **id-sha1**, **id-sha256**, **id-sha384**, and **id-sha512**, identify the respective hash functions:

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

id-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 a value of type **AlgorithmIdentifier** shall have a value of type **NULL**.

Note. Version 1.5 of PKCS #1 also allowed for the use of MD4 in signature schemes. The cryptanalysis of MD4 has progressed significantly in the intervening years. For example, Dobbertin [18] demonstrated how to find collisions for MD4 and that the first two rounds of MD4 are not one-way [20]. Because of these results and others (e.g. [8]), 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 [43] 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. [9][19]).

To address these concerns, SHA-1, SHA-256, SHA-384, or SHA-512 are recommended for new applications. As of today, the best (known) collision attacks against these hash functions are generic attacks with complexity $2^{L/2}$, where L is the bit length of the hash output. For the signature schemes in this document, a collision attack is easily translated into a signature forgery. Therefore, the value $L / 2$ should be at least equal to the desired security level in bits of the signature scheme (a security level of B bits means that the best attack has complexity 2^B). The same rule of thumb can be applied to RSAES-OAEP; it is recommended that the bit length of the seed (which is equal to the bit length of the hash output) be twice the desired security level in bits.

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 given here: MGF1, which is based on a hash function. MGF1 coincides with the mask generation functions defined in IEEE Std 1363-2000 [26] 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 (*mgfSeed, maskLen*)

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

Error: "mask too long"

Steps:

1. If *maskLen* > 2^{32} *hLen*, output "mask too long" and stop.
2. Let *T* be the empty octet string.
3. For *counter* from 0 to $\lceil$ *maskLen / hLen* $\rceil - 1$, do the following:
    a. Convert *counter* to an octet string *C* of length 4 octets (see Section 4.1):

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

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

$$T = T \,\|\, \text{Hash}(mgfSeed \,\|\, C) .$$

4. Output the leading *maskLen* 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 a value of type **AlgorithmIdentifier** shall have a value of type **hashAlgorithm**, 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: 2.1 $

-- 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 are 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 encoding of this in hexadecimal is:
-- (0x)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 rsaEncryption is used in an AlgorithmIdentifier the parameters
-- MUST be present and MUST be NULL.
--
rsaEncryption    OBJECT IDENTIFIER ::= { pkcs-1 1 }

--
-- 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-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 }
```

```
--
-- When the following OIDs are used in an AlgorithmIdentifier the parameters
-- MUST be present and MUST be NULL.
--
md2WithRSAEncryption       OBJECT IDENTIFIER ::= { pkcs-1 2 }
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 }

--
-- 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
}

--
-- OIDs for MD2 and MD5, allowed only in EMSA-PKCS1-v1_5.
--
id-md2 OBJECT IDENTIFIER ::= {
    iso(1) member-body(2) us(840) rsadsi(113549) digestAlgorithm(2) 2
}

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

--
-- When id-mgf1 is used in an AlgorithmIdentifier the parameters MUST be
-- present and MUST be a HashAlgorithm, for example sha1.
--
id-mgf1    OBJECT IDENTIFIER ::= { pkcs-1 8 }

-- ================
--   Useful types
-- ================

ALGORITHM-IDENTIFIER ::= CLASS {
    &id    OBJECT IDENTIFIER  UNIQUE,
    &Type  OPTIONAL
}
    WITH SYNTAX { OID &id [PARAMETERS &Type] }

-- 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 ::= {
--     { OID id-md2  PARAMETERS NULL }|
--     { OID id-md5  PARAMETERS NULL }|
--     { OID id-sha1 PARAMETERS NULL }
-- }
--

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

```
-- ==============
--   Algorithms
-- ==============

--
-- Allowed EME-OAEP and EMSA-PSS digest algorithms.
--
OAEP-PSSDigestAlgorithms    ALGORITHM-IDENTIFIER ::= {
    { OID id-sha1 PARAMETERS NULL   }|
    { OID id-sha256 PARAMETERS NULL }|
    { OID id-sha384 PARAMETERS NULL }|
    { OID id-sha512 PARAMETERS NULL },
    ...  -- Allows for future expansion --
}

--
-- Allowed EMSA-PKCS1-v1_5 digest algorithms.
--
PKCS1-v1-5DigestAlgorithms    ALGORITHM-IDENTIFIER ::= {
    { OID id-md2 PARAMETERS NULL    }|
    { OID id-md5 PARAMETERS NULL    }|
    { OID id-sha1 PARAMETERS NULL   }|
    { OID id-sha256 PARAMETERS NULL }|
    { OID id-sha384 PARAMETERS NULL }|
    { OID id-sha512 PARAMETERS NULL }
}

sha1    HashAlgorithm ::= {
    algorithm   id-sha1,
    parameters  SHA1Parameters : NULL
}

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

SHA1Parameters ::= NULL

--
-- Allowed mask generation function algorithms.
-- If the identifier is id-mgf1, the parameters are a HashAlgorithm.
--
PKCS1MGFAlgorithms    ALGORITHM-IDENTIFIER ::= {
    { OID id-mgf1 PARAMETERS HashAlgorithm },
    ...  -- Allows for future expansion --
}

--
-- Default AlgorithmIdentifier for id-RSAES-OAEP.maskGenAlgorithm and
-- id-RSASSA-PSS.maskGenAlgorithm.
--
mgf1SHA1    MaskGenAlgorithm ::= {
    algorithm   id-mgf1,
    parameters  HashAlgorithm : sha1
}

MaskGenAlgorithm ::= AlgorithmIdentifier { {PKCS1MGFAlgorithms} }

--
-- Allowed algorithms for pSourceAlgorithm.
--
PKCS1PSourceAlgorithms    ALGORithm-IDENTIFIER ::= {
    { OID id-pSpecified PARAMETERS EncodingParameters },
    ...  -- Allows for future expansion --
}
```

```
EncodingParameters ::= OCTET STRING(SIZE(0..MAX))

--
-- This identifier means that the label L is an empty string, so the digest
-- of the empty string appears in the RSA block before masking.
--

pSpecifiedEmpty    PSourceAlgorithm ::= {
    algorithm   id-pSpecified,
    parameters  EncodingParameters : emptyString
}

PSourceAlgorithm ::= AlgorithmIdentifier { {PKCS1PSourceAlgorithms} }

emptyString    EncodingParameters ::= ''H

--
-- Type identifier definitions for the PKCS #1 OIDs.
--
PKCS1Algorithms    ALGORITHM-IDENTIFIER ::= {
    { OID rsaEncryption              PARAMETERS NULL } |
    { OID md2WithRSAEncryption       PARAMETERS NULL } |
    { OID md5WithRSAEncryption       PARAMETERS NULL } |
    { OID sha1WithRSAEncryption      PARAMETERS NULL } |
    { OID sha256WithRSAEncryption    PARAMETERS NULL } |
    { OID sha384WithRSAEncryption    PARAMETERS NULL } |
    { OID sha512WithRSAEncryption    PARAMETERS NULL } |
    { OID id-RSAES-OAEP PARAMETERS RSAES-OAEP-params } |
    PKCS1PSourceAlgorithms                           |
    { OID id-RSASSA-PSS PARAMETERS RSASSA-PSS-params } ,
    ...  -- 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,  -- 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 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 {
    hashAlgorithm      [0] HashAlgorithm     DEFAULT sha1,
    maskGenAlgorithm   [1] MaskGenAlgorithm  DEFAULT mgf1SHA1,
    pSourceAlgorithm   [2] PSourceAlgorithm  DEFAULT pSpecifiedEmpty
}

--
-- Identifier for default RSAES-OAEP algorithm identifier.
-- The DER Encoding of this is in hexadecimal:
-- (0x)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    RSAES-AlgorithmIdentifier ::= {
    algorithm   id-RSAES-OAEP,
    parameters  RSAES-OAEP-params : {
        hashAlgorithm       sha1,
        maskGenAlgorithm    mgf1SHA1,
        pSourceAlgorithm    pSpecifiedEmpty
    }
}

RSAES-AlgorithmIdentifier ::= AlgorithmIdentifier { {PKCS1Algorithms} }

--
-- AlgorithmIdentifier.parameters for id-RSASSA-PSS.
-- Note that the tags in this Sequence are explicit.
--
RSASSA-PSS-params ::= SEQUENCE {
    hashAlgorithm      [0] HashAlgorithm      DEFAULT sha1,
    maskGenAlgorithm   [1] MaskGenAlgorithm   DEFAULT mgf1SHA1,
    saltLength         [2] INTEGER            DEFAULT 20,
    trailerField       [3] TrailerField       DEFAULT trailerFieldBC
}

TrailerField ::= INTEGER { trailerFieldBC(1) }

--
-- Identifier for default RSASSA-PSS algorithm identifier
-- The DER Encoding of this is in hexadecimal:
-- (0x)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    RSASSA-AlgorithmIdentifier ::= {
    algorithm   id-RSASSA-PSS,
    parameters  RSASSA-PSS-params : {
        hashAlgorithm       sha1,
        maskGenAlgorithm    mgf1SHA1,
        saltLength          20,
```

```
        trailerField        trailerFieldBC
    }
}

RSASSA-AlgorithmIdentifier ::= AlgorithmIdentifier { {PKCS1Algorithms} }

--
-- Syntax for the EMSA-PKCS1-v1_5 hash identifier.
--
DigestInfo ::= SEQUENCE {
    digestAlgorithm DigestAlgorithm,
    digest OCTET STRING
}

DigestAlgorithm ::= AlgorithmIdentifier { {PKCS1-v1-5DigestAlgorithms} }

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*" [23]. The PSS signature scheme is specified in the IEEE P1363a draft [27], which was in ballot resolution when this document was published.

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 [35].

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 Draft D2: Key Establishment Using Integer Factorization Cryptography*. Working Draft, March 2002.

[2] M. Bellare, A. Desai, D. Pointcheval and P. Rogaway. Relations Among Notions of Security for Public-Key Encryption Schemes. In H. Krawczyk, editor, *Advances in Cryptology – Crypto '98*, volume 1462 of *Lecture Notes in Computer Science*, pp. 26 – 45. Springer Verlag, 1998.

[3] M. Bellare and P. Rogaway. Optimal Asymmetric Encryption – How to Encrypt with RSA. In A. De Santis, editor, *Advances in Cryptology – Eurocrypt '94*, volume 950 of *Lecture Notes in Computer Science*, pp. 92 – 111. Springer Verlag, 1995.

[4] M. Bellare and P. Rogaway. The Exact Security of Digital Signatures – How to Sign with RSA and Rabin. In U. Maurer, editor, *Advances in Cryptology – Eurocrypt '96*, volume 1070 of *Lecture Notes in Computer Science*, 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 H. Krawczyk, editor, *Advances in Cryptology – Crypto '98*, volume 1462 of *Lecture Notes in Computer Science*, 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 1998.

[8] B. den Boer and A. Bosselaers. An Attack on the Last Two Rounds of MD4. In J. Feigenbaum, editor, *Advances in Cryptology – Crypto '91*, volume 576 of *Lecture Notes in Computer Science*, pp. 194 – 203. Springer Verlag, 1992.

[9] B. den Boer and A. Bosselaers. Collisions for the Compression Function of MD5. In T. Helleseth, editor, *Advances in Cryptology – Eurocrypt '93*, volume 765 of *Lecture Notes in Computer Science*, pp. 293 – 304. Springer Verlag, 1994.

[10] D. Coppersmith, M. Franklin, J. Patarin and M. Reiter. Low-Exponent RSA with Related Messages. In U. Maurer, editor, *Advances in Cryptology – Eurocrypt '96*, volume 1070 of *Lecture Notes in Computer Science*, pp. 1 – 9. Springer Verlag, 1996.

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

[12] J.-S. Coron. On the Exact Security of Full Domain Hashing. In M. Bellare, editor, *Advances in Cryptology – Crypto 2000*, volume 1880 of *Lecture Notes in Computer Science*, pp. 229 – 235. Springer Verlag, 2000.

[13] J.-S. Coron. Optimal Security Proofs for PSS and Other Signature Schemes. In L. Knudsen, editor, *Advances in Cryptology – Eurocrypt 2002*, volume 2332 of *Lecture Notes in Computer Science*, pp. 272 – 287. Springer Verlag, 2002.

[14] J.-S. Coron, M. Joye, D. Naccache and P. Paillier. New Attacks on PKCS #1 v1.5 Encryption. In B. Preneel, editor, *Advances in Cryptology – Eurocrypt 2000*, volume 1807 of *Lecture Notes in Computer Science*, pp. 369 – 379. Springer Verlag, 2000.

[15] J.-S. Coron, D. Naccache and J. P. Stern. On the Security of RSA Padding. In M. Wiener, editor, *Advances in Cryptology – Crypto '99*, volume 1666 of *Lecture Notes in Computer Science*, pp. 1 – 18. Springer Verlag, 1999.

[16] Y. Desmedt and A.M. Odlyzko. A Chosen Text Attack on the RSA Cryptosystem and Some Discrete Logarithm Schemes. In H.C. Williams, editor, *Advances in Cryptology – Crypto '85*, volume 218 of *Lecture Notes in Computer Science*, pp. 516 – 522. Springer Verlag, 1986.

[17] T. Dierks and C. Allen. *IETF RFC 2246: The TLS Protocol Version 1.0*. January 1999.

[18] H. Dobbertin. Cryptanalysis of MD4. In D. Gollmann, editor, *Fast Software Encryption '96*, volume 1039 of *Lecture Notes in Computer Science*, pp. 55 – 72. Springer Verlag, 1996.

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

[20] H. Dobbertin. The First Two Rounds of MD4 are Not One-Way. In S. Vaudenay, editor, *Fast Software Encryption '98*, volume 1372 in *Lecture Notes in Computer Science*, pp. 284 – 292. Springer Verlag, 1998.

[21] E. Fujisaki, T. Okamoto, D. Pointcheval and J. Stern. RSA-OAEP is Secure under the RSA Assumption. In J. Kilian, editor, *Advances in Cryptology – Crypto 2001*, volume 2139 of *Lecture Notes in Computer Science*, pp. 260 – 274. Springer Verlag, 2001.

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

[23] 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.

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

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

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

[27] IEEE P1363 working group. *IEEE P1363a D10: Standard Specifications for Public Key Cryptography: Additional Techniques.* November 1, 2001. Available from http://grouper.ieee.org/groups/1363/.

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

[29] *ISO/IEC FDIS 9796-2: Information Technology – Security Techniques – Digital Signature Schemes Giving Message Recovery – Part 2: Integer Factorization Based Mechanisms.* Final Draft International Standard, December 2001.

[30] *ISO/IEC 18033-2: Information Technology – Security Techniques – Encryption Algorithms – Part 2: Asymmetric Ciphers*. V. Shoup, editor, Text for 2nd Working Draft, January 2002.

[31] J. Jonsson. Security Proof for the RSA-PSS Signature Scheme (extended abstract). *Second Open NESSIE Workshop.* September 2001. Full version available from http://eprint.iacr.org/2001/053/.

[32] J. Jonsson and B. Kaliski. On the Security of RSA Encryption in TLS. In *Advances in Cryptology – Crypto 2002*, to appear.

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

[34] B. Kaliski. On Hash Function Identification in Signature Schemes. In B. Preneel, editor, *RSA Conference 2002, Cryptographers' Track*, volume 2271 of *Lecture Notes in Computer Science*, pp. 1 – 16. Springer Verlag, 2002.

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

[36] J. Manger. A Chosen Ciphertext Attack on RSA Optimal Asymmetric Encryption Padding (OAEP) as Standardized in PKCS #1 v2.0. In J. Kilian, editor, *Advances in Cryptology – Crypto 2001*, volume 2139 of *Lecture Notes in Computer Science*, pp. 260 – 274. Springer Verlag, 2001.

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

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

[39] National Institute of Standards and Technology (NIST). *Draft FIPS 180-2: Secure Hash Standard.* Draft, May 2001. Available from http://www.nist.gov/sha/.

[40] J.-J. Quisquater and C. Couvreur. Fast Decipherment Algorithm for RSA Public-Key Cryptosystem. *Electronics Letters*, 18 (21), pp. 905 – 907, October 1982

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

[42] 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.

[43] 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 1995.

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

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

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

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

[48] V. Shoup. OAEP Reconsidered. In J. Kilian, editor, *Advances in Cryptology – Crypto 2001*, volume 2139 of *Lecture Notes in Computer Science*, pp. 239 – 259. Springer Verlag, 2001.

[49] 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/.

[50] G. J. Simmons. Subliminal communication is easy using the DSA. In T. Helleseth, editor, *Advances in Cryptology – Eurocrypt '93*, volume 765 of *Lecture Notes in Computer Science*, 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
174 Middlesex Turnpike
Bedford, MA 01730 USA
pkcs-editor@rsasecurity.com
http://www.rsasecurity.com/rsalabs/pkcs