Groupe de travail Réseau
Request for Comments : 3447
RFC rendue obsolète : 2437
Catégorie : Information
Traduction Claude Brière de L'Isle

J. Jonsson
B. Kaliski
RSA Laboratories
février 2003

Normes de cryptographie à clé publique (PKCS) n° 1 : Spécifications de la cryptographie RSA version 2.1

Statut de ce mémoire
Le présent mémoire apporte des informations à la communauté de l'Internet. Il ne spécifie aucune norme de l'Internet. Sa distribution n'est soumise à aucune restriction.

Résumé
Le présent mémoire présente une republication de PKCS n° 1 v2.1 d'après la série de normes de cryptographie à clé publique (PKCS, *Public-Key Cryptography Standards*) des laboratoires RSA, et le contrôle des changements est conservé au sein du processus PKCS. Le corps du présent document est directement tiré du document PKCS n° 1 v2.1, avec certaines corrections apportées durant le processus de publication.

Table des matières

1. Introduction

Le présent document apporte des recommandations pour la mise en œuvre de la cryptographie à clé publiques fondée sur l'algorithme RSA [42], sur les aspects suivants :

* primitives cryptographiques
* schémas de chiffrement
* schémas de signature avec appendice
* syntaxe ASN.1 pour la représentation des clés et l'identification des schémas

Les recommandations sont destinées à l'application générale au sein des systèmes d'ordinateurs et de communication, et comme tels sont très souples. Il est prévu que les normes d'application fondées sur ces spécifications pourront comporter des contraintes supplémentaires. Les recommandations sont destinées à être compatibles avec la norme IEEE-1363-2000 [26] et les projets de normes actuellement en cours de développement par les groupes de travail ANSI X9F1 [1] et IEEE P1363 [27].

Le présent document subroge PKCS n° 1 version 2.0 [35], [44] mais comporte des techniques compatibles.

L'organisation du document est la suivante :

* La Section 1 est une introduction.
* La Section 2 définit la notation utilisée.
* La Section 3 définit les types de clés RSA publiques et privées.
* Les sections 4 et 5 définissent plusieurs primitives, ou opérations mathématiques de base. Les primitives de conversion de données sont à la Section 4, et les primitives cryptographiques (chiffrement/déchiffrement, signature, vérification) sont à la Section 5.
* Les sections 6, 7 et 8 traitent des schémas de chiffrement et de signature. La Section 6 donne une vue générale. Avec les méthodes qui se trouvent dans PKCS n° 1 v1.5, la Section 7 définit un schéma de chiffrement fondé sur OAEP, et la Section 8 définit un schéma de signature fondé sur PSS [4], [5] avec les appendices.
* La Section 9 définit la méthode de codage pour les schémas de signature de la Section 8.
* L'Appendice A définit la syntaxe ASN.1 pour les clés définies à la Section 3 et les schémas des sections 7 et 8.
* L'Appendice B définit les fonctions de hachage et de génération de gabarit utilisées dans ce document, y compris la syntaxe ASN.1 pour les techniques.
* L'Appendice C donne un module ASN.1.
* Les Appendices D, E, F et G couvrent les questions de propriété intellectuelle, retracent l'histoire des révisions de PKCS n° 1, donnent les références aux autres normes et publications, et donnent des informations générales sur les normes de cryptographie à clé publique.

2. Notation

c	représentation de texte chiffré, un entier entre 0 et n-1
C	texte chiffré, une chaîne d'octets
d	exposant privé RSA
d_i	exposant CRT du facteur additionnel r_i, entier positif tel que e * d_i == 1 (mod (r_i-1)), i = 3, ..., u
dP	exposant CRT de p, entier positif tel que e * dP == 1 (mod (p-1))
dQ	exposant CRT de q, entier positif tel que e * dQ == 1 (mod (q-1))
e	exposant public RSA
EM	message codé, une chaîne d'octets
emBits	longueur (prévue) en bits d'un message codé (EM)
emLen	longueur (prévue) en octets d'un message codé (EM)
GCD(. , .)	plus grand commun diviseur de deux entiers non négatifs
Hash	fonction de hachage
hLen	longueur résultante en octets de la fonction de hachage Hash
k	longueur en octets du module RSA n
K	clé privée RSA
L	étiquette RSAES-OAEP facultative, une chaîne d'octets
LCM(., ..., .)	plus petit commun multiple d'une liste d'entiers non négatifs
m	représentation de message, un entier entre 0 et n-1
M	message, une chaîne d'octets
mask	résultat de MGF, une chaîne d'octets
maskLen	longueur (prévue) du gabarit de la chaîne d'octets
MGF	*(mask generation function)* fonction de génération de gabarit
mgfSeed	germe à partir duquel le gabarit est généré, une chaîne d'octets
mLen	longueur en octets d'un message M

n	module RSA, n = r_1 * r_2 * ... * r_u , u >= 2
(n, e)	clé publique RSA
p, q	deux premiers facteurs premiers du module RSA n
qInv	coefficient de CRT, entier positif inférieur à p tel que q * qInv == 1 (mod p)
r_i	facteurs premiers du module RSA n, incluant r_1 = p, r_2 = q, et des facteurs supplémentaires s'il en est
s	représentation de signature, un entier entre 0 et n-1
S	signature, une chaîne d'octets
sLen	longueur en octets du sel EMSA-PSS
t_i	coefficient de CRT du facteur premier r_i supplémentaire, entier positif inférieur à r_i tel que : r_1 * r_2 * ... * r_(i-1) * t_i == 1 (mod r_i) , i = 3, ... , u
u	nombre de facteurs premiers du module RSA, u ≥ 2
x	entier non négatif
X	chaîne d'octets correspondant à x
xLen	longueur (prévue) d la chaîne d'octets X
0x	indicateur de la représentation hexadécimale d'un octet ou d'une chaîne d'octets ; "0x48" note l'octet qui a la valeur hexadécimale 48 ; "(0x)48 09 0e" note la chaîne de trois octets consécutifs avec, respectivement, la valeur hexadécimale 48, 09, et 0e
\lambda(n)	LCM(r_1-1, r_2-1, ... , r_u-1)
\xor	OU exclusif au bit près de deux chaînes d'octets
\plafond(.)	fonction plafond ; \plafond(x) est le plus petit entier supérieur ou égal au nombre réel x
||	opérateur d'enchaînement
==	symbole de congruence ; a == b (mod n) signifie que l'entier n divise l'entier a - b

Note : Le CRT peut être appliqué de façon non récurrente aussi bien que récurrente. Dans le présent document, on utilise une approche récurrente suivant l'algorithme de Garner [22]. Voir aussi la Note 1 du paragraphe 3.2.

3. Types de clé

Deux types de clé sont employés dans les primitives et schémas définis dans le présent document : la clé publique RSA et la clé privée RSA. Ensemble, une clé publique RSA et une clé privée RSA forment une paire de clés RSA.

La présente spécification prend en charge ce qu'on appelle le RSA "multi-premier" où le module peut avoir plus de deux facteurs premiers. L'avantage du RSA multi-premier est un moindre coût de calcul des primitives de déchiffrement et de signature, pourvu que soit utilisé le théorème du reste chinois (CRT, *Chinese Remainder Theorem*). De meilleures performances peuvent être obtenues sur des plates-formes à un seul processeur, mais encore mieux sur des plates-formes multiprocesseur, où les exponentiations modulaires impliquées peuvent être faites en parallèle.

Pour un exposé sur la façon dont le multi-premier affecte la sécurité du système de chiffrement RSA, se référer à [49].

3.1 Clé publique RSA

Pour les besoins du présent document, une clé publique RSA comporte deux composants :
n : le module RSA, un entier positif
e : l'exposant public RSA, un entier positif

Dans une clé publique RSA valide, le module RSA est le produit de u premiers impairs distincts r_i, i = 1, 2, ..., u, où u ≥ 2, et l'exposant public RSA e est un entier entre 3 et n - 1 satisfaisant GCD(e, \lambda(n)) = 1, où \lambda(n) = LCM(r_1 - 1, ..., r_u - 1). Par convention, les deux premiers nombres premiers r_1 et r_2 peuvent aussi être notés respectivement p et q.

Une syntaxe recommandée pour interchanger les clés publiques RSA entre les mises en œuvre est donnée à l'appendice A.1.1 ; la représentation interne d'une mise en œuvre peut en différer.

3.2 Clé privée RSA

Pour les besoins du présent document, une clé privée RSA peut avoir l'une de ces deux représentations.

1. La première représentation consiste en la paire (n, d), où les composants ont la signification suivante :
   n le module RSA, un entier positif
   d l'exposant privé RSA, un entier positif

2. La seconde représentation consiste en un quintuplet (p, q, dP, dQ, qInv) et une séquence (éventuellement vide) de triplets (r_i, d_i, t_i), i = 3, ..., u, un pour chaque nombre premier qui n'est pas dans le quintuplet, où les composants ont la significations suivante :
   p le premier facteur, un entier positif
   q le second facteur, un entier positif
   dP l'exposant de CRT du premier facteur, un entier positif
   dQ l'exposant de CRT du second, un entier positif
   qInv le (premier) coefficient de CRT, un entier positif
   r_i le $i^{ème}$ facteur, un entier positif
   d_i l'exposant de CRT du $i^{ème}$ facteur, un entier positif
   t_i le coefficient de CRT du $i^{ème}$ facteur, un entier positif

Dans une clé privée RSA valide avec la première représentation, le module RSA n est le même que dans la clé publique RSA correspondante et il est le produit de u nombres premiers impairs r_i distincts, i = 1, 2, ..., u, où $u \geq 2$. L'exposant privé RSA d est un entier positif inférieur à n qui satisfait à e * d == 1 (mod \lambda(n)), où e est l'exposant public RSA correspondant et \lambda(n) est défini comme dans le paragraphe 3.1.

Dans une clé privée RSA valide avec la seconde représentation, les deux facteurs p et q sont les deux premiers facteurs premiers du module RSA n (c'est-à-dire, r_1 et r_2), les exposants de CRT dP et dQ sont des entiers positifs inférieurs respectivement à p et q qui satisfont à

e * dP == 1 (mod (p-1))
e * dQ == 1 (mod (q-1)) ,

et le coefficient de CRT qInv est un entier positif inférieur à p qui satisfait q * qInv == 1 (mod p).

Si u > 2, la représentation va inclure un ou plusieurs triplets (r_i, d_i, t_i), i = 3, ..., u. Les facteurs r_i sont les facteurs premiers supplémentaires du module RSA n. Chaque exposant de CRT d_i (i = 3, ..., u) satisfait à

e * d_i == 1 (mod (r_i - 1)).

Chaque coefficient de CRT t_i (i = 3, ..., u) est un entier positif inférieur à r_i qui satisfait à

R_i * t_i == 1 (mod r_i) ,

où R_i = r_1 * r_2 * ... * r_(i-1).

Une syntaxe recommandée pour échanger les clés privées RSA entre les mises en œuvre, qui inclut les composants provenant des deux représentations, est donnée à l'appendice A.1.2 ; la représentation interne d'une mise en œuvre peut en différer.

Notes.

1. La définition des coefficients de CRT et les formules qui les utilisent dans les primitives de la Section 5 suivent généralement l'algorithme de Garner [22] (voir aussi l'algorithme 14.71 dans [37]). Cependant, pour la compatibilité avec les représentations des clés privées RSA dans PKCS n°1 v2.0 et les versions précédentes, les rôles de p et q sont inversés par rapport au reste des nombres premiers. Donc, le premier coefficient de CRT, qInv, est défini comme l'inverse de q mod p, plutôt que l'inverse de R_1 mod r_2, c'est-à-dire, de p mod q.

2. Quisquater et Couvreur [40] ont observé les avantages de l'application du théorème du reste chinois aux opérations de RSA.

4. Primitives de conversion des données

Deux primitives de conversion de données sont employées dans les schémas définis dans le présent document :

* I2OSP – *(Integer-to-Octet-String primitive)* primitive d'entier à chaîne d'octets
* OS2IP – *(Octet-String-to-Integer primitive)* primitive de chaîne d'octets à entier

Pour les besoins du présent document, et conformément à la syntaxe ASN.1, une chaîne d'octets est une séquence ordonnée d'octets (de huit bits). La séquence est indexée du premier (par convention, le plus à gauche) au dernier (le plus à droite). Pour les besoins de la conversion de et en entiers, le premier octet est considéré comme de plus fort poids dans les primitives de conversion suivantes.

4.1 I2OSP

I2OSP convertit un entier non négatif en une chaîne d'octets d'une longueur spécifiée.

I2OSP (x, xLen)

Entrée :
x entier non négatif à convertir
xLen longueur prévue de la chaîne d'octets résultante

Résultat : X chaîne d'octets correspondante de longueur xLen

Erreur : "entier trop grand"

Étapes :
1. Si x ≥ 256^xLen, sortir en résultat "entier trop grand" et arrêter.
2. Écrire l'entier x dans sa représentation unique xLen-digit en base 256 :
   x = x_(xLen-1) 256^(xLen-1) + x_(xLen-2) 256^(xLen-2) + ... + x_1 256 + x_0,
   où 0 ≤ x_i < 256 (noter qu'un ou plusieurs chiffres seront à zéro en tête si x est inférieur à 256^(xLen-1)).
3. Soit l'octet X_i qui a la valeur d'entier x_(xLen-i) pour 1 ≤ i ≤ xLen. Sortir la chaîne d'octet
   X = X_1 X_2 ... X_xLen.

4.2 OS2IP

OS2IP convertit une chaîne d'octets en un entier non négatif.

OS2IP (X)

Entrée : X chaîne d'octets à convertir

Résultat : x entier non négatif correspondant

Étapes :
1. Soit X_1 X_2 ... X_xLen qui sont les octets de X du premier au dernier, et soit x_(xLen-i) la valeur d'entier de l'octet X_i pour 1 <= i <= xLen.
2. Soit x = x_(xLen-1) 256^(xLen-1) + x_(xLen-2) 256^(xLen-2) + ... + x_1 256 + x_0.
3. Sortir x.

5. Primitives cryptographiques

Les primitives cryptographiques sont des opérations mathématiques de base sur lesquelles des schémas cryptographiques peuvent être construits. Elles sont destinées à être mises en œuvre dans les matériels ou comme modules de logiciel, et ne sont pas destinées à fournir de sécurité en dehors d'un certain schéma.

Quatre types de primitives sont spécifiés dans le présent document, organisés en paires : chiffrement et déchiffrement ; et signature et vérification.

La spécifications de la primitive suppose que certaines conditions sont satisfaites par les entrées, en particulier que les clés publique et privée RSA sont valides.

5.1 Primitives de chiffrement et de déchiffrement

Une primitive de chiffrement produit une représentation d'un texte chiffré *(ciphertext)* à partir de la représentation d'un message sous le contrôle d'une clé publique, et une primitive de déchiffrement récupère la représentation du message à partir de la représentation du texte chiffré sous le contrôle de la clé privée correspondante.

Une paire de primitives de chiffrement et de déchiffrement est employée dans les schémas de chiffrement définis dans le présent document et est spécifiée ici : RSAEP/RSADP. RSAEP et RSADP impliquent la même opération mathématique, avec des clés différentes en entrée.

Les primitives définies ici sont les mêmes que IFEP-RSA/IFDP-RSA dans la norme IEEE 1363-2000 [26] (sauf que la prise en charge du multi-premier RSA a été ajoutée) et sont compatibles avec PKCS n°1 v1.5.

La principale opération mathématique dans chaque primitive est l'exponentiation.

5.1.1 RSAEP

RSAEP ((n, e), m)

Entrée :
(n, e) clé publique RSA
m représentation de message, un entier entre 0 et n - 1

Résultat : c représentation de texte chiffré, un entier entre 0 et n - 1

Erreur : "représentation de message hors gamme"

Hypothèse : La clé publique RSA (n, e) est valide

Étapes :
1. Si la représentation de message m n'est pas entre 0 et n - 1, sortir "représentation de message hors gamme" et arrêter.
2. Soit c = m^e mod n.
3. Sortir c.

5.1.2 RSADP

RSADP (K, c)

Entrée :
K : clé privée RSA, où K a une des formes suivantes :
- une paire (n, d)
- un quintuplet (p, q, dP, dQ, qInv) et une séquence de triplets (r_i, d_i, t_i), i = 3, ..., u éventuellement vide.

c : représentation de texte chiffré, un entier entre 0 et n - 1

Résultat : m représentation de message, un entier entre 0 et n - 1

Erreur : "représentation de texte chiffré hors gamme"

Hypothèse : La clé privée RSA K est valide

Étapes :
1. Si la représentation de texte chiffré c n'est pas entre 0 et n - 1, sortir "représentation de texte chiffré hors gamme" et arrêter.
2. La représentation de message m est calculée comme suit :
   a. Si la première forme (n, d) de K est utilisée, soit m = c^d mod n.
   b. Si la seconde forme (p, q, dP, dQ, qInv) et (r_i, d_i, t_i) de K est utilisée, continuer comme suit :
      i. Soit m_1 = c^dP mod p et m_2 = c^dQ mod q.
      ii. Si u > 2, soit m_i = c^(d_i) mod r_i, i = 3, ..., u.
      iii. Soit h = (m_1 - m_2) * qInv mod p.
      iv. Soit m = m_2 + q * h.
      v. Si u > 2, soit R = r_1 et pour i = 3 à u faire
         1. Soit R = R * r_(i-1).
         2. Soit h = (m_i - m) * t_i mod r_i.
         3. Soit m = m + R * h.
3. Sortir m.

Note : L'étape 2.b peut être réécrite comme une seule boucle, pourvu qu'on inverse l'ordre de p et q. Cependant pour rester cohérent avec PKCS n°1 v2.0, les deux premiers nombres premiers p et q sont traités séparément des nombres premiers supplémentaires.

5.2 Primitives de signature et de vérification

Une primitive de signature produit une représentation de signature à partir d'une représentation de message sous le contrôle d'une clé privée, et une primitive de vérification récupère la représentation de message à partir de la représentation de signature sous le contrôle de la clé publique correspondante. Une paire de primitives de signature et de vérification est employée dans les schémas de signature définis dans le présent document et est spécifiée ici : RSASP1/RSAVP1.

Les primitives définies ici sont les mêmes que IFSP-RSA1/IFVP-RSA1 dans IEEE 1363-2000 [26] (excepté l'ajout de la prise en charge des multi-premiers RSA) et sont compatibles avec PKCS n°1 v1.5.

La principale opération mathématique dans chaque primitive est l'exponentiation, comme dans les primitives de chiffrement et de déchiffrement du paragraphe 5.1. RSASP1 et RSAVP1 sont les mêmes que RSADP et RSAEP sauf les noms de leurs arguments d'entrée et de sortie ; ils sont distincts car ils sont destinés à des objets différents.

5.2.1 RSASP1

RSASP1 (K, m)

Entrée :
K : clé privée RSA, où K a une des formes suivantes :
- une paire (n, d)
- un quintuplet (p, q, dP, dQ, qInv) et une séquence (éventuellement vide) de triplets (r_i, d_i, t_i), i = 3, ..., u

m : représentation de message, un entier entre 0 et n - 1

Résultat : s représentation de signature, un entier entre 0 et n - 1

Erreur : "représentation de message hors de gamme"

Hypothèse : la clé privée RSA K est valide

Étapes :
1. Si la représentation de message m n'est pas entre 0 et n - 1, sortir "représentation de message hors gamme" et arrêter.
2. La représentation de signature s est calculée comme suit :
   a. Si la première forme (n, d) de K est utilisée, soit s = m^d mod n.
   b. Si la seconde forme (p, q, dP, dQ, qInv) et (r_i, d_i, t_i) de K est utilisée, continuer comme suit :
      i. Soit s_1 = m^dP mod p et s_2 = m^dQ mod q.
      ii. Si u > 2, soit s_i = m^(d_i) mod r_i, i = 3, ..., u.
      iii. Soit h = (s_1 - s_2) * qInv mod p.
      iv. Soit s = s_2 + q * h.
      v. Si u > 2, soit R = r_1 et pour i = 3 à u, faire
         1. Soit R = R * r_(i-1).
         2. Soit h = (s_i - s) * t_i mod r_i.
         3. Soit s = s + R * h.
3. Sortir s.

Note : L'étape 2.b peut être réécrite comme une seule boucle, pourvu qu'on inverse l'ordre de p et q. Cependant, pour la cohérence avec PKCS n°1 v2.0, les deux premiers nombres premiers p et q sont traités séparément des nombres premiers supplémentaires.

5.2.2 RSAVP1

RSAVP1 ((n, e), s)

Entrée :
(n, e) : clé publique RSA
s : représentation de signature, un entier entre 0 et n - 1

Résultat : m représentation de message, un entier entre 0 et n - 1

Erreur : "représentation de signature hors gamme"

Hypothèse : la clé publique RSA (n, e) est valide

Étapes :
1. Si la représentation de signature s n'est pas entre 0 et n - 1, sortir "représentation de signature hors gamme" et arrêter.
2. Soit m = s^e mod n.
3. Sortir m.

6. Survol des schémas

Un schéma combine les primitives cryptographiques et d'autres techniques pour atteindre un objectif de sécurité particulier. Deux types de schéma sont spécifiés dans le présent document : des schémas de chiffrement et des schémas de signature avec un appendice.

Les schémas spécifiés dans le présent document sont de portée limitée en ce que leurs opérations consistent seulement en étapes de traitement des données avec une clé publique ou privée RSA, et ils n'incluent pas d'étapes pour obtenir ou valider la clé. Donc, en plus des opérations des schémas, une application va normalement inclure des opérations de gestion de clé par lesquelles les parties peuvent choisir des clés publiques et des clés privées RSA pour le fonctionnement d'un schéma. Les opérations supplémentaires spécifiques et les autres détails sortent du domaine d'application de ce document.

Comme c'était le cas pour les primitives cryptographiques (Section 5) les spécifications des opérations de schémas supposent que certaines conditions sont satisfaites par les entrées, en particulier que les clés publiques et privées RSA sont valides. Le comportement d'une mise en œuvre est donc non spécifié lorsque une clé est invalide. L'impact d'un tel comportement non spécifié dépend de l'application. Les moyens possibles de traiter la validation de clé incluent la validation explicite de la clé par l'application, la validation de la clé au sein de l'infrastructure de clé publique, et l'attribution de la responsabilité des opérations effectuées avec une clé invalide à la partie qui a généré la clé.

Une pratique cryptographique généralement bonne est d'employer une certaine paire de clés RSA dans un seul schéma. Cela évite le risque que des faiblesses dans un schéma puissent compromettre la sécurité des autres, et cela peut être essentiel pour maintenir une sécurité démontrable. Alors que RSAES-PKCS1-v1_5 (paragraphe 7.2) et RSASSA-PKCS1-v1_5 (paragraphe 8.2) ont été traditionnellement employés ensemble sans aucune mauvaise interaction connue (bien sûr, c'est le modèle introduit par PKCS n°1 v1.5) un tel usage combiné d'une paire de clés RSA n'est pas recommandé pour de nouvelles applications.

Pour illustrer les risques qui se rapportent à l'emploi d'une paire de clés RSA dans plus d'un schéma, supposons qu'une paire de clés RSA soit employée à la fois dans RSAES-OAEP (paragraphe 7.1) et RSAES-PKCS1-v1_5. Bien que par lui-même RSAES-OAEP résiste à l'attaque, un agresseur pourrait être capable d'exploiter une faiblesse de la mise en œuvre de RSAES-PKCS1-v1_5 pour récupérer des messages chiffrés avec l'un ou l'autre schéma. Comme autre exemple, supposons qu'une paire de clés RSA soit employée à la fois dans RSASSA-PSS (paragraphe 8.1) et dans RSASSA-PKCS1-v1_5. La preuve de sécurité pour RSASSA-PSS ne serait alors plus suffisante car la preuve ne tient pas compte de la possibilité que les signatures pourraient être générées avec un second schéma. Des considérations similaires peuvent s'appliquer si une paire de clés RSA est employée dans un des schémas définis ici et dans une variante définie ailleurs.

7. Schémas de chiffrement

Pour les besoins du présent document, un schéma de chiffrement consiste en une opération de chiffrement et une opération de déchiffrement, où l'opération de chiffrement produit un texte chiffré à partir d'un message avec la clé publique RSA d'un receveur, et l'opération de déchiffrement récupère le message à partir du texte chiffré avec la clé privée RSA correspondante du receveur.

Un schéma de chiffrement peut être employé dans diverses applications. Une application typique est un protocole d'établissement de clé, où le message contient le matériel de clé à livrer confidentiellement d'une partie à l'autre. Par exemple, PKCS n°7 [45] emploie un tel protocole pour livrer une clé de chiffrement de contenu d'un envoyeur à un receveur ; les schémas de chiffrement définis ici conviendraient comme algorithmes de chiffrement de clé dans ce contexte.

Deux schémas de chiffrement sont spécifiés dans le présent document : RSAES-OAEP et RSAES-PKCS1-v1_5. RSAES-OAEP est recommandé pour les nouvelles applications ; RSAES-PKCS1-v1_5 n'est inclus que pour la compatibilité avec les applications existantes, et n'est pas recommandé pour les nouvelles applications.

Les schémas de chiffrement donnés ici suivent un modèle général similaire à ceux employés dans la norme IEEE 1363-

2000 [26], combinant les primitives de chiffrement et de déchiffrement avec une méthode de codage pour le chiffrement. Les opérations de chiffrement appliquent une opération de codage de message à un message pour produire un message codé, qui est alors converti en une représentation de message entier. Une primitive de chiffrement est appliquée à la représentation de message pour produire le texte chiffré. En inversant cela, les opérations de déchiffrement appliquent une primitive de déchiffrement au texte chiffré pour récupérer une représentation de message, qui est alors convertie en un message codé en chaîne d'octets. Une opération de décodage de message est appliquée au message codé pour récupérer le message et vérifier l'exactitude du déchiffrage.

Pour éviter les faiblesses de mise en œuvre en rapport avec la façon dont les erreurs sont traitées au sein de l'opération de décodage (voir [6] et [36]) les opérations de codage et décodage pour RSAES-OAEP et RSAES-PKCS1-v1_5 sont incorporées dans les spécifications des schémas de chiffrement respectifs plutôt que définis dans des spécifications distinctes. Les deux schémas de chiffrement sont compatibles avec les schémas correspondants dans PKCS n°1 v2.0.

7.1 RSAES-OAEP

RSAES-OAEP combine les primitives RSAEP et RSADP (paragraphes 5.1.1 et 5.1.2) avec la méthode de codage EME-OAEP (étape 1.b du paragraphe 7.1.1 et étape 3 du paragraphe 7.1.2). EME-OAEP se fonde sur le schéma Chiffrement asymétrique optimal de Bellare et Rogaway [3]. (OAEP, *Optimal Asymmetric Encryption Padding* veut dire bourrage de chiffrement asymétrique optimal). Il est compatible avec le schéma IFES défini dans la norme IEEE 1363-2000 [26], où les primitives de chiffrement et de déchiffrement sont IFEP-RSA et IFDP-RSA et la méthode de codage du message est EME-OAEP. RSAES-OAEP peut opérer sur des messages de longueur allant jusqu'à k - 2hLen - 2 octets, où hLen est la longueur du résultat de la fonction de hachage sous-jacente et k est la longueur en octets du module RSA du receveur.

En supposant que le calcul de la $e^{ème}$ racine modulo n est infaisable et que la fonction de génération de gabarit dans RSAES-OAEP a les propriétés appropriées, RSAES-OAEP est sémantiquement sûr contre les attaques de texte choisi chiffré adaptatif. Cette assurance est prouvable au sens que la difficulté de casser RSAES-OAEP peut être mise en rapport direct avec la difficulté d'inverser la fonction RSA, pourvu que la fonction de génération de gabarit soit vue comme une boîte noire ou un oracle aléatoire ; voir [21] et la note ci-dessous pour plus d'explications.

Les opérations de chiffrement et de déchiffrement de RSAES-OAEP prennent toutes deux la valeur d'une étiquette L comme entrée. Dans cette version de PKCS n°1, L est la chaîne vide ; les autres utilisations de l'étiquette sortent du domaine d'application du présent document. Voir à l'Appendice A.2.1 la syntaxe ASN.1 pertinente.

RSAES-OAEP est paramétré par le choix d'une fonction de hachage et d'une fonction de génération de gabarit. Ce choix devrait être fixé pour une certaine clé RSA. Des suggestions de fonctions de hachage et de génération de gabarits sont données à l'Appendice B.

Note De récents résultats ont aidé à préciser les propriétés de sécurité de la méthode de codage OAEP [3] (en gros, la procédure décrite à l'étape 1.b du paragraphe 7.1.1). Le cadre est le suivant. En 1994, Bellare et Rogaway [3] ont introduit un concept de sécurité qu'ils ont appelé "connaissance du texte en clair" *(plaintext awareness)* (PA94). Ils ont prouvé que si une primitive de chiffrement de clé publique déterministe (par exemple, RSAEP) est difficile à inverser sans la clé privée, le schéma de chiffrement correspondant fondé sur OAEP est à connaissance du texte en clair (dans le modèle d'oracle aléatoire) ce qui signifie en gros qu'un adversaire ne peut pas produire un texte chiffré valide sans réellement "connaître" le texte en clair sous-jacent. La connaissance du texte en clair d'un schéma de chiffrement est en relation étroite avec la résistance du schéma aux attaques de texte chiffré choisi. Dans de telles attaques, un adversaire a l'opportunité d'envoyer des interrogations à un oracle qui simule la primitive de déchiffrement. En utilisant les résultats de ces interrogations, l'adversaire tente de déchiffrer un texte chiffré qui sert d'épreuve.

Cependant, il y a deux types d'attaques de texte chiffré choisi, et PA94 n'implique la sécurité que contre l'une d'entre elles. Les différences relèvent de ce que qu'il est permis à l'adversaire de faire après qu'il a reçu le texte chiffré de l'épreuve. Le scénario d'attaque indifférente (noté CCA1) ne permet aucune interrogation à l'oracle déchiffreur après que l'adversaire a obtenu le texte chiffré de l'épreuve, alors que le scénario adaptatif (noté CCA2) le permet (sauf que l'oracle déchiffreur refuse de déchiffrer le texte chiffré de l'épreuve une fois qu'il est publié). En 1998, Bellare et Rogaway, avec Desai et Pointcheval [2], ont présenté une nouvelle notion plus forte de connaissance du texte en clair (PA98) qui implique la sécurité contre CCA2.

Pour résumer, il y a eu deux sources potentielles de malentendu : que PA94 et PA98 sont des concepts équivalents ; ou que CCA1 et CCA2 sont des concepts équivalents. L'une et l'autre de ces hypothèses conduit à la conclusion que l'article de Bellare-Rogaway implique la sécurité de OAEP contre CCA2, ce qui n'est pas vrai.

Note Il serait juste de mentionner que PKCS n°1 v2.0 cite [3] et annonce que "une attaque de texte chiffré choisi est inefficace contre un schéma de chiffrement à connaissance du texte en clair tel que RSAES-OAEP" sans spécifier quelle sorte de connaissance du texte en clair ou attaque de texte chiffré choisi est considérée.)

Il n'a jamais été prouvé qu OAEP est sûr contre CCA2 ; en fait, Victor Shoup [48] a démontré qu'il n'existe pas une telle preuve dans le cas général. En bref, Shoup montre qu'un adversaire, dans le scénario CCA2, qui sait comment inverser partiellement la primitive de chiffrement mais ne sait pas comment l'inverser complètement peut bien être capable de casser ce schéma. Par exemple, on peut imaginer un attaquant qui soit capable de casser RSAES-OAEP si il sait comment récupérer tout sauf les 20 premiers octets d'un entier aléatoire chiffré avec RSAEP. Un tel attaquant n'a pas besoin d'être capable d'inverser complètement RSAEP, parce qu'il n'utilise pas les 20 premiers octets dans son attaque.

Il reste que RSAES-OAEP est sûr contre CCA2, ce qui a été prouvé par Fujisaki, Okamoto, Pointcheval, et Stern [21] peu après l'annonce du résultat de Shoup. En utilisant d'habiles techniques de réduction en treillis, ils ont montré comment inverser complètement RSAEP connaissant une certaine partie suffisamment grande de la pré-image. Cette observation, combinée avec une preuve que OAEP est sûr contre CCA2 si la primitive de chiffrement sous-jacente est difficile à inverser partiellement, comble le trou entre ce que Bellare et Rogaway ont prouvé sur RSAES-OAEP et ce que certains ont pu croire qu'ils avaient prouvé. Un peu paradoxalement, on est donc sauvé par une ostensible faiblesse de RSAEP (c'est-à-dire, l'inverse entier peut être déduit de ses parties).

Malheureusement, la réduction de sécurité n'est pas efficace pour les paramètres concrets. Alors que la démonstration met bien en relation un adversaire Adv contre la sécurité CCA2 de RSAES-OAEP avec un algorithme Inv qui inverse RSA, la probabilité de succès de Inv est seulement approximativement de \epsilon^2 / 2^18, où \epsilon est la probabilité de succès de Adv.

Note Dans [21] la probabilité de succès de l'inverseur était \epsilon^2 / 4. Le facteur supplémentaire 1 / 2^16 est dû aux huit bits zéro fixés au début du message EM codé, qui ne sont pas présents dans la variante d'OAEP considérée dans [21] (Inv doit appliquer Adv deux fois pour inverser RSA, et chaque application correspond à un facteur 1 / 2^8).)

De plus, le temps de démarrage pour Inv est approximativement t^2, où t est le temps de démarrage de l'adversaire. La conséquence est qu'on ne peut pas exclure la possibilité que l'attaque de RSAES-OAEP soit considérablement plus facile que d'inverser RSA pour des paramètres concrets. Cependant, l'existence d'une preuve de sécurité donne une certaine assurance que la construction RSAES-OAEP est plus fondée que les constructions ad hoc telles que RSAES-PKCS1-v1_5.

Des schémas de chiffrement hybrides fondés sur le paradigme de l'encapsulation de clé RSA-KEM offrent des preuves strictes de sécurité directement applicables à des paramètres concrets ; voir l'exposé dans [30]. De futures versions de PKCS n°1 pourraient spécifier des schémas fondés sur ce paradigme.

7.1.1 Opération de chiffrement

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

Options :
Hash : Fonction de hachage (hLen note la longueur en octets du résultat de la fonction de hachage)
MGF : Fonction de génération de gabarit

Entrée :
(n, e) ; clé publique RSA du receveur (k note la longueur en octets du module n RSA)
M : message à chiffrer, une chaîne d'octet de longueur mLen, où mLen ≤ k - 2hLen - 2
L : étiquette facultative à associer au message ; la valeur par défaut pour L, si L n'est pas fourni, est la chaîne vide

Résultat :
C : texte chiffré, une chaîne d'octet de longueur k
Erreurs :"message trop long"; "étiquette trop longue"
Hypothèse : la clé publique RSA (n, e) est valide

Étapes :
1. Longueur de vérification :
   a. Si la longueur de L est supérieure à la limitation d'entrée pour la fonction de hachage (2^61 - 1 octets pour SHA-1) sortir "étiquette trop longue" et arrêter.
   b. Si mLen > k - 2hLen - 2, sortir "message trop long" et arrêter.
2. Codage EME-OAEP (voir la Figure 1 ci-dessous) :
   a. Si l'étiquette L n'est pas fournie, L est la chaîne vide. Soit lHash = Hash(L), une chaîne d'octet de longueur hLen (voir la note ci-dessous).
   b. Générer une chaîne d'octet PS consistant en k - mLen - 2hLen - 2 octets à zéro. La longueur de PS peut être zéro.

c. Concaténer lHash, PS, un seul octet de valeur hexadécimale 0x01, et le message M pour former un bloc de données DB de longueur k -hLen - 1 octets comme DB = lHash || PS || 0x01 || M.
d. Générer une chaîne d'octet aléatoire de germe de longueur hLen.
e. Soit dbMask = MGF(germe, k - hLen - 1).
f. Soit maskedDB = DB \xor dbMask.
g. Soit seedMask = MGF(maskedDB, hLen).
h. Soit maskedSeed = germe \xor seedMask.
i. Concaténer un seul octet de valeur hexadécimale 0x00, maskedSeed, et maskedDB pour former un message EM codé de longueur k octets comme EM = 0x00 || maskedSeed || maskedDB.

3. Chiffrement RSA :
   a. Convertir le message codé EM en une représentation entière de message m (voir au paragraphe 4.2): m = OS2IP (EM).
   b. Appliquer la primitive de chiffrement RSAEP (paragraphe 5.1.1) à la clé publique RSA (n, e) et à la représentation de message m pour produire une représentation entière de texte chiffré c: c = RSAEP ((n, e), m).
   c. Convertir la représentation de texte chiffré c en un texte chiffré C de longueur k octets (voir au paragraphe 4.1) : C = I2OSP (c, k).
4. Sortir le texte chiffré C.

Note. Si L est la chaîne vide, la valeur de hachage correspondante lHash a la représentation hexadécimale suivante pour les différents choix de 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

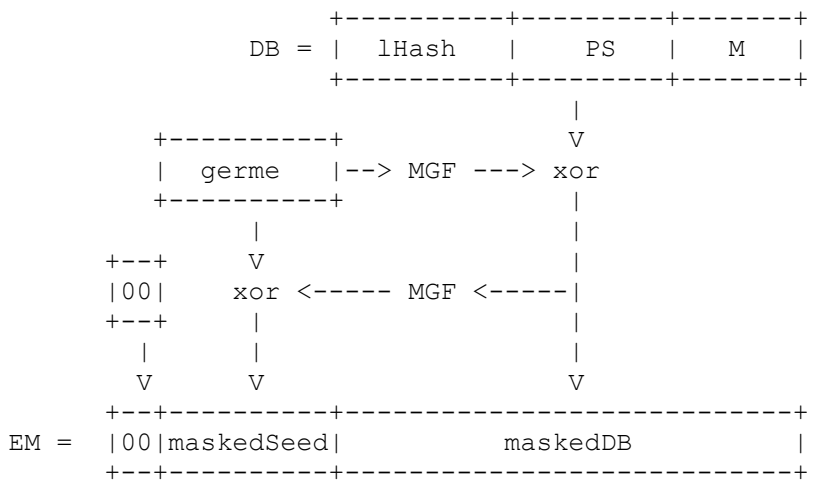

Figure 1 : Opération de codage EME-OAEP. lHash est le hachage de l'étiquette facultative L. L'opération de décodage suit les étapes inverses pour récupérer M et vérifier lHash et PS.

7.1.2 Opération de déchiffrement

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

Options :
Hash : Fonction de hachage (hLen note la longueur en octets du résultat de la fonction de hachage)
MGF : Fonction de génération du gabarit

Entrées :
K : clé privée RSA du receveur (k note la longueur en octets du module n RSA)
C : texte chiffré à déchiffrer, une chaîne d'octet de longueur k, où k = 2hLen + 2
L : étiquette facultative dont l'association au message est à vérifier ; la valeur par défaut pour L, si L n'est pas fourni, est la chaîne vide.

Résultat :
M : message, une chaîne d'octet de longueur mLen, où mLen <= k - 2hLen - 2

Erreur : "erreur de déchiffrement"

Étapes :
1. Vérification de la longueur :
   a. Si la longueur de L est supérieure à la limitation d'entrée pour la fonction de hachage (2^61 - 1 octets pour SHA-1) sortir "erreur de déchiffrement" et arrêter.
   b. Si la longueur du texte chiffré C n'est pas k octets, sortir "erreur de déchiffrement" et arrêter.
   c. Si k < 2hLen + 2, sortir "erreur de déchiffrement" et arrêter.
2. Déchiffrement RSA :
   a. Convertir le texte chiffré C en une représentation de texte chiffré c entière (voir au paragraphe 4.2) : c = OS2IP (C).
   b. Appliquer la primitive de déchiffrement RSADP (paragraphe 5.1.2) à la clé privée RSA K et à la représentation de texte chiffré c pour produire une représentation de message m entière : m = RSADP (K, c).
      Si RSADP sort "représentation de texte chiffré hors gamme" (ce qui signifie que c ≥ n), sortir "erreur de déchiffrement" et arrêter.
   c. Convertir la représentation de message m en un message codé EM de longueur k octets (voir paragraphe 4.1) : EM = I2OSP (m, k).
3. Décodage EME-OAEP :
   a. Si l'étiquette L n'est pas fournie, L est la chaîne vide. Soit lHash = Hash(L) une chaîne d'octet de longueur hLen (voir la note du paragraphe 7.1.1).
   b. Séparer le message codé EM en un seul octet Y, une chaîne d'octet maskedSeed de longueur hLen, et une chaîne d'octet maskedDB de longueur k - hLen - 1 comme EM = Y || maskedSeed || maskedDB.
   c. Soit seedMask = MGF(maskedDB, hLen).
   d. Soit seed = maskedSeed \xor seedMask.
   e. Soit dbMask = MGF(seed, k - hLen - 1).
   f. Soit DB = maskedDB \xor dbMask.
   g. Séparer DB en une chaîne d'octet lHash' de longueur hLen, une chaîne de bourrage (éventuellement vide) PS consistant en octets de valeur hexadécimale 0x00, et un message M comme DB = lHash' || PS || 0x01 || M.
      Si il n'y a pas d'octet de valeur hexadécimale 0x01 pour séparer PS de M, si lHash n'est pas égal à lHash', ou si Y est différent de zéro, sortir "erreur de déchiffrement" et arrêter. (Voir la note ci-dessous.)
4. Sortir le message M.

Note. Il faut veiller à s'assurer qu'un agresseur ne puisse pas distinguer les différentes conditions d'erreur à l'étape 3.g, que ce soit par le message d'erreur ou par le temps écoulé, ou, plus généralement, apprendre des informations partielles sur le message codé EM. Autrement, un agresseur peut être capable d'obtenir des informations utiles sur le déchiffrement du texte chiffré C, ce qui conduit à une attaque de texte chiffré choisi telle que celle observée par Manger [36].

7.2 RSAES-PKCS1-v1_5

RSAES-PKCS1-v1_5 combine les primitives RSAEP et RSADP (paragraphes 5.1.1 et 5.1.2) avec la méthode de codage EME-PKCS1-v1_5 (étape 1 du paragraphe 7.2.1 et étape 3 du paragraphe 7.2.2). Il est mathématiquement équivalent au schéma de chiffrement de PKCS n°1 v1.5. RSAES-PKCS1-v1_5 peut fonctionner sur un messages d'une longueur jusqu'à k - 11 octets (k la longueur en octets du module RSA) bien qu'il faille faire attention à éviter certaines attaques sur les faibles exposants RSA d'après Coppersmith, Franklin, Patarin, et Reiter lorsque de longs messages sont chiffrés (voir la troisième astérisque des notes ci-dessous et [10] ; [14] contient une attaque éprouvée). En règle générale, l'utilisation de ce schéma pour chiffrer un message arbitraire, par opposition à une clé générée au hasard, n'est pas recommandée.

Il est possible de générer des textes chiffrés RSAES-PKCS1-v1_5 valides sans connaître les textes en clair correspondants, avec une raisonnable probabilité de succès. Cette capacité peut être exploitée dans une attaque de texte chiffré choisi comme montré dans [6]. Donc, si RSAES-PKCS1-v1_5 doit être utilisé, certaines contre-mesures faciles à mettre en œuvre devraient être prises pour déjouer l'attaque qu'on trouvera dans [6]. Des exemples typiques sont l'ajout d'une structure aux données à coder, une vérification rigoureuse de la conformité de PKCS #1 v1.5 (et des autres redondances) dans les messages déchiffrés, et la consolidation des messages d'erreur dans un protocole client-serveur fondé sur PKCS n°1 v1.5. Cela peut constituer des contre mesures efficaces et n'implique pas de changement à un protocole fondé sur PKCS n°1 v1.5. Voir dans [7] un exposé des contre-mesures et d'autres. Il a été récemment montré que la sécurité du protocole de prise de contact SSL/TLS [17], qui utilise RSAES-PKCS1-v1_5 et certaines contre mesures, peut être mise en relation avec

une variante du problème de RSA ; voir l'exposé de [32].

Note. Les passages qui suivent décrivent des recommandations de sécurité qui relèvent de l'utilisation de RSAES-PKCS1-v1_5. Les recommandations provenant de la version 1.5 du présent document sont incluses ainsi que de nouvelles recommandations motivées par les avancées de la cryptanalyse ces dernières années.

* Il est recommandé que les octets pseudo aléatoires dans l'étape 2 du paragraphe 7.2.1 soient générés indépendamment dans chaque processus de chiffrement, en particulier si les mêmes données sont entrées dans plus d'un processus de chiffrement. Les résultats de Haastad [24] sont une des motivations de cette recommandation.

* La chaîne de bourrage PS à l'étape 2 du paragraphe 7.2.1 fait au moins huit octets, ce qui est une condition de sécurité pour les opérations de clé publique qui rend difficile à un agresseur la récupération des données en essayant tous les blocs de chiffrement possibles.

* Les octets pseudo aléatoires peuvent aussi aider à déjouer un attaque grâce à Coppersmith et al. [10] (voir dans [14] une amélioration de l'attaque lorsque la taille du message à chiffrer reste petite. L'attaque fonctionne sur RSA à faible exposant lorsque des messages similaires sont chiffrés avec la même clé publique RSA. Plus précisément, dans une version de l'attaque, lorsque deux entrées à RSAEP s'accordent sur une large fraction de bits (8/9) et qu'un faible exposant RSA (e = 3) est utilisé pour les chiffrer tous deux, il est possible de récupérer les deux entrées avec l'attaque. Une autre version de l'attaque réussit à déchiffrer un seul texte chiffré lorsque une grosse fraction (2/3) de l'entrée à RSAEP est déjà connue. Pour des applications normales, le message à chiffrer est court (par exemple, une clé symétrique de 128 bits) de sorte qu'il n'y a pas assez d'informations connues ou communes entre deux messages pour permettre l'attaque. Cependant, si un long message est chiffré, ou si une partie du message est connue, on peut redouter l'attaque. Dans tous les cas, le schéma RSAES-OAEP déjouera l'attaque.

7.2.1 Opération de chiffrement

RSAES-PKCS1-V1_5-ENCRYPT ((n, e), M)

Entrée :
(n, e) : clé publique RSA du receveur (k note la longueur en octets du module n)
M ; message à chiffrer, une chaîne d'octet de longueur mLen, où mLen ≤ k - 11

Résultat :
C : texte chiffré, une chaîne d'octet de longueur k

Erreur : "message trop long"

Étapes :
1. Vérification de la longueur : Si mLen > k - 11, sortir "message trop long" et arrêter.
2. Codage en EME-PKCS1-v1_5 :
   a. Générer une chaîne d'octets PS de longueur k - mLen - 3 consistant en octets différents de zéro générés de façon pseudo-aléatoire. La longueur de PS sera d'au moins huit octets.
   b. Enchaîner PS, le message M, et le bourrage pour former un message EM codé de longueur k octets comme EM = 0x00 || 0x02 || PS || 0x00 || M.
3. Chiffrement RSA :
   a. Convertir le message EM codé en une représentation entière du message m (voir le paragraphe 4.2) : m = OS2IP (EM).
   b. Appliquer la primitive de chiffrement RSAEP (paragraphe 5.1.1) à la clé publique RSA (n, e) et à la représentation du message m pour produire une représentation entière du texte chiffré c : c = RSAEP ((n, e), m).
   c. Convertir la représentation du texte chiffré c en un texte chiffré C de longueur k octets (voir le paragraphe 4.1) : C = I2OSP (c, k).
4. Sortir le texte chiffré C.

7.2.2 Opération de déchiffrement

RSAES-PKCS1-V1_5-DECRYPT (K, C)

Entrée :
K : clé privée RSA du receveur
C : texte chiffré à déchiffrer, une chaîne d'octets de longueur k, où k est la longueur en octets du module RSA n

Résultat :
M : message, une chaîne d'octets de longueur d'au plus k - 11

Erreur : "erreur de déchiffrement"

Étapes :
1. Vérification de la longueur : Si la longueur du texte chiffré C n'est pas k octets (ou si k < 11) sortir "erreur de déchiffrement" et arrêter.
2. Déchiffrement RSA :
   a. Convertir le texte chiffré C en une représentation entière du texte chiffré c (voir le paragraphe 4.2) : c = OS2IP (C).
   b. Appliquer la primitive de déchiffrement RSADP (paragraphe 5.1.2) à la clé privée RSA (n, d) et à la représentation du texte chiffré c pour produire une représentation entière du message m : m = RSADP ((n, d), c).
   Si RSADP sort "représentation de texte chiffré hors gamme" (ce qui signifie que c ≥ n) sortir "erreur de déchiffrement" et arrêter.
   c. Convertir la représentation du message m en un message EM codé de longueur k octets (voir le paragraphe 4.1) : EM = I2OSP (m, k).
3. Décodage EME-PKCS1-v1_5 : Séparer le message codé EM en une chaîne d'octet PS consistant en octets différents de zéro et un message M comme EM = 0x00 || 0x02 || PS || 0x00 || M.
   Si le premier octet de EM n'a pas la valeur hexadécimale 0x00, si le second octet de EM n'a pas la valeur hexadécimale 0x02, si il n'y a pas d'octet avec la valeur hexadécimale 0x00 pour séparer PS de M, ou si la longueur de PS est inférieure à 8 octets, sortir "erreur de déchiffrement" et arrêter. (Voir la note ci-dessous.)
4. Sortir M.

Note. Il faut veiller à s'assurer qu'un agresseur ne peut pas distinguer les différentes conditions d'erreur dans l'étape 3, que ce soit par un message d'erreur ou par le rythme. Autrement, un agresseur pourrait être capable d'obtenir des informations utiles sur le déchiffrement du texte chiffré C, ce qui conduirait à une version renforcée de l'attaque de Bleichenbacher [6] ; à comparer à l'attaque de Manger [36].

8. Schémas de signature avec appendice

Pour les besoins du présent document, un schéma de signature avec appendice consiste en une opération de génération de signature et une vérification de signature, où l'opération de génération de signature produit une signature à partir d'un message avec la clé privée RSA du signataire, et l'opération de vérification de signature vérifie la signature sur le message avec la clé publique RSA correspondante du signataire .Pour vérifier une signature construite avec ce type de schéma, il est nécessaire d'avoir le message lui-même. De cette façon, les schémas de signature avec appendice se distinguent des schémas de signature avec récupération de message, qui ne sont pas pris en charge dans le présent document.

Un schéma de signature avec appendice peut être employé dans diverses applications. Par exemple, les schémas de signature avec appendice définis ici seraient des algorithmes de signature convenables pour les certificats X.509 [28]. De tels schémas de signature pourraient être employés dans PKCS n° 7 [45], bien que pour des raisons techniques, la version actuelle de PKCS n° 7 sépare une fonction de hachage d'un schéma de signature, ce qui est différent de ce qui est fait ici ; voir la note de l'Appendice A.2.3 pour plus d'explications.

Deux schémas de signature avec appendice sont spécifiés dans le présent document : RSASSA-PSS et RSASSA-PKCS1-v1_5. Bien qu'aucune attaque ne soit connue contre RSASSA-PKCS1-v1_5, dans l'intérêt d'une robustesse accrue, RSASSA-PSS est recommandé pour une adoption éventuelle dans les nouvelles applications. RSASSA-PKCS1-v1_5 est inclus pour la compatibilité avec les applications existantes, et bien qu'encore approprié pour les nouvelles applications, une transition graduelle vers RSASSA-PSS est recommandée.

Les schémas de signature avec appendice donnés ici suivent un modèle général similaire à celui employé dans la norme IEEE 1363-2000 [26], combinant les primitives de signature et de vérification avec une méthode de codage pour les signatures. Les opérations de génération de signature appliquent une opération de codage du message à un message pour produire un message codé, qui est alors converti en une représentation du message entier. Une primitive de signature est appliquée à la représentation de message pour produire la signature. En inversant cela, les opérations de vérification de signature appliquent une primitive de vérification de signature à la signature pour récupérer la représentation du message, qui est alors convertie en un message codé en chaîne d'octets. Une opération de vérification est appliquée au message et au message codé pour déterminer si ils sont cohérents.

Si la méthode de codage est déterministe (par exemple, EMSA-PKCS1-v1_5) l'opération de vérification peut appliquer l'opération de codage du message au message et comparer le message codé résultant au message codé déduit

précédemment. Si il y a correspondance, la signature est considérée comme valide. Si la méthode est rendue aléatoire (par exemple, avec EMSA-PSS) l'opération de vérification est normalement plus compliquée. Par exemple, l'opération de vérification dans EMSA-PSS extrait le sel aléatoire et un résultat de hachage du message codé et vérifie si le résultat du hachage, le sel, et le message sont cohérents ; le résultat du hachage est une fonction déterministe en termes de message et de sel.

Pour les deux schémas de signature avec appendice définis dans le présent document, les opérations de génération de signature et de vérification de signature sont directement mises en œuvre comme opérations en "une seule fois" si la signature est placée après le message. Voir dans PKCS n° 7 [45] un exemple de format dans le cas de RSASSA-PKCS1-v1_5.

8.1 RSASSA-PSS

RSASSA-PSS combine les primitives RSASP1 et RSAVP1 avec la méthode de codage EMSA-PSS. Il est compatible avec le schéma IFSSA tel qu'amendé par le projet IEEE P1363a [27], où les primitives de signature et vérification sont IFSP-RSA1 et IFVP-RSA1 comme défini dans la norme IEEE 1363-2000 [26] et la méthode de codage du message est EMSA4. EMSA4 est légèrement plus général que EMSA-PSS car il agit sur des chaînes de bits plutôt que sur des chaînes d'octets. EMSA-PSS est équivalent à EMSA4 restreint au cas où les opérandes ainsi que les valeurs de hachage et de sel sont des chaînes d'octets.

La longueur des messages sur lesquels RSASSA-PSS peut opérer est soit sans restriction, soit contrainte par un très grand nombre, selon la fonction de hachage sous-jacente à la méthode de codage EMSA-PSS.

En supposant que le calcul des racines $e^{ème}$ modulo n est infaisable et que les fonctions de hachage et de génération de gabarit dans EMSA-PSS ont les propriétés appropriées, RSASSA-PSS fournit des signatures sûres. Cette assurance est prouvable en ce sens que la difficulté de falsifier des signatures peut être mise en relation directe avec la difficulté d'intervertir la fonction RSA, pourvu que les fonctions de hachage et de génération de gabarit soient vues comme des boîtes noires ou des oracles aléatoires. Les limites de la preuve de sécurité sont essentiellement "serrées", ce qui signifie que la probabilité de succès et de temps de fonctionnement pour le meilleur falsificateur contre RSASSA-PSS sont très proches des paramètres correspondants pour le meilleur algorithme d'inversion RSA ; voir un exposé plus détaillé dans [4], [13], et [31].

À la différence du schéma de signature RSASSA-PKCS1-v1_5, un identifiant de fonction de hachage n'est pas incorporé dans le message codé EMSA-PSS, de sorte qu'en théorie, il est possible à un adversaire de substituer une fonction de hachage différente (et éventuellement plus faible) que celle choisie par le signataire. Il est donc recommandé que la fonction de génération de gabarit EMSA-PSS se fonde sur la même fonction de hachage. De cette manière, le message codé entier va dépendre de la fonction de hachage et il sera difficile à un agresseur de substituer une fonction de hachage différente de celle prévue par le signataire. Cette correspondance des fonctions de hachage n'est destinée qu'à empêcher une substitution de fonction de hachage, et n'est pas nécessaire si la substitution de fonction de hachage est traitée par d'autres moyens (par exemple, si le vérificateur n'accepte qu'une fonction de hachage désignée). Voir dans [34] un exposé complémentaire sur ces points. La sécurité prouvable de RSASSA-PSS ne s'appuie pas sur la similarité de la fonction de hachage dans la fonction de génération de gabarit et dans celle appliquée au message.

RSASSA-PSS est différent des autres schémas de signature fondés sur RSA en ce qu'il est probabiliste plutôt que déterministe, incorporant une valeur de sel générée de façon aléatoire. La valeur du sel améliore la sécurité du schéma en permettant une preuve de sécurité "plus serrée" que les solutions de remplacement déterministes telles que le hachage de domaine entier (FDH, *Full Domain Hashing*) ; voir l'exposé dans [4]. Cependant, l'aléa n'est pas critique pour la sécurité. Dans les situations où la génération aléatoire n'est pas possible, une valeur fixe ou un numéro de séquence pourrait être employé à la place, avec une sécurité prouvable résultante similaire à celle de FDH [12].

8.1.1 Opération de génération de signature

RSASSA-PSS-SIGN (K, M)

Entrée :
K : clé privée RSA du signataire
M : message à signer, une chaîne d'octets

Résultat :
S : signature, une chaîne d'octets de longueur k, où k est la longueur en octets du module RDA n

Erreurs :"message trop long", "erreur de codage"

Étapes :

1. Codage EMSA-PSS : Appliquer l'opération de codage EMSA-PSS (paragraphe 9.1.1) au message M pour produire un message codé EM de longueur \plafond ((modBits - 1)/8) octets telle que la longueur en bits de l'entier OS2IP (EM) (voir le paragraphe 4.2) soit au plus modBits - 1, où modBits est la longueur en bits du module RSA n : EM = EMSA-PSS-ENCODE (M, modBits - 1).
   Noter que la longueur en octets de EM sera de un de moins que k si modBits - 1 est divisible par 8 et égal à k autrement. Si l'opération de codage sort "message trop long", sortir "message trop long" et arrêter. Si l'opération de codage sort "erreur de codage", sortir "erreur de codage" et arrêter.
2. Signature RSA :
   a. Convertir le message codé EM en une représentation entière du message m (voir le paragraphe 4.2) : m = OS2IP (EM).
   b. Appliquer la primitive de signature RSASP1 (paragraphe 5.2.1) à la clé privée RSA K et à la représentation de message m pour produire une représentative de signature entière s : s = RSASP1 (K, m).
   c. Convertir la représentative de signature s à la signature S de longueur k octets (voir au paragraphe 4.1) : S = I2OSP (s, k).
3. Sortir la signature S.

8.1.2 Opération de vérification de signature

RSASSA-PSS-VERIFY ((n, e), M, S)

Entrée :

(n, e) : clé publique RSA du signataire
M : message dont la signature est à vérifier, une chaîne d'octets
S : signature à vérifier, une chaîne d'octets de longueur k, où k est la longueur en octets du module RSA n

Résultat : "signature valide" ou "signature invalide"

Étapes :

1. Vérification de longueur : Si la longueur de la signature S n'est pas k octets, sortir "signature invalide" et arrêter.
2. Vérification RSA :
   a. Convertir la signature S en une représentation entière de signature s (voir au paragraphe 4.2) :s = OS2IP (S).
   b. Appliquer la primitive de vérification RSAVP1 (paragraphe 5.2.2) à la clé publique RSA (n, e) et à la représentation de signature s pour produire une représentation entière du message m : m = RSAVP1 ((n, e), s).
      Si RSAVP1 sort "représentation de signature hors gamme", sortir "signature invalide" et arrêter.
   c. Convertir la représentation de message m en un message codé EM de longueur emLen = \plafond ((modBits - 1)/8) octets, où modBits est la longueur en bits du module RSA n (voir le paragraphe 4.1) : EM = I2OSP (m, emLen).
      Noter que emLen sera de un de moins que k si modBits - 1 est divisible par 8 et égal à k autrement. Si I2OSP sort "entier trop grand", sortir "signature invalide" et arrêter.
3. Vérification EMSA-PSS : Appliquer l'opération de vérification EMSA-PSS (paragraphe 9.1.2) au message M et au message codé EM pour déterminer si ils sont cohérents : Résultat = EMSA-PSS-VERIFY (M, EM, modBits - 1).
4. Si Résultat = "cohérent", sortir "signature valide". Autrement, sortir "signature invalide".

8.2 RSASSA-PKCS1-v1_5

RSASSA-PKCS1-v1_5 combine les primitives RSASP1 et RSAVP1 avec la méthode de codage EMSA-PKCS1-v1_5. Il est compatible avec le schéma IFSSA défini dans la norme IEEE 1363-2000 [26], où les primitives de signature et de vérification sont IFSP-RSA1 et IFVP-RSA1 et la méthode de codage du message est EMSA-PKCS1-v1_5 (qui n'est pas définie dans la norme IEEE 1363-2000, mais est dans le projet IEEE P1363a [27]).

La longueur des messages sur lesquels RSASSA-PKCS1-v1_5 peut opérer est soit non restreinte, soit contrainte par un très grand nombre, selon la fonction de hachage sous-jacente à la méthode EMSA-PKCS1-v1_5.

En supposant qu'il est infaisable de calculer la e^{ème} racine modulo n et que la fonction de hachage dans EMSA-PKCS1-v1_5 a les propriétés appropriées, RSASSA-PKCS1-v1_5 est censé fournir des signatures sûres. Plus précisément, la falsification de signatures sans connaître la clé privée RSA est supposé être impossible à calculer. Aussi, dans la méthode de codage EMSA-PKCS1-v1_5, un identifiant de fonction de hachage est incorporé dans le codage. À cause de ce dispositif, un adversaire qui essaye de trouver un message avec la même signature qu'un message signé précédemment doit

trouver des collisions de la fonction de hachage particulière qui est utilisée ; attaquer une fonction de hachage différente de celle choisie par le signataire est sans utilité pour l'agresseur. Voir les détails dans [34].

Note. Comme noté dans PKCS n° 1 v1.5, la méthode de codage EMSA-PKCS1-v1_5 a comme propriété que le message codé, converti en une représentation de message entière, a la garantie d'être grand et au moins dans une certaine mesure "aléatoire".Cela empêche les attaques de la sorte proposée par Desmedt et Odlyzko [16] où des relations multiplicatives entre représentations de message sont développées en factorisant les représentations de message dans un ensemble de petites valeurs (par exemple, un ensemble de petits nombres premiers). Coron, Naccache, et Stern [15] ont montré qu'une forme plus forte de ce type d'attaque pourrait être assez efficace contre certaines instances du schéma de signature ISO/CEI 9796-2. Ils ont aussi analysé la complexité de ce type d'attaque contre la méthode de codage EMSA-PKCS1-v1_5 et ont conclu qu'une attaque serait impraticable, exigeant plus d'opérations qu'une recherche de collision sur la fonction de hachage sous-jacent (c'est-à-dire, plus de 2^80 opérations). Coppersmith, Halevi, et Jutla [11] ont ultérieurement étendu l'attaque de Coron et autres pour casser le schéma de signature ISO/CEI 9796-1 avec récupération de message. Les diverses attaques illustrent l'importance d'une construction soigneuse des entrées d'une primitive de signature RSA, en particulier dans un schéma de signature avec récupération de message. En conséquence, la méthode de codage EMSA-PKCS-v1_5 inclut explicitement une opération de hachage et n'est pas destinée aux schémas de signature avec récupération de message. De plus, bien qu'aucune attaque ne soit connue contre la méthode de codage EMSA-PKCS-v1_5, une transition graduelle vers EMSA-PSS est recommandée comme précaution contre de futurs développements.

8.2.1 Opération de génération de signature

RSASSA-PKCS1-V1_5-SIGN (K, M)

Entrée :
K : clé privée RSA du signataire
M : message à signer, une chaîne d'octets

Résultat :
S : signature, une chaîne d'octets de longueur k, où k est la longueur en octets du module RSA n

Erreurs : "message trop long" ; "module RSA trop court"

Étapes :
1. Codage EMSA-PKCS1-v1_5 : Appliquer l'opération de codage EMSA-PKCS1-v1_5 (paragraphe 9.2) au message M pour produire un message codé EM de longueur k octets : EM = EMSA-PKCS1-V1_5-ENCODE (M, k).
   Si l'opération de codage sort "message trop long", sortir "message trop long" et arrêter. Si l'opération de codage sort "longueur prévue du message codé trop courte", sortir "module RSA trop court" et arrêter.
2. Signature RSA :
   a. Convertir le message codé EM en une représentation entière du message m (voir au paragraphe 4.2) : m = OS2IP (EM).
   b. Appliquer la primitive de signature RSASP1 (paragraphe 5.2.1) à la clé privée RSA K et à la représentation de message m pour produire une représentation entière de signature s : s = RSASP1 (K, m).
   c. Convertir la représentation de signature s en une signature S de longueur k octets (voir au paragraphe 4.1) : S = I2OSP (s, k).
3. Sortir la signature S.

8.2.2 Opération de vérification de signature

RSASSA-PKCS1-V1_5-VERIFY ((n, e), M, S)

Entrée :
(n, e) : clé publique RSA du signataire
M : message dont la signature est à vérifier, une chaîne d'octets
S : signature à vérifier, une chaîne d'octets de longueur k, où k est la longueur en octets du module RSA n

Résultat : "signature valide" ou "signature invalide"

Erreurs : "message trop long" ; "module RSA trop court"

Étapes :
1. Vérification de longueur : si la longueur de la signature S n'est pas k octets, sortir "signature invalide" et arrêter.

2. Vérification RSA :
   a. Convertir la signature S en une représentation de signature entière s (voir le paragraphe 4.2) : s = OS2IP (S).
   b. Applique la primitive de vérification RSAVP1 (paragraphe 5.2.2) à la clé publique RSA (n, e) et à la représentation de signature s pour produire une représentation entière du message m : m = RSAVP1 ((n, e), s).
      Si RSAVP1 sort "représentation de signature hors gamme" sortir "signature invalide" et arrêter.
   c. Convertir la représentation de message m en un message codé EM de longueur k octets (voir au paragraphe 4.1) : EM' = I2OSP (m, k).
      Si I2OSP sort "entier trop grand", sortir "signature invalide" et arrêter.
3. Codage EMSA-PKCS1-v1_5 : Appliquer l'opération de codage EMSA-PKCS1-v1_5 (paragraphe 9.2) au message M pour produire un second message codé EM' de longueur k octets : EM' = EMSA-PKCS1-V1_5-ENCODE (M, k).
   Si l'opération de codage sort "message trop long", sortir "message trop long" et arrêter. Si l'opération de codage sort "longueur du message codé prévu trop courte", sortir "module RSA trop court" et arrêter.
4. Comparer le message codé EM et le second message codé EM'. Si ils sont identiques, sortir "signature valide" ; autrement, sortir "signature invalide".

Note. Une autre façon de mettre en œuvre l'opération de vérification de signature est d'appliquer une opération "de décodage" (non spécifiée dans le présent document) au message codé pour récupérer la valeur du hachage sous-jacent, et de la comparer à un nouveau calcul de la valeur de hachage. Cela présente l'avantage d'exiger moins de mémorisation intermédiaire (deux valeurs de hachage plutôt que deux messages codés) mais a l'inconvénient d'exiger plus de code.

9. Méthodes de codage pour signatures avec appendice

Les méthodes de codage consistent en opérations qui transposent entre messages en chaîne d'octets et messages codés en chaîne d'octets, qui sont convertis en, et de, représentations entières de messages dans les schémas. Les représentations entières de message sont traitées via les primitives. Les méthodes de codage fournissent donc la connexion entre les schémas, qui traitent les messages, et les primitives.

Une méthode de codage pour les signatures avec appendice, pour les besoins du présent document, consiste en une opération de codage et facultativement une opération de vérification. Une opération de codage transpose un message M en un message codé EM d'une longueur spécifiée. Une opération de vérification détermine si un message M et un message codé EM sont cohérents, c'est-à-dire, si le message codé EM est un codage valide du message M.

L'opération de codage peut introduire une certaine dose d'aléa, afin que des applications différentes de l'opération de codage au même message produisent des messages codés différents, ce qui est avantageux pour une sécurité démontrable. Pour une telle méthode de codage, une opération de codage et une opération de vérification sont toutes deux nécessaires sauf si le vérificateur peut reproduire l'aléation (par exemple, en obtenant la valeur du sel du signataire). Pour une méthode de codage déterministe, seule une opération de codage est nécessaire.

Deux méthodes de codage pour les signatures avec appendice sont employées dans les schémas de signature et sont spécifiés ici : EMSA-PSS et EMSA-PKCS1-v1_5.

9.1 EMSA-PSS

Cette méthode de codage est paramétrée par le choix d'une fonction de hachage, d'une fonction de génération de gabarit, et par la longueur du sel. Ces options devraient être fixées pour chaque clé RSA, sauf que la longueur du sel peut être variable (voir l'exposé dans [31]). Les fonctions de hachage et de génération de gabarit suggérées sont données à l'Appendice B. La méthode de codage est fondée sur le schéma de signature probabiliste (PSS) de Bellare et Rogaway [4], [5]. Il est au hasard et a une opération de codage et une opération de vérification.

La Figure 2 illustre l'opération de codage.

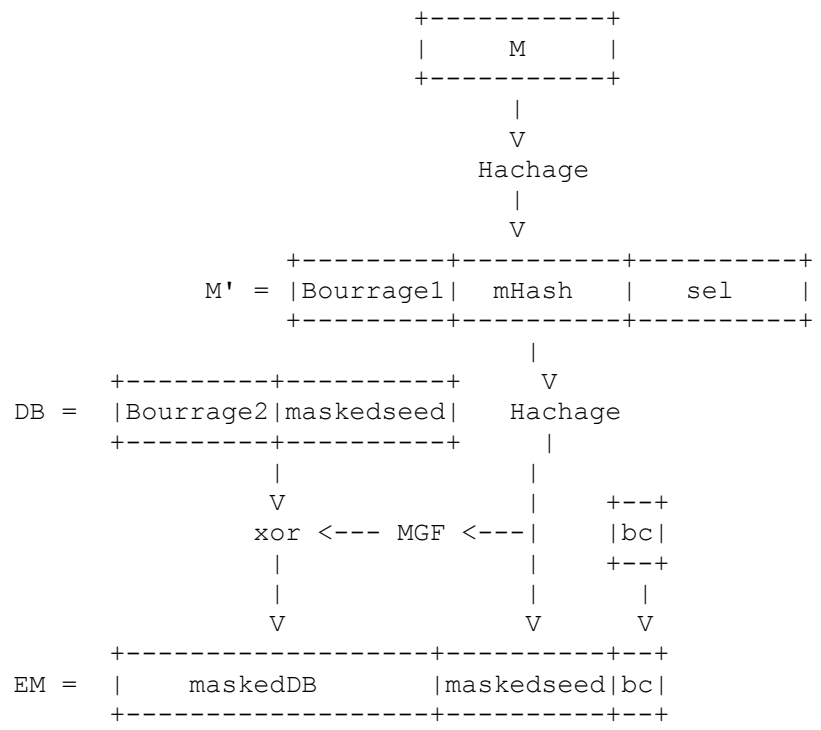

Figure 2 : Opération de codage EMSA-PSS. L'opération de vérification suit l'ordre inverse pour récupérer le sel, puis suit ces étapes pour recalculer et comparer H.

Notes.

1. La méthode de codage définie ici diffère de celle de la soumission à IEEE P1363a [5] de Bellare et Rogaway sous trois aspects :
   * Elle applique une fonction de hachage plutôt qu'une fonction de génération de gabarit au message. Bien que la fonction de génération de gabarit se fonde sur une fonction de hachage, il semble plus naturel d'appliquer directement la fonction de hachage.
   * La valeur qui est hachée avec la valeur du sel est la chaîne (0x)00 00 00 00 00 00 00 00 || mHash plutôt que le message M lui-même. Ici, mHash est le hachage de M. Noter que la fonction de hachage est la même dans les deux étapes. Voir la note 3 ci-dessous pour un exposé plus poussé. (Aussi, le nom de "sel" est utilisé au lieu de celui de "germe", car il reflète mieux le rôle de la valeur.)
   * Le message codé dans EMSA-PSS a neuf bits fixés ; le premier bit est 0 et les huit derniers bits forment un "champ d'en-queue", l'octet 0xbc. Dans le schéma d'origine, seul le premier bit est fixé. La raison du champ d'en-queue est la compatibilité avec la primitive de signature IFSP-RW de Rabin-Williams dans la norme IEEE 1363-2000 [26] et la primitive correspondante dans le projet ISO/IEC 9796-2 [29].

2. En supposant que la fonction de génération de gabarit se fonde sur une fonction de hachage, il est recommandé que la fonction de hachage soit la même que celle appliquée au message ; voir les explications complémentaires au paragraphe 8.1.

3. Sans compromettre la preuve de sécurité pour RSASSA-PSS, on peut effectuer les étapes 1 et 2 de EMSA-PSS-ENCODE et EMSA-PSS-VERIFY (l'application de la fonction de hachage au message) en dehors du module qui calcule le reste de l'opération de signature, afin que mHash soit entré dans le module plutôt que le message M lui-même. En d'autres termes, la preuve de sécurité pour RSASSA-PSS tient quand même si un agresseur peut contrôler la valeur de mHash. Ceci est pratique si le module a une bande passante d'entrée/sortie limitée, par exemple, une carte à puce. Noter que les versions antérieures de PSS [4], [5] n'avaient pas cette propriété. Bien sûr, il peut être souhaitable pour d'autres raisons de sécurité de faire que le module traite le message complet. Par exemple, le module peut avoir besoin de "voir" ce qu'il signe si il ne fait pas confiance aux composants qui calculent la valeur du hachage.

4. Les longueurs de sel normales en octets sont hLen (la longueur du résultat de la fonction de hachage Hash) et 0. Dans les deux cas, la sécurité de RSASSA-PSS peut être mise en relation étroite avec la difficulté d'inverser RSAVP1. Bellare et Rogaway [4] donnent une limite inférieure légère pour la sécurité du schéma RSA-PSS original qui correspond en gros au cas précédent, alors que Coron [12] donne une limite inférieure pour le schéma de hachage de domaine complet qui s'y rapporte, qui correspond en gros au premier cas. Dans [13] Coron donne un traitement général avec diverses longueurs de sel allant de 0 à hLen ; voir l'exposé dans [27]. Voir aussi [31], qui adapte les preuves de

sécurité fournies dans [4] et [13] pour traiter les différences entre les versions originale et présente de RSA-PSS telles qu'énumérées dans la note 1 ci-dessus.

5. Comme noté dans IEEE P1363a [27], l'utilisation de l'aléation dans les schémas de signature – telle que la valeur de sel dans EMSA-PSS – peut fournir un "canal secret" pour transmettre des informations autres que le message à signer. Pour en savoir plus sur les canaux secrets, voir [50].

9.1.1 Opération de codage

EMSA-PSS-ENCODE (M, emBits)

Options :
Hash : Fonction de hachage (hLen note la longueur en octets du résultat de la fonction de hachage)
MGF : Fonction de génération de gabarit
sLen : Longueur du sel prévue en octets

Entrée :
M : Message à coder, une chaîne d'octets
emBits : Longueur maximale en bits de l'entier OS2IP (EM) (voir le paragraphe 4.2) au moins 8hLen + 8sLen + 9

Résultat :
EM : Message codé, une chaîne d'octets de longueur emLen = \plafond (emBits/8)

Erreurs : "erreur de codage" ; "message trop long"

Étapes :
1. Si la longueur de M est supérieure à la limitation d'entrée pour la fonction de hachage (2^61 - 1 octets pour SHA-1) sortir "message trop long" et arrêter.
2. Soit mHash = Hash(M) une chaîne d'octets de longueur hLen.
3. Si emLen < hLen + sLen + 2, sortir "erreur de codage" et arrêter.
4. Générer une chaîne d'octet aléatoire de sel de longueur sLen ; si sLen = 0, le sel est alors la chaîne vide.
5. Soit M' = (0x)00 00 00 00 00 00 00 00 || mHash || sel ; M' est une chaîne d'octet de longueur 8 + hLen + sLen avec huit octets initiaux à zéro.
6. Soit H = Hash(M') une chaîne d'octet de longueur hLen.
7. Générer une chaîne d'octet PS consistant en emLen - sLen - hLen - 2 octets à zéro. La longueur de PS peut être 0.
8. Soit DB = PS || 0x01 || sel ; DB est une chaîne d'octets de longueur emLen - hLen - 1.
9. Soit dbMask = MGF(H, emLen - hLen - 1).
10. Soit maskedDB = DB \xor dbMask.
11. Soient les 8 emLen - emBits bits les plus à gauche de l'octet de gauche dans maskedDB à zéro.
12. Soit EM = maskedDB || H || 0xbc.
13. Sortir EM.

9.1.2 Opération de vérification

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

Options :
Hash Fonction de hachage (hLen note la longueur en octets du résultat de la fonction de hachage)
MGF Fonction de génération de gabarit
sLen longueur du sel prévue en octets

Entrée :
M message à vérifier, une chaîne d'octets
EM message codé, une chaîne d'octets de longueur emLen = \plafond (emBits/8)
emBits longueur maximale en bits de l'entier OS2IP (EM) (voir le paragraphe 4.2) au moins 8hLen + 8sLen + 9

Résultat : "cohérent" ou "incohérent"

Étapes :
1. Si la longueur de M est supérieure à la limitation d'entrée pour la fonction de hachage (2^61 - 1 octets pour SHA-1) sortir "incohérent" et arrêter.
2. Soit mHash = Hash(M), une chaîne d'octets de longueur hLen.

3. Si emLen < hLen + sLen + 2, sortir "incohérent" et arrêter.
4. Si l'octet de droite de EM n'a pas la valeur hexadécimale 0xbc, sortir "incohérent" et arrêter.
5. Soit maskedDB les emLen - hLen - 1 octets les plus à gauche de EM, et soit H les hLen octets suivants.
6. Si les 8emLen - emBits bits plus à gauche de l'octet de gauche de maskedDB ne sont pas tous égaux à zéro, sortir "incohérent" et arrêter.
7. Soit dbMask = MGF(H, emLen - hLen - 1).
8. Soit DB = maskedDB \xor dbMask.
9. Régler les 8emLen - emBits bits les plus à gauche de l'octet de gauche de DB à zéro.
10. Si les emLen - hLen - sLen - 2 octets les plus à gauche de DB ne sont pas zéro ou si l'octet à la position emLen - hLen - sLen - 1 (la position la plus à gauche est la "position 1") n'a pas la valeur hexadécimale 0x01, sortir "incohérent" et arrêter.
11. Soit les derniers sLen octets de DB le sel.
12. Soit M' = (0x)00 00 00 00 00 00 00 00 || mHash || sel ; M' est une chaîne d'octet de longueur 8 + hLen + sLen avec huit octets à zéro initiaux.
13. Soit H' = Hash(M'), une chaîne d'octets de longueur hLen.
14. Si H = H', sortir "cohérent". Autrement, sortir "incohérent."

9.2 EMSA-PKCS1-v1_5

Cette méthode de codage est déterministe et a seulement une opération de codage.

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

Option :

Hash Fonction de hachage (hLen note la longueur en octets du résultat de la fonction de hachage)

Entrée :

M Message à coder

emLen Longueur prévue en octets du message codé, au moins tLen + 11, où tLen est la longueur d'octets du codage DER T d'une certaine valeur calculée durant l'opération de codage

Résultat :

EM Message codé, une chaîne d'octets de longueur emLen

Erreurs : "message trop long" ; "longueur prévue du message codé trop courte"

Étapes :

1. Appliquer la fonction de hachage au message M pour produire une valeur de hachage H : H = Hash(M).
   Si la fonction de hachage sort "message trop long", sortir "message trop long" et arrêter.
2. Coder l'identifiant d'algorithme pour la fonction de hachage et la valeur de hachage dans la valeur ASN.1 de type DigestInfo (voir à l'Appendice A.2.4) avec les règles de codage distinctives (DER) où le type DigestInfo a la syntaxe :

   ```
   DigestInfo ::= SEQUENCE {
       digestAlgorithm     AlgorithmIdentifier,
       digest              CHAINE D'OCTETS
   }
   ```

   Le premier champ identifie la fonction de hachage et le second contient la valeur du hachage. Soit T le codage en DER de la valeur de DigestInfo (voir les notes ci-dessous) et soit tLen la longueur en octets de T.
3. Si emLen < tLen + 11, sortir "longueur prévue du message codé trop courte" et arrêter.
4. Générer une chaîne d'octet PS consistant en emLen - tLen - 3 octets avec la valeur hexadécimale 0xff. La longueur de PS sera au moins de 8 octets.
5. Enchaîner PS, le codage DER T, et autre bourrage, pour former le message codé EM comme : EM = 0x00 || 0x01 || PS || 0x00 || T.
6. Sortir EM.

Notes :

1. Pour les six fonctions de hachage mentionnées à l'Appendice B.1, le codage DER T de la valeur DigestInfo est égal à ce qui suit :

   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. Dans la version 1.5 du présent document, T était défini comme codage BER plutôt que comme codage DER de la valeur

DigestInfo. En particulier, il est possible – au moins en théorie – que l'opération de vérification définie dans le présent document (ainsi que dans la version 2.0) rejette une signature qui serait valide à l'égard de la spécification donnée dans PKCS n°1 v1.5. Cela se produit si d'autres règles que DER sont appliquées à DigestInfo (par exemple, une longueur de codage indéfinie du type de SEQUENCE sous-jacent). Bien qu'il soit peu probable que cela pose un problème pratique, une mise en œuvre précautionneuse peut choisir d'employer une opération de vérification sur la base d'une opération de décodage BER comme spécifié dans PKCS n° 1 v1.5. De cette manière, la compatibilité avec toute mise en œuvre valide fondée sur PKCS n° 1 v1.5 est obtenue. Une telle opération de vérification devrait indiquer si le codage BER sous-jacent est un codage DER et donc si la signature est valide par rapport à la spécification donnée dans le présent document.

Appendice A : Syntaxe ASN.1

A.1 Représentation des clés RSA

Cette section définit les identifiants d'objet (OID, *Object Identifier*) ASN.1 pour les clés RSA publiques et privées, et définit les types RSAPublicKey et RSAPrivateKey. L'application prévue pour ces définitions inclut les certificats X.509, PKCS n° 8 [46], et PKCS n° 12 [47].

L'identifiant d'objet rsaEncryption identifie les clés RSA publiques et privées comme défini dans les appendices A.1.1 et A.1.2. Le champ Paramètres associé à cet OID dans une valeur de type AlgorithmIdentifier devra avoir une valeur de type NUL.

IDENTIFIANT D'OBJET rsaEncryption ::= { pkcs-1 1 }

Les définitions de cette section ont été étendues pour la prise en charge de RSA à plusieurs nombres premiers, mais elles sont rétro compatibles avec les versions précédentes.

A.1.1 Syntaxe de clé publique RSA

Une clé publique RSA devrait être représentée avec le type ASN.1 RSAPublicKey :

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

Les champs de type RSAPublicKey ont la signification suivante :

* modulus est le module RSA n.
* publicExponent est l'exposant public RSA e.

A.1.2 Syntaxe de clé privée RSA

Une clé privée RSA devrait être représentée avec le type ASN.1 RSAPrivateKey :

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

Les champs de type RSAPrivateKey ont la signification suivante :

* version est le numéro de version, pour la compatibilité avec les futures révisions de ce document. Ce sera 0 pour cette version du document, sauf si plusieurs nombres premiers sont utilisés, auquel cas ce devra être 1.

```
   Version ::= ENTIER { deux-premiers(0), multi(1) }
     (CONTRAINT PAR
      {-- version doit être multi si otherPrimeInfos est présent --})
```

* modulus est le module RSA n.
* publicExponent est l'exposant public RSA e.
* privateExponent est l'exposant privé RSA d.
* prime1 est le facteur premier p de n.
* prime2 est le facteur premier q de n.
* exponent1 est d mod (p - 1).
* exponent2 est d mod (q - 1).
* coefficient est le coefficient CRT q^(-1) mod p.
* otherPrimeInfos contient les informations pour les premiers supplémentaires r_3, ..., r_u, dans l'ordre. Il devra être omis si la version est 0 et devra contenir au moins une instance de OtherPrimeInfo si version est 1.

```
OtherPrimeInfos ::= TAILLE DE SEQUENCE (1..MAX) DE OtherPrimeInfo
    OtherPrimeInfo ::= SEQUENCE {
       prime             ENTIER,  -- ri
       exponent          ENTIER,  -- di
       coefficient       ENTIER   -- ti
      }
```

Les champs de type OtherPrimeInfo ont la signification suivante :

* prime est un facteur premier r_i of n, où $i \geq 3$.
* exponent est d_i = d mod (r_i - 1).
* coefficient est le coefficient CRT t_i = (r_1 * r_2 * ... * r_(i- 1))^(-1) mod r_i.

Note. Il est important de protéger la clé privée RSA contre la divulgation et la modification. Les techniques pour une telle protection sortent du domaine d'application du présent document. Les méthodes de mémorisation et distribution de clés privées et autres données cryptographiques sont décrites dans PKCS n° 12 et n° 15.

A.2 Identification de schéma

Cette section définit les identifiants d'objet pour les schémas de chiffrement et de signature. Les schémas compatibles avec PKCS n° 1 v1.5 ont les mêmes définitions que dans PKCS n°1 v1.5. L'application prévue pour ces définitions inclut les certificats X.509 et PKCS n° 7.

Définitions d'identifiant de type pour les OID PKCS n° 1 :

```
IDENTIFIANT-D'ALGORITHME  PKCS1Algorithms        ::= {
     { OID rsaEncryption                  PARAMETERS NUL } |
     { OID md2WithRSAEncryption           PARAMETERS NUL } |
     { OID md5WithRSAEncryption           PARAMETERS NUL } |
     { OID sha1WithRSAEncryption          PARAMETERS NUL } |
     { OID sha256WithRSAEncryption        PARAMETERS NUL } |
     { OID sha384WithRSAEncryption        PARAMETERS NUL } |
     { OID sha512WithRSAEncryption        PARAMETERS NUL } |
     { OID id-RSAES-OAEP                  PARAMETERS RSAES-OAEP-params } | PKCS1PSourceAlgorithms |
     { OID id-RSASSA-PSS                  PARAMETERS RSASSA-PSS-params }
 , ... – Permet de futures expansions --
   }
```

A.2.1 RSAES-OAEP

L'identifiant d'objet id-RSAES-OAEP identifie le schéma de chiffrement RSAES-OAEP.

IDENTIFIANT D'OBJET id-RSAES-OAEP ::= { pkcs-1 7 }

Le champ Paramètres associé à cet OID dans une valeur de type AlgorithmIdentifier devra avoir une valeur de type RSAES-OAEP-params :

```
RSAES-OAEP-params ::= SEQUENCE {
```

```
    hashAlgorithm       [0] HashAlgorithm       DEFAUT sha1,
    maskGenAlgorithm    [1] MaskGenAlgorithm    DEFAUT mgf1SHA1,
    pSourceAlgorithm    [2] PSourceAlgorithm    DEFAUT pSpecifiedEmpty
  }
```

Les champs de type RSAES-OAEP-params ont la signification suivante :

* hashAlgorithm identifie la fonction de hachage. Il devra être un identifiant d'algorithme avec un OID dans l'ensemble OAEP-PSSDigestAlgorithms. Pour un exposé des fonctions de hachage prises en charge, voir l'Appendice B.1.

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

IDENTIFIANT-D'ALGORITHME   OAEP-PSSDigestAlgorithms  ::= {
       { OID id-sha1          PARAMETERS NUL   }|
       { OID id-sha256        PARAMETERS NUL }|
       { OID id-sha384        PARAMETERS NUL }|
       { OID id-sha512        PARAMETERS NUL },
       ...  – Permet une future expansion --
    }
```

La fonction de hachage par défaut est SHA-1 :

```
sha1      HashAlgorithm ::= {
   algorithme     id-sha1,
   paramètres     SHA1Parameters : NULL
   }

SHA1Parameters ::= NULL
```

* maskGenAlgorithm identifie la fonction de génération du gabarit. Il devra être un identifiant d'algorithme avec un OID dans l'ensemble PKCS1MGFAlgorithms, qui pour cette version devra consister en id-mgf1, identifiant la fonction de génération de gabarit MGF1 (voir à l'Appendice B.2.1). Le champ Paramètres associé à id-mgf1 devra être un identifiant d'algorithme avec un OID dans l'ensemble OAEP-PSSDigestAlgorithms, identifiant la fonction de hachage sur laquelle se fonde MGF1.

```
MaskGenAlgorithm ::= AlgorithmIdentifier {
     {PKCS1MGFAlgorithms}
   }
   PKCS1MGFAlgorithms    ALGORITHM-IDENTIFIER ::= {
      { OID id-mgf1 PARAMETERS HashAlgorithm },
      ...  – Permet une future expansion --
   }
```

La fonction de génération de gabarit par défaut est MGF1 avec SHA-1 :

```
mgf1SHA1    MaskGenAlgorithm ::= { algorithme   id-mgf1, paramètres  HashAlgorithm : sha1
   }
```

* pSourceAlgorithm identifie la source (et éventuellement la valeur) de l'étiquette L. Il devra être un identifiant d'algorithme avec un OID dans l'ensemble PKCS1PSourceAlgorithms, qui pour cette version doit consister en id-pSpecified, qui indique que l'étiquette est spécifiée explicitement. Le champ Paramètres associé à id-pSpecified devra avoir une valeur de type CHAINE D'OCTET, contenant l'étiquette. Dans les précédentes versions de cette spécification, le terme "paramètres de codage" était utilisé plutôt que "étiquette", d'où, le nom de type dessous.

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

IDENTIFIANT D'ALGORITHME   PKCS1PSourceAlgorithms  ::= {
      { OID id-pSpecified PARAMETERS EncodingParameters },
      ...  – Permet une future expansion --
       }
```

```
IDENTIFIANT D'OBJET  id-pSpecified ::= { pkcs-1 9 }

EncodingParameters ::= CHAINE D'OCTET(TAILLE(0..MAX))
```

L'étiquette par défaut est une chaîne vide (afin que lHash contienne le hachage de la chaîne vide) :

```
pSpecifiedEmpty    PSourceAlgorithm ::= {
      algorithme     id-pSpecified,
      paramètres     EncodingParameters : emptyString
      }

   emptyString      EncodingParameters ::= "H
```

Si toutes les valeurs par défaut des champs dans RSAES-OAEP-params sont utilisées, alors l'identifiant d'algorithme aura la valeur suivante :

```
rSAES-OAEP-Default-Identifier    RSAES-AlgorithmIdentifier ::= {
      algorithme                 id-RSAES-OAEP,
      paramètres                 RSAES-OAEP-params : {
      hashAlgorithm              sha1,
      maskGenAlgorithm           mgf1SHA1,
      pSourceAlgorithm           pSpecifiedEmpty
     }
  }

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

A.2.2 RSAES-PKCS1-v1_5

L'identifiant d'objet rsaEncryption (voir l'Appendice A.1) identifie le schéma de chiffrement RSAES-PKCS1-v1_5. Le champ Paramètres associé à cet OID dans une valeur de type AlgorithmIdentifier devra avoir une valeur de type NUL. C'est le même que dans PKCS n° 1 v1.5.

```
IDENTIFIANT D'OBJET rsaEncryption  ::= { pkcs-1 1 }
```

A.2.3 RSASSA-PSS

L'identifiant d'objet id-RSASSA-PSS identifie le schéma de chiffrement RSASSA-PSS.

```
IDENTIFIANT D'OBJET id-RSASSA-PSS  ::= { pkcs-1 10 }
```

Le champ Paramètres associé à cet OID dans une valeur de type AlgorithmIdentifier devra avoir une valeur de type RSASSA-PSS-params :

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

Les champs de type RSASSA-PSS-params ont la signification suivante :

* hashAlgorithm identifie la fonction de hachage. Il doit être un identifiant d'algorithme avec un OID dans l'ensemble OAEP-PSSDigestAlgorithms (voir l'Appendice A.2.1). La fonction de hachage par défaut est SHA-1.

* maskGenAlgorithm identifie la fonction de génération de gabarit. Il doit être un identifiant d'algorithme avec un OID dans l'ensemble PKCS1MGFAlgorithms (voir l'Appendice A.2.1). La fonction de génération de gabarit par défaut est MGF1 avec SHA-1. Pour MGF1 (et plus généralement, pour les autres fonctions de génération de gabarit fondées sur une fonction de hachage) il est recommandé que la fonction de hachage sous-jacente soit la même que celle identifiée par hashAlgorithm ; voir la Note 2 du paragraphe 9.1 pour d'autres commentaires.

* saltLength est la longueur en octets du sel. Elle doit être un entier. Pour un certain hashAlgorithm, la valeur par défaut de saltLength est la longueur en octets de la valeur du hachage. À la différence des autres champs de type RSASSA-

PSS-params, saltLength n'a pas besoin d'être fixe pour une certaine paire de clés RSA.

* trailerField est le nombre de champs en queue, pour la compatibilité avec le projet IEEE P1363a [27]. Il doit être 1 pour la présente version du document, qui représente le champ de queue avec la valeur hexadécimale 0xbc. Les autres champs de queue (y compris le champ de queue HashID || 0xcc dans IEEE P1363a) ne sont pas pris en charge dans le présent document.

TrailerField ::= ENTIER { trailerFieldBC(1) }

Si les valeurs par défaut des champs hashAlgorithm, maskGenAlgorithm, et trailerField de RSASSA-PSS-params sont utilisés, l'identifiant d'algorithme aura alors la valeur suivante :

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

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

Note. Dans certaines applications, la fonction de hachage sous-jacente à un schéma de signature est identifiée séparément du reste des opérations dans le schéma de signature. Par exemple, dans PKCS n° 7 [45], un identifiant de fonction de hachage est placé avant le message et un identifiant d'algorithme de "chiffrement de résumé" (qui indique le reste des opérations) est porté avec la signature. Afin que PKCS n° 7 prenne en charge le schéma de signature RSASSA-PSS, un identifiant d'objet aurait besoin d'être défini pour les opérations dans RSASSA-PSS après la fonction de hachage (analogue à l'OID RSAEncryption pour le schéma RSASSA-PKCS1-v1_5). La syntaxe de message cryptographique (CMS, *Cryptographic Message Syntaxe*) S/MIME [25] adopte une approche différente. Bien qu'un identifiant de fonction de hachage soit placé avant le message, un identifiant d'algorithme pour le schéma de signature complet peut être porté avec une signature CMS (cela est fait pour les signatures DSA). Suivant cette convention, l'OID id-RSASSA-PSS peut être utilisé pour identifier les signatures RSASSA-PSS dans la CMS. Comme CMS est considéré comme le successeur de PKCS n° 7 et que de nouveaux développements tels que l'ajout de la prise en charge de RSASSA-PSS seront poursuivis par rapport à CMS plutôt qu'à PKCS n° 7, un OID pour le "reste de" RSASSA-PSS n'est pas défini dans cette version de PKCS n° 1.

A.2.4 RSASSA-PKCS1-v1_5

L'identifiant d'objet pour RSASSA-PKCS1-v1_5 devra être un des suivants. Le choix de l'OID dépend du choix de l'algorithme de hachage : MD2, MD5, SHA-1, SHA-256, SHA-384, ou SHA-512. Noter que si on utilise MD2 ou MD5, l'OID est alors juste comme dans PKCS n° 1 v1.5. Pour chaque OID, le champ Paramètres associé à cet OID dans une valeur de type AlgorithmIdentifier devra avoir une valeur de type NUL. L'OID devrait être choisi conformément au tableau suivant :

Algorithme de hachage	**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}

La méthode de codage EMSA-PKCS1-v1_5 inclut une valeur ASN.1 de type DigestInfo, où le type DigestInfo a la syntaxe

```
DigestInfo          ::= SEQUENCE {
  digestAlgorithm   DigestAlgorithm,
  digest            CHAINE D'OCTET
}
```

digestAlgorithm identifie la fonction de hachage et devra être un identifiant d'algorithme avec un OID dans l'ensemble PKCS1-v1-5DigestAlgorithms. Voir à l'Appendice B.1 la liste des fonctions de hachage prises en charge.

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

IDENTIFIANT D'ALGORITHME  PKCS1-v1-5DigestAlgorithms  ::= {
        { OID id-md2        PARAMETERS NUL }|
        { OID id-md5        PARAMETERS NUL }|
        { OID id-sha1       PARAMETERS NUL }|
        { OID id-sha256     PARAMETERS NUL }|
        { OID id-sha384     PARAMETERS NUL }|
        { OID id-sha512     PARAMETERS NUL }
    }
```

Appendice B : Techniques de prise en charge

Cette section donne plusieurs exemples de fonctions sous-jacentes qui prennent en charge les schémas de chiffrement de la Section 7 et les méthodes de codage de la Section 9. Une gamme de techniques est donnée ici pour permettre la compatibilité avec les applications existantes ainsi que la migration vers les nouvelles techniques. Bien que ces techniques de prise en charge soient appropriées pour une mise en œuvre par les applications, la mise en œuvre d'aucune d'entre elles n'est exigée. Il est prévu que des profils pour PKCS n° 1 v2.1 soient développés et qu'ils spécifient les techniques particulières de prise en charge.

Cette section donne aussi les identifiants d'objet pour les techniques de prise en charge.

B.1 Fonctions de hachage

Les fonctions de hachage sont utilisées dans les opérations contenues dans les sections 7 et 9. Les fonctions de hachage sont déterministes, ce qui signifie que le résultat est complètement déterminé par l'entrée. Les fonctions de hachage prennent des chaînes d'octets de longueur variable, et génèrent des chaînes d'octets de longueur fixe.

Les fonctions de hachage utilisées dans les opérations contenues dans les sections 7 et 9 devraient généralement être résistantes aux collisions. Cela signifie qu'il est infaisable de trouver deux entrées distinctes pour la fonction de hachage qui produisent le même résultat. Une fonction de hachage résistante aux collisions a aussi la propriété enviable d'être unidirectionnelle ; cela signifie qu'étant donné un certain résultat, il est infaisable de trouver une entrée dont le hachage soit le résultat spécifié. En plus des exigences, la fonction de hachage devrait donner une fonction de génération de gabarit (Appendice B.2) avec résultat pseudo-aléatoire.

Six fonctions de hachage sont données en exemple pour les méthodes de codage, MD2 [33], MD5 [41], SHA-1 [38], et les algorithmes proposés, SHA-256, SHA-384, et SHA-512 [39] de ce document. Pour le schéma de chiffrement RSAES-OAEP et la méthode de codage EMSA-PSS, seuls SHA-1 et SHA-256/384/512 sont recommandés. Pour la méthode de codage EMSA-PKCS1-v1_5, SHA-1 ou SHA-256/384/512 sont recommandés pour les nouvelles applications. MD2 et MD5 ne sont recommandés que pour la compatibilité avec les applications existantes fondées sur PKCS n°1 v1.5.

Les identifiants d'objet id-md2, id-md5, id-sha1, id-sha256, id-sha384, et id-sha512, identifient respectivement les fonctions de hachage :

```
IDENTIFIANT D'OBJET  id-md2      ::= {
   iso(1) member-body(2) us(840) rsadsi(113549)
   digestAlgorithm(2) 2
}

IDENTIFIANT D'OBJET  id-md5  ::= {
   iso(1) member-body(2) us(840) rsadsi(113549)
   digestAlgorithm(2) 5
}

IDENTIFIANT D'OBJET  id-sha1 ::= {
   iso(1) identified-organization(3) oiw(14) secsig(3)
   algorithms(2) 26
}
```

```
IDENTIFIANT D'OBJET  id-sha256   ::= {
   joint-iso-itu-t(2) country(16) us(840) organization(1)
   gov(101) csor(3) nistalgorithm(4) hashalgs(2) 1
}

IDENTIFIANT D'OBJET  id-sha384  ::= {
   joint-iso-itu-t(2) country(16) us(840) organization(1)
   gov(101) csor(3) nistalgorithm(4) hashalgs(2) 2
}

IDENTIFIANT D'OBJET  id-sha512  ::= {
   joint-iso-itu-t(2) country(16) us(840) organization(1)
   gov(101) csor(3) nistalgorithm(4) hashalgs(2) 3
}
```

Les champs Paramètres associés à id-md2 et id-md5 dans une valeur de type AlgorithmIdentifier doivent avoir une valeur de type NUL.

Les champs Paramètres associés à id-sha1, id-sha256, id-sha384, et id-sha512 devraient être omis, mais s'ils sont présents, ils devront avoir une valeur de type NUL.

Note : La version 1.5 de PKCS n° 1 permettait aussi l'utilisation de MD4 dans les schémas de signature. La cryptanalyse de MD4 a fait des progrès significatifs ces dernières années. Par exemple, Dobbertin [18] a démontré comment trouver des collisions pour MD4 et que les deux premiers tours de MD4 ne sont pas unidirectionnels [20]. À cause de ces résultats et d'autres (par exemple, [8]), MD4 n'est plus recommandé. Il y a eu aussi des avancées dans la cryptanalyse de MD2 et MD5, bien que pas suffisantes pour en demander le retrait des applications existantes. Rogier et Chauvaud [43] ont démontré comment trouver des collisions dans une version modifiée de MD2. Personne n'a démontré comment trouver des collisions pour l'algorithme MD5 complet, bien que des résultats partiels aient été trouvés (par exemple, [9], [19]).

Pour régler ces problèmes, SHA-1, SHA-256, SHA-384, ou SHA-512 sont recommandés pour les nouvelles applications. Aujourd'hui, les meilleures attaques de collision (connues) contre ces fonctions de hachage sont des attaques génériques d'une complexité de 2^(L/2), où L est la longueur en bits du résultat du hachage. Pour les schémas de signature du présent document, une attaque de collision est facilement traduite en une falsification de signature. Donc, la valeur L / 2 devrait être au moins égale au niveau de sécurité en bits désiré du schéma de signature (un niveau de sécurité de B bits signifie que la meilleure attaque a une complexité de 2^B). La même règle d'approximation peut être appliquée à RSAES-OAEP ; il est recommandé que la longueur en bits du germe (qui est égale à la longueur en bits du résultat du hachage) soit deux fois le niveau de sécurité désiré en bits.

B.2 Fonctions de génération de gabarit

Une fonction de génération de gabarit prend une chaîne d'octets de longueur variable et une longueur de sortie désirée comme entrée, et sort une chaîne d'octets de la longueur désirée. Il peut y avoir des restrictions à la longueur des chaînes d'octets de l'entrée et de la sortie, mais de telles limites sont généralement très larges. Les fonctions de génération de gabarit sont déterministes ; la chaîne d'octets de sortie est complètement déterminée par la chaîne d'octets d'entrée. Le résultat d'une fonction de génération de gabarit devrait être pseudo-aléatoire : connaissant une partie du résultat mais pas l'entrée, il devrait être infaisable de prédire une autre partie du résultat. La sécurité prouvable de RSAES- OAEP et RSASSA-PSS repose sur la nature aléatoire du résultat de la fonction de génération de gabarit, qui à son tour s'appuie sur la nature aléatoire du hachage sous-jacent.

Une fonction de génération de gabarit est données ici, MGF1, qui se fonde sur une fonction de hachage. MGF1 coïncide avec la fonction de génération de gabarits définie dans la norme IEEE 1363-2000 [26] et dans le projet ANSI X9.44 [1]. De futures versions du présent document pourraient définir d'autres fonctions de génération de gabarit.

B.2.1 MGF1

MGF1 est une fonction de génération de gabarit fondé sur une fonction de hachage.

MGF1 (mgfSeed, maskLen)

Options :
Hash : fonction de hachage (hLen note la longueur en octets du résultat de la fonction de hachage)

Entrée :
mgfSeed : germe à partir duquel le gabarit est généré, une chaîne d'octets
maskLen : longueur prévue en octets du gabarit, au plus 2^32 hLen

Résultat :
mask : gabarit, une chaîne d'octets de longueur maskLen

Erreur : "gabarit trop long"

Étapes :
1. Si maskLen > 2^32 hLen, sortir "gabarit trop long" et arrêter.
2. Soit T la chaîne d'octets vide.
3. Pour le compteur de 0 à \plafond (maskLen / hLen) - 1, faire ce qui suit :
   a. Convertir compteur en une chaîne d'octets C de longueur 4 octets (voir au paragraphe 4.1) :
      C = I2OSP (compteur, 4) .
   b. Enchaîner le hachage du germe mgfSeed et C à la chaîne d'octets T :
      T = T || Hash(mgfSeed || C).
4. Sortir les maskLen octets de tête de T comme gabarit de chaîne d'octets.

L'identifiant d'objet id-mgf1 identifie la fonction de génération de gabarit MGF1 :
IDENTIFIANT D'OBJET id-mgf1 ::= { pkcs-1 8 }

Le champ Paramètres associé à cet OID dans une valeur de type AlgorithmIdentifier devra avoir une valeur de type hashAlgorithm, qui identifie la fonction de hachage sur laquelle se fonde MGF1.

Appendice C : Module ASN.1

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

-- $ Revision: 2.1r1 $

-- La conformité de ce module à l'ASN.1 standard a été vérifiée avec l'outil OSS ASN.1.

ÉTIQUETTES EXPLICITES DE DÉFINITIONS ::=

DÉBUT

-- EXPORTS ALL
-- Tous les types et valeurs définis dans ce module sont exportés pour utilisation dans les autres modules ASN.1.

IMPORTS

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

-- Identifiants d'objets de base

-- Le codage DER en hexadécimal en est : (0x)06 08 2A 86 48 86 F7 0D 01 01
--
IDENTIFIANT D'OBJET pkcs-1 ::= { iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1) 1
}

--
-- Lorsque rsaEncryption est utilisé dans un AlgorithmIdentifier, les paramètres DOIVENT être présents et DOIVENT être
NUL.
--
```

```
IDENTIFIANT D'OBJET rsaEncryption  ::= { pkcs-1 1 }

--
-- Lorsque id-RSAES-OAEP est utilisé dans un AlgorithmIdentifier, les paramètres DOIVENT être présents et DOIVENT
être RSAES-OAEP-params.
--
IDENTIFIANT D'OBJET id-RSAES-OAEP  ::= { pkcs-1 7 }

--
-- Lorsque id-pSpecified est utilisé dans un AlgorithmIdentifier, les paramètres DOIVENT être une CHAINE D'OCTETS.
--
IDENTIFIANT D'OBJET id-pSpecified  ::= { pkcs-1 9 }

-- Lorsque id-RSASSA-PSS est utilisé dans un AlgorithmIdentifier, les paramètres DOIVENT être présents et DOIVENT
être RSASSA-PSS-params.
--
IDENTIFIANT D'OBJET id-RSASSA-PSS  ::= { pkcs-1 10 }

--
-- Lorsque les OID suivants sont utilisés dans un AlgorithmIdentifier, les paramètres DOIVENT être présents et DOIVENT
être NUL.
--
IDENTIFIANT D'OBJET md2WithRSAEncryption   ::= { pkcs-1 2 }
IDENTIFIANT D'OBJET md5WithRSAEncryption   ::= { pkcs-1 4 }
IDENTIFIANT D'OBJET sha1WithRSAEncryption   ::= { pkcs-1 5 }
IDENTIFIANT D'OBJET sha256WithRSAEncryption   ::= { pkcs-1 11 }
IDENTIFIANT D'OBJET sha384WithRSAEncryption  ::= { pkcs-1 12 }
IDENTIFIANT D'OBJET sha512WithRSAEncryption   ::= { pkcs-1 13 }

--
-- Cet OID appartient réellement à un module avec les OID secsig.
--
IDENTIFIANT D'OBJET id-sha1  ::= {
   iso(1) identified-organization(3) oiw(14) secsig(3) algorithms(2) 26
}

--
-- Les OID pour MD2 et MD5, permis seulement dans EMSA-PKCS1-v1_5.
--

IDENTIFIANT D'OBJET id-md2 ::= {
   iso(1) member-body(2) us(840) rsadsi(113549) digestAlgorithm(2) 2
}

IDENTIFIANT D'OBJET id-md5 ::= {
   iso(1) member-body(2) us(840) rsadsi(113549) digestAlgorithm(2) 5
}

--
-- Lorsque id-mgf1 est utilisé dans un AlgorithmIdentifier, les paramètres DOIVENT être présents et DOIVENT être un
HashAlgorithm, par exemple sha1.
--
IDENTIFIANT D'OBJET id-mgf1 ::= { pkcs-1 8 }

-- Types utiles

IDENTIFIANT D'ALGORITHME ::= CLASSE {
  IDENTIFIANT D'OBJET          &id       UNIQUE,
   &Type          FACULTATIF
}
AVEC SYNTAXE { OID &id [PARAMETRES &Type] }

--
```

```
-- Note : Le paramètre InfoObjectSet dans les définitions suivantes permet un objet d'information distinct établi pour être
-- spécifié pour des ensembles d'algorithmes tels que :
-- IDENTIFIANT D'ALGORITHME DigestAlgorithms  ::= {
--     { OID id-md2        PARAMETERS NUL }|
--     { OID id-md5        PARAMETERS NUL }|
--     { OID id-sha1       PARAMETERS NUL }
-- }
--

AlgorithmIdentifier { IDENTIFIANT D'ALGORITHME :InfoObjectSet } ::= SEQUENCE {
   algorithme      IDENTIFIANT D'ALGORITHME .&id({InfoObjectSet}),
   paramètres      IDENTIFIANT D'ALGORITHME .&Type({InfoObjectSet}{@.algorithm})    FACULTATIF
}

-- Algorithmes

-- Algorithmes de résumé EME-OAEP et EMSA-PSS permis.
--
IDENTIFIANT D'ALGORITHME OAEP-PSSDigestAlgorithms ::= {
   { OID id-sha1          PARAMETERS NUL }|
   { OID id-sha256        PARAMETERS NUL }|
   { OID id-sha384        PARAMETERS NUL }|
   { OID id-sha512        PARAMETERS NUL },
   ...  -- Allows for future expansion --
}

--
-- Algorithmes de résumé EMSA-PKCS1-v1_5 digest permis.
--
IDENTIFIANT D'ALGORITHME PKCS1-v1-5DigestAlgorithms  ::= {
   { OID id-md2           PARAMETERS NUL }|
   { OID id-md5           PARAMETERS NUL }|
   { OID id-sha1          PARAMETERS NUL }|
   { OID id-sha256        PARAMETERS NUL }|
   { OID id-sha384        PARAMETERS NUL }|
   { OID id-sha512        PARAMETERS NUL }
}

-- Lorsque id-md2 et id-md5 sont utilisés dans un AlgorithmIdentifier, les paramètres DOIVENT être présents et
DOIVENT être NUL.

-- Lorsque id-sha1, id-sha256, id-sha384 et id-sha512 sont utilisés dans un AlgorithmIdentifier, les paramètres (qui sont
facultatifs) DEVRAIENT être omis. Cependant, une mise en œuvre DOIT aussi accepter les valeurs de AlgorithmIdentifier
où les paramètres sont NUL.

sha1    HashAlgorithm ::= {
   algorithme      id-sha1,
   paramètres      SHA1Parameters : NUL  -- inclus pou la compatibilité avec les mises en œuvre existantes
}

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

SHA1Parameters ::= NUL

--
-- Algorithmes permis de fonction de génération de gabarit. Si l'identifiant est id-mgf1, les paramètres sont un
HashAlgorithm.
--
IDENTIFIANT D'ALGORITHME PKCS1MGFAlgorithms  ::= {
   { OID id-mgf1 PARAMETERS HashAlgorithm },
   ...  – Permet une future expansion --
}
```

```
--
-- AlgorithmIdentifier par défaut pour id-RSAES-OAEP.maskGenAlgorithm et id-RSASSA-PSS.maskGenAlgorithm.
--
mgf1SHA1    MaskGenAlgorithm ::= {
   algorithme      id-mgf1,
   paramètres      HashAlgorithm : sha1
}

MaskGenAlgorithm ::= AlgorithmIdentifier { {PKCS1MGFAlgorithms} }

--
-- Algorithmes permis pour pSourceAlgorithm.
--
IDENTIFIANT D'ALGORITHME PKCS1PSourceAlgorithms  ::= {
   { OID id-pSpecified PARAMETERS EncodingParameters },
   ...  – Permet une future expansion --
}

EncodingParameters ::= CHAINE D'OCTETS(TAILLE(0..MAX))

--
-- Cet identifiant signifie que l'étiquette L est une chaîne vide, de sorte que le résumé de la chaîne vide apparaît dans le bloc
RSA avant la mise au gabarit.
--
pSpecifiedEmpty    PSourceAlgorithm ::= {
   algorithme      id-pSpecified,
   paramètres      EncodingParameters : emptyString
}

PSourceAlgorithm ::= AlgorithmIdentifier { {PKCS1PSourceAlgorithms} }

emptyString    EncodingParameters ::= "H

--
-- Définitions d'identifiant de type pour les OID PKCS n° 1.
--
IDENTIFIANT D'ALGORITHME PKCS1Algorithms  ::= {
   { OID rsaEncryption                  PARAMETERS NUL } |
   { OID md2WithRSAEncryption           PARAMETERS NUL } |
   { OID md5WithRSAEncryption           PARAMETERS NUL } |
   { OID sha1WithRSAEncryption          PARAMETERS NUL } |
   { OID sha256WithRSAEncryption        PARAMETERS NUL } |
   { OID sha384WithRSAEncryption        PARAMETERS NUL } |
   { OID sha512WithRSAEncryption        PARAMETERS NUL } |
   { OID id-RSAES-OAEP                  PARAMETERS RSAES-OAEP-params } | PKCS1PSourceAlgorithms  |
   { OID id-RSASSA-PSS                  PARAMETERS RSASSA-PSS-params } ,
   ...  – Permet une future expansion --
}

--   Structures principales

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

--
-- Représentation de clé privée RSA avec informations sur l'algorithme CRT.
--
RSAPrivateKey ::= SEQUENCE {
   version             Version,
   modulus             ENTIER,  -- n
   publicExponent      ENTIER,  -- e
```

```
   privateExponent     ENTIER,  -- d
   prime1              ENTIER,  -- p
   prime2              ENTIER,  -- q
   exponent1           ENTIER,  -- d mod (p-1)
   exponent2           ENTIER,  -- d mod (q-1)
   coefficient         ENTIER,  -- (inverse de q) mod p
   otherPrimeInfos     OtherPrimeInfos FACULTATIF
}

Version ::= ENTIER { deux-premiers(0), multi(1) }
   (CONTRAINT PAR {
      -- la version doit être multi si otherPrimeInfos est présent --
   })

OtherPrimeInfos ::= SEQUENCE TAILLE(1..MAX) DE OtherPrimeInfo

OtherPrimeInfo ::= SEQUENCE {
   prime               ENTIER,  -- ri
   exposant            ENTIER,  -- di
   coefficient         ENTIER   -- ti
}

--
-- AlgorithmIdentifier.parameters pour id-RSAES-OAEP.
-- Noter que les étiquettes dans cette séquence sont explicites.
--
RSAES-OAEP-params ::= SEQUENCE {
   hashAlgorithm       [0] HashAlgorithm       DEFAUT sha1,
   maskGenAlgorithm    [1] MaskGenAlgorithm    DEFAUT mgf1SHA1,
   pSourceAlgorithm    [2] PsourceAlgorithm    DEFAUT pSpecifiedEmpty
}

--
-- Identifiant pour l'identifiant d'algorithme RSAES-OAEP par défaut.
-- Le codage en DER de ceci en hexadécimal est : (0x)30 0D 06 09 2A 86 48 86 F7 0D 01 01 07 30 00
-- Remarquer que le codage en DER des valeurs par défaut est "vide".
--

rSAES-OAEP-Default-Identifier RSAES-AlgorithmIdentifier ::= {
   algorithme                  id-RSAES-OAEP,
   paramètres                  RSAES-OAEP-params : {
      hashAlgorithm            sha1,
      maskGenAlgorithm         mgf1SHA1,
      pSourceAlgorithm         pSpecifiedEmpty
   }
}

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

--
-- AlgorithmIdentifier.parameters pour id-RSASSA-PSS.
-- Noter que les étiquettes dans cette séquence sont explicites.
--
RSASSA-PSS-params ::= SEQUENCE {
   hashAlgorithm       [0] HashAlgorithm       DEFAUT sha1,
   maskGenAlgorithm    [1] MaskGenAlgorithm    DEFAUT mgf1SHA1,
   saltLength          [2] ENTIER              DEFAUT 20,
   trailerField        [3] TrailerField        DEFAUT trailerFieldBC
}

TrailerField ::= ENTIER { trailerFieldBC(1) }

--
```

```
-- Identifiant pour l'identifiant d'algorithme RSASSA-PSS par défaut.
-- Le codage DER en hexadécimal de ceci est : (0x)30 0D 06 09 2A 86 48 86 F7 0D 01 01 0A 30 00
-- Remarquer que le codage DER des valeurs par défaut est "vide".

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

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

--
-- Syntaxe pour l'identifiant de hachage EMSA-PKCS1-v1_5.
--
DigestInfo ::= SEQUENCE {
   digestAlgorithm     DigestAlgorithm,
   digest              CHAINE D'OCTETS
}

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

FIN  -- PKCS1Definitions
```

Appendice D : Considérations de propriété intellectuelle

Le système de chiffrement à clé publique de RSA est décrit dans le brevet U.S. 4,405,829, qui arrivait à expiration le 26 septembre 2000. RSA Security Inc. ne fait pas d'autre revendication de brevet sur les constructions décrites dans le présent document, bien que des techniques sous-jacents spécifiques puissent être couvertes.

RSA à nombres premiers multiples est décrit dans le brevet U.S. 5,848,159. L'Université de Californie a indiqué qu'elle a un brevet déposé sur le schéma de signature PSS [5]. Elle a aussi produit une lettre au groupe de travail P1363 de l'IEEE déclarant que si le schéma de signature PSS est inclus dans une norme IEEE, "l'Université de Californie accordera, lorsque cette norme aura été adoptée, LIBREMENT licence à toute mise en œuvre conforme à PSS comme technique de réalisation d'une signature numérique avec appendice" [23]. Le schéma de signature PSS est spécifié dans le projet IEEE P1363a [27], qui était en cours de vote lorsque le présent document a été publié.

Licence de copier le présent document est accordée pourvu qu'il soit identifié comme "Norme de cryptographie à clé publique (PKCS, *Public-Key Cryptography Standards*) de RSA Security Inc." dans tous les matériaux qui mentionnent ou font référence à ce document.

RSA Security Inc. ne fait aucune autre déclaration à l'égard de revendications de propriété intellectuelle par d'autres parties. Une telle détermination est de la responsabilité de l'utilisateur.

Appendice E : Historique des révisions

Versions 1.0 - 1.3
Les versions 1.0 - 1.3 ont été distribuées aux participants aux réunions des normes de cryptographie à clé publique de RSA Data Security, en février et mars 1991.

Version 1.4
La version 1.4 faisait partie de la livraison publique initiale du 3 juin 1991 de PKCS. La version 1.4 a été publiée comme document SEC-SIG-91-18 de l'atelier de travail de mise en œuvre NIST/OSI.

Version 1.5
La version 1.5 incorporait plusieurs changements rédactionnels, incluant des mises à jour des références et l'ajout d'un historique des révisions. Les changements de substances suivants ont été faits :
- Section 10 : les processus de signature et vérification "MD4 avec RSA" ont été ajoutés.
- Section 11 : l'identifiant d'objet md4WithRSAEncryption a été ajouté.

La version 1.5 a été republiée comme RFC 2313 par l'IETF.

Version 2.0
La version 2.0 incorporait des changement rédactionnels majeurs en termes de structure du document et introduisait le schéma de chiffrement RSAES-OAEP. Cette version continuait de prendre en charge les processus de chiffrement et de signature de la version 1.5, bien que l'algorithme MD4 de hachage ne soit plus admis du fait des avancées de la cryptanalyse dans cette période. La version 2.0 a été republiée comme RFC 2437 [35] par l'IETF.

Version 2.1
La version 2.1 introduit le RSA à nombres premiers multiples et le schéma de signature RSASSA-PSS avec un appendice ainsi que plusieurs améliorations rédactionnelles. Cette version continue de prendre en charge les schémas de la version 2.0.

Appendice F : Références

[1 ANSI groupe de travail X9F1, projet ANSI X9.44 D2, "Key Establishment Using Integer Factorization Cryptography". Document de travail, mars 2002.

[2 M. Bellare, A. Desai, D. Pointcheval,P. Rogaway, "Relations Among Notions of Security for Public-Key Encryption Schemes". Dans H. Krawczyk, éditeur, Advances in Cryptology - Crypto '98, vol. 1462 des Lecture Notes in Computer Science, pp. 26 - 45. Springer Verlag, 1998.

[3] M. Bellare et P. Rogaway. "Optimal Asymmetric Encryption - How to Encrypt with RSA". Dans A. De Santis, éditeur, Advances in Cryptology - Eurocrypt '94, volume 950 des Lecture Notes in Computer Science, pp. 92 - 111. Springer Verlag, 1995.

[4] M. Bellare et P. Rogaway. "The Exact Security of Digital Signatures - How to Sign with RSA et Rabin". Dans U. Maurer, éditeur, Advances in Cryptology - Eurocrypt '96, vol. 1070 des Lecture Notes in Computer Science, pp. 399 - 416. Springer Verlag, 1996.

[5] M. Bellare et P. Rogaway. "PSS: Provably Secure Encoding Method for Digital Signatures". Suumission au groupe de travail IEEE P1363, août 1998. Disponible à http://grouper.ieee.org/groups/1363/ .

[6] D. Bleichenbacher. "Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1". Dans H. Krawczyk, éditeur, Advances in Cryptology - Crypto '98, vol. 1462 des Lecture Notes in Computer Science, pp. 1 - 12. Springer Verlag, 1998.

[7] D. Bleichenbacher, B. Kaliski et J. Staddon. "Recent Results on PKCS #1: RSA Encryption Standard". Bulletin n° 7 des Laboratoires RSA, juin 1998.

[8] B. den Boer et A. Bosselaers. "An Attack on the Last Two Rounds of MD4". Dans J. Feigenbaum, éditeur, Advances in Cryptology - Crypto '91, vol. 576 des Lecture Notes in Computer Science, pp. 194 - 203. Springer Verlag, 1992.

[9] B. den Boer et A. Bosselaers. "Collisions for the Compression Function of MD5". Dans T. Helleseth, éditeur, Advances in Cryptology - Eurocrypt '93, vol. 765 des Lecture Notes in Computer Science, pp. 293 - 304. Springer Verlag, 1994.

[10] D. Coppersmith, M. Franklin, J. Patarin et M. Reiter. "Low-Exponent RSA with Related Messages". Dans U. Maurer, éditeur, Advances in Cryptology - Eurocrypt '96, vol. 1070 des Lecture Notes in Computer Science, pp. 1 - 9. Springer Verlag, 1996.

[11] D. Coppersmith, S. Halevi et C. Jutla. "ISO 9796-1 et the New Forgery Strategy". Présenté à la session finale de Crypto '99, août 1999.

[12] J.-S. Coron. "On the Exact Security of Full Domain Hashing". Dans M. Bellare, éditeur, Advances in Cryptology -

Crypto 2000, vol. 1880 des Lecture Notes in Computer Science, pp. 229 - 235. Springer Verlag, 2000.

[13] J.-S. Coron. "Optimal Security Proofs for PSS et Other Signature Schemes". Dans L. Knudsen, éditeur, Advances in Cryptology - Eurocrypt 2002, vol. 2332 des Lecture Notes in Computer Science, pp. 272 - 287. Springer Verlag, 2002.

[14] J.-S. Coron, M. Joye, D. Naccache et P. Paillier. "New Attacks on PKCS #1 v1.5 Encryption". Dans B. Preneel, éditeur, Advances in Cryptology - Eurocrypt 2000, vol. 1807 des Lecture Notes in Computer Science, pp. 369 - 379. Springer Verlag, 2000.

[15] J.-S. Coron, D. Naccache et J. P. Stern. "On the Security of RSA Padding". Dans M. Wiener, éditeur, Advances in Cryptology - Crypto '99, vol. 1666 des Lecture Notes in Computer Science, pp. 1 - 18. Springer Verlag, 1999.

[16] Y. Desmedt et A.M. Odlyzko. "A Chosen Text Attack on the RSA Cryptosystem et Some Discrete Logarithm Schemes". Dans H.C. Williams, éditeur, Advances in Cryptology - Crypto '85, vol. 218 des Lecture Notes in Computer Science, pp. 516 - 522. Springer Verlag, 1986.

[17] T. Dierks et C. Allen, "Protocole TLS version 1.0", RFC2246, janvier 1999.

[18] H. Dobbertin. "Cryptanalysis of MD4". Dans D. Gollmann, éditeur, Fast Software Encryption '96, vol. 1039 des Lecture Notes in Computer Science, pp. 55 - 72. Springer Verlag, 1996.

[19] H. Dobbertin. "Cryptanalysis of MD5 Compress". Présenté à la rump session de Eurocrypt '96, mai 1996.

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

[21] E. Fujisaki, T. Okamoto, D. Pointcheval et J. Stern. "RSA-OAEP is Secure under the RSA Assumption". Dans J. Kilian, éditeur, Advances in Cryptology - Crypto 2001, vol. 2139 des 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, juin 1959.

[23] M.L. Grell. "Re: Encoding Methods PSS/PSS-R". Lettre au groupe de travail IEEE P1363, University of California, 15 juin 1999. Disponible à http://grouper.ieee.org/groups/1363/P1363/patents.html .

[24] J. Haastad. "Solving Simultaneous Modular Equations of Low Degree". SIAM Journal of Computing, volume 17, pp. 336 - 341, 1988.

[25] R. Housley, "Algorithmes de syntaxe de message cryptographique (CMS)", RFC3370, août 2002. *(P.S.)*

[26] IEEE Std 1363-2000. "Standard Specifications for Public Key Cryptography". IEEE, août 2000.

[27] Groupe de travail IEEE P1363. "IEEE P1363a D11: Draft Standard Specifications for Public Key Cryptography -- Amendment 1: Additional Techniques". 16 décembre 2002. Disponible à http://grouper.ieee.org/groups/1363/ .

[28] ISO/CEI 9594-8:1997 : "Technologies de l'information - Interconnexion des systèmes ouverts – L'annuaire : cadre d'authentification". 1997.

[29] ISO/CEI FDIS 9796-2: "Technologies de l'information - Techniques de sécurité – Schémas de signature numérique donnant la récupération du message – Partie 2 : Mécanismes fondés sur la factorisation d'entiers". Projet final de norme internationale, décembre 2001.

[30] ISO/CEI 18033-2 : "Technologies de l'information - Techniques de sécurité - Algorithmes de chiffrement - Partie 2 : Chiffrements asymétriques". V. Shoup, éditeur, Texte pour le 2nd projet de travail, janvier 2002.

[31] J. Jonsson. "Security Proof for the RSA-PSS Signature Scheme (extended abstract)". Second atelier NESSIE ouvert, septembre 2001. Version complète disponible à http://eprint.iacr.org/2001/053/ .

[32] J. Jonsson & B. Kaliski. "On the Security of RSA Encryption in TLS". Dans M. Yung, éditeur, Advances in Cryptology - Crypto 2002, vol. 2442 des Lecture Notes in Computer Science, pp. 127 - 142. Springer Verlag, 2002.

[33] B. Kaliski, "Algorithme de résumé de message MD2", RFC1319, avril 1992. *(Historique, Information)*

[34] B. Kaliski. "On Fonction de hachage Identification in Signature Schemes". Dans B. Preneel, éditeur, RSA Conference 2002, Cryptographers' Track, vol. 2271 des Lecture Notes in Computer Science, pp. 1 - 16. Springer Verlag, 2002.

[35] B. Kaliski et J. Staddon, "PKCS n° 1 : Spécifications de la cryptographie RSA version 2.0", RFC2437, octobre 1998. *(Obsolète, voir le présent document) (Information)*

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

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

[38] National Institute of Standards et Technology (NIST). FIPS Publication 180-1: "Secure Hash Standard". avril 1994.

[39] National Institute of Standards et Technology (NIST). Draft FIPS 180-2: "Secure Hash Standard". Projet, mai 2001. Disponible à http://www.nist.gov/sha/ .

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

[41] R. Rivest, "Algorithme de résumé de message MD5", RFC1321, avril 1992. *(Information)*

[42] R. Rivest, A. Shamir et L. Adleman. "A Method for Obtaining Digital Signatures et Public-Key Cryptosystems". Communications of the ACM, 21 (2), pp. 120-126, février 1978.

[43] N. Rogier et P. Chauvaud. "The Compression Function of MD2 is not Collision Free". Présenté à Selected Areas of Cryptography '95. Carleton University, Ottawa, Canada. mai 1995.

[44] RSA Laboratories. "PKCS #1 v2.0: RSA Encryption Standard". octobre 1998.

[45] RSA Laboratories. "PKCS #7 v1.5: Cryptographic Message Syntax Standard". novembre 1993. (Republié comme RFC2315.)

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

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

[48] V. Shoup. "OAEP Reconsidered". Dans J. Kilian, éditeur, Advances in Cryptology - Crypto 2001, volume 2139 des Lecture Notes in Computer Science, pp. 239 - 259. Springer Verlag, 2001.

[49] R. D. Silverman. "A Cost-Based Security Analysis of Symmetric et Asymmetric Key Lengths". RSA Laboratories Bulletin n°. 13, avril 2000. Disponible à http://www.rsasecurity.com.rsalabs/bulletins/ .

[50] G. J. Simmons. "Subliminal communication is easy using the DSA". Dans T. Helleseth, éditeur, Advances in Cryptology - Eurocrypt '93, volume 765 des Lecture Notes in Computer Science, pp. 218-232. Springer-Verlag, 1993.

Appendice G : sur PKCS

Les normes de cryptographie à clé publique sont des spécifications produites par les RSA Laboratories en coopération avec des développeurs de systèmes sûrs du monde entier pour accélérer le déploiement de la cryptographie à clés publiques. D'abord publiés en 1991 suite à des réunions d'un petit groupe d'adeptes précoces de la technologie de la clé publique, les documents PKCS sont devenus largement référencés et mis en œuvre. Les contributions de la série PKCS ont été incorporées dans de nombreuses normes formelles et de fait, incluant les documents ANSI X9 et IEEE P1363, PKIX, SET, S/MIME, SSL/TLS, et WAP/WTLS.

De nouveaux développements de PKCS surviennent au travers des discussions de la liste de diffusion et par des ateliers occasionnels, et les suggestions d'amélioration sont les bienvenues. Pour plus d'informations, contacter :

PKCS Editor
RSA Laboratories
174 Middlesex Turnpike
Bedford, MA 01730 USA
pkcs-editor@rsasecurity.com
http://www.rsasecurity.com/rsalabs/pkcs

Appendice H : Corrections faites durant le processus de publication de la RFC

Les corrections suivantes ont été faites lors de la conversion du document PKCS n° 1 v2.1 en cette RFC :

* L'exigence que les paramètres dans une valeur de AlgorithmIdentifier pour id-sha1, id-sha256, id-sha384, et id-sha512 soient NULL a été changée en une recommandation que les paramètres soient omis (tout en permettant que les paramètres soient NULL). Ceci est pour s'aligner avec les definitions promulguées à l'origine pae le NIST. Les mises en œuvre DOIVENT accepter les valeurs de AlgorithmIdentifier aussi bien sans paramètres qu'avec les paramètres NULL.

* Les notes après RSADP et RSASP1 (paragraphes 5.1.2 et 5.2.1) ont été corrigés pour se référer à l'étape 2.b plutôt que 2.a.

* Les réferences [25], [27] et [32] ont été mises à jour pour citer les nouvelles publications.

Ces corrections seront reflétées dans les futures éditions de PKCS #1 v2.1.

Considérations pour la sécurité

Les questions de sécurité forment l'objet même du présent mémoire.

Remerciements

Le présent document se fonde sur une contribution des laboratoires RSA, le centre de recherche de RSA Security Inc. Toute utilisation substantielle du texte du présent document doit mentionner RSA Security Inc. RSA Security Inc. demande que tous les matériels qui mentionnent ou font référence au présent document l'identifient comme "PKCS n° 1 v2.1 de RSA Security Inc.".

Adresse des auteurs

Jakob Jonsson
Philipps-Universitaet Marburg
Fachbereich Mathematik und Informatik
Hans Meerwein Strasse, Lahnberge
DE-35032 Marburg
téléphone : +49 6421 28 25672
mél : jonsson@mathematik.uni-marburg.de

Burt Kaliski
RSA Laboratories
174 Middlesex Turnpike
Bedford, MA 01730
USA
téléphone : +1 781 515 7073
mél : bkaliski@rsasecurity.com

Déclaration complète de droits de reproduction

processus de normes pour Internet doivent être suivies, ou si nécessaire pour le traduire dans des langues autres que l'anglais.

Les permissions limitées accordées ci-dessus sont perpétuelles et ne seront pas révoquées par la Société Internet ou ses successeurs ou ayants droit.

Ce document et les renseignements qu'il contient sont fournis "TELS QUELS" et l'INTERNET SOCIETY et l'INTERNET ENGINEERING TASK FORCE déclinent toute garantie, expresse ou implicite, y compris mais sans s'y limiter, toute garantie que l'utilisation de l'information ici présente n'enfreindra aucun droit ou aucune garantie implicite de commercialisation ou d'adaptation a un objet particulier.

Remerciement
Le financement de la fonction d’édition des RFC est actuellement fourni par la Internet Society.