En général, il convient que deux chiffrements consécutifs d'un même message clair avec la même clef produisent deux chiffrés distincts. Ainsi, un intrus ne peut pas observer que ces deux messages chiffrés correspondent au même message clair. En chiffrement symétrique, cette garantie est apportée par exemple par l'emploi d'un mode opératoire utilisant un vecteur d'initialisation. L'introduction d'une part aléatoire est aussi fortement recommandée en cryptographie asymétrique, en particulier avec le chiffrement RSA.

Par ailleurs, un risque propre au chiffrement asymétrique apparait si ce chiffrement est utilisé pour l'envoi de messages très courts car un ennemi pourrait chiffrer avec la clef publique tous les messages courts potentiels et être en mesure de déchiffrer les messages chiffrés sans même connaître la clef privée.

C'est pourquoi RSA nécessite en général de faire appel à une technique de bourrage. Deux techniques principales existent : le PKCS#1, qui est plus simple mais qui n'est plus recommandé pour les nouvelles applications, et l'« Optimal asymmetric encryption padding » (OAEP) qui se décline sous plusieurs formes selon la taille des clefs utilisées et la fonction de hâchage choisie.

Enfin le chiffrement RSA nécessite de coder le message, vu comme une suite d'octets, sous la forme d'un entier représentatif. Le procédé de chiffrement d'une suite d'octets consiste à construire un entier associé à M, à calculer l'entier $C = M^e \pmod n$ et à renvoyer un tableau d'octets correspondant à C. Il nécessite donc à la fois de coder et de décoder une suite d'octets sous la forme d'un entier.

Exercice E.1 Entier long représentatif d'un tableau d'octets La première étape du chiffrement d'un tableau d'octets par RSA consiste à le représenter par un entier positif. Si le tableau X comporte l octets, alors l'entier x correspondant sera naturellement codé sur $8 \times l$ bits, au plus, puisque chaque octet correspond à 8 bits. Plus précisément, selon la RFC 2437, la règle d'encodage des suites d'octets sous forme d'entiers est la suivante :

```
OS2IP converts an octet string to a nonnegative integer.
Input: X, octet string to be converted
Output: x, corresponding nonnegative integer
Steps:
1. Let X_1 X_2 ... X_l  be the l octets of X from first to last, and
   let x_{l-i} have value X_i for 1<= i <= l.
2. Let x = x_{l-1} * 256^{l-1} + x_{l-2} * 256^{l-2} +...+ x_1 * 256 + x_0.
3. Output x.
```

Notez que l'on peut reformuler la spécification de la fonction **`os2ip()`** ainsi : chaque suite d'octets $X[0], ..., X[l-1]$ est codée par l'entier

$$x = X[0] \times 256^{l-1} + X[1] \times 256^{l-2} + \ldots + X[l-2] \times 256 + X[l-1]$$

en considérant naturellement chaque octet $X[k]$ comme un entier compris entre 0 et 255. Autrement dit, l'écriture de x en base 256 est donnée par le tableau X. Observez ici que deux messages qui ne diffèrent que par le nombre d'octets nuls en début de tableau conduiront au même entier représentatif.

Question 1. Que vaut x lorsque le tableau X correspond aux caractères ASCII de la chaîne « KYOTO » ?

Vous pourrez effectuer ce calcul en Java, en C ou même « à la main » en observant que le codage ASCII des caractères de « KYOTO » peut être obtenu à l'aide de la commande suivante :

```
$ echo -n "KYOTO" | od -tu1
0000000    75  89  79  84  79
```

Question 2. Inversement, comment écrit-on le nombre décimal 323 620 918 351 en base 256 ?

Sans surprise, comme pour écrire un nombre décimal en base 2, vous pourrez effectuer une série de divisions euclidiennes successives pour obtenir un-à-un les chiffres de cette écriture.

Question 3. En tirant profit du fait que $16^2 = 256$, en déduisez-en une écriture du nombre 323 620 918 351 en hexadécimal.

Question 4. Qu'affiche la commande suivante ?

```
$ echo -n "KYOTO" | od -tx1
```

Exercice E.2 Bourrage historique : PKCS#1 Les programmes `RSA_PKCS1.java` et `rsa_PKCS1.c` de l'archive `RSA.zip` utilisent une paire de clefs RSA dont le module n qui s'écrit sur 1024 bits. Ces programmes sont indiqués sur les figures 34 et 35. Ils chiffrent en RSA à l'aide de cette paire de clefs un tableau **m** qui est codé sous la forme d'un entier représentatif x, conformément à la RFC 2437, à l'aide de fonctions appropriées.

Comme souligné plus haut deux tableaux $\mathbf{m}_1$ et $\mathbf{m}_2$ qui ne différent que par un ou plusieurs octets nuls en début de tableau conduiront au même nombre représentatif. Ces octets nuls initiaux sont susceptibles d'être omis lors du déchiffrement, sauf si l'on connaît *a priori* la taille exact du tableau. L'objectif de cet exercice est d'insérer un procédé de *bourrage* avant l'encodage sous la forme d'un entier représentatif x sur lequel est appliqué la formule de chiffrement classique bien connue : $c = x^e \pmod n$.

Le bourrage PKCS#1 prend en entrée le message **m** formé de k octets et produit un tableau d'octets **em** formé par la concaténation de l'octet **0x00**, de l'octet **0x02**, d'une suite d'octets *non nuls* PS, éventuellement vide, de l'octet **0x00** et des octets du message **m** :

$$\mathbf{em} = \mathbf{0x00} \| \mathbf{0x02} \| PS \| \mathbf{0x00} \| \mathbf{m}$$

Lorsque le module de la clef RSA fait 1024 bits, soit 128 octets, le tableau **em** doit comporter 128 octets. Par conséquent le message **m** ne peut dépasser 125 octets. D'autre part, puisque le premier octet est égal à l'octet nul **0x00**, l'entier représenté par le tableau **em** aura bien une valeur strictement inférieure à la valeur du module n, puisqu'il s'écrit en réalité sur 127 octets, au plus. Dans cet exercice, le tableau d'octets **m** est constitué des cinq octets suivants : **0x4B**, **0x59**, **0x4F**, **0x54** et **0x4F**.

Question 1. Modifiez l'un de ces programmes afin de construire un tableau **em** de 128 octets à partir du tableau **m** selon un bourrage PKCS#1 ; veillez ensuite à ce que l'entier x représente **em** et non **m**. Il vous faudra ici engendrer la suite aléatoire PS formée d'octets *non nuls*.

Question 2. Vérifiez que chaque exécution du programme produit un chiffré différent.

Question 3. Ajoutez ensuite à ce programme le déchiffrement du chiffré ainsi que la suppression du bourrage. Il faudra donc ici élaborer un moyen de calculer le tableau **m** à partir du tableau **em**.

Question 4. Vérifiez que le résultat du déchiffrement correspond exactement au message **m** initial.

Question 5. Le tableau d'octet formant le chiffré correspond au nombre $c = x^e \pmod n$, qui est évidemment plus petit que n ; il s'écrit donc sur 1024 bits, au plus, en binaire. Assurez-vous que le chiffré soit toujours formé exactement de 128 octets.

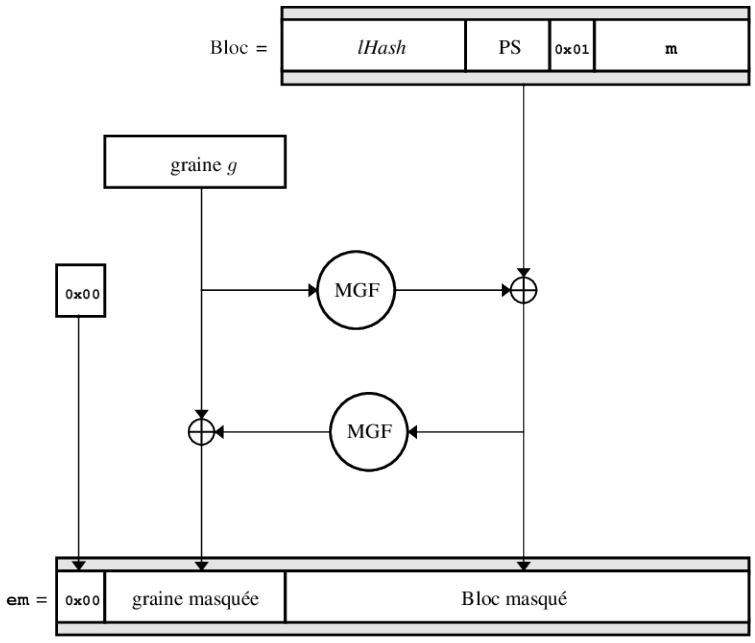

FIGURE 33 – Schéma du bourrage OAEP

Exercice E.3 Bourrage OAEP avec SHA-1 et des clefs de 1024 bits Défini dans le document « RSA Cryptography Standard » produit par la société RSA Security®, le bourrage OAEP doit être préféré au bourrage PKCS#1 dont les faiblesses sont connues. Ce procédé est décrit sur la figure 33.

Dans cet exercice, il s'agit de modifier le programme précédent afin de mettre en oeuvre le bourrage OAEP en lieu et place du bourrage PKCS#1. Nous conservons donc ici le tableau `m` donné et la paire de clef RSA fournie. De plus, le tableau `em` contiendra à nouveau 128 octets (c'est-à-dire 1024 bits). Outre le premier octet égal à `0x00`, le tableau `em` comportera 127 octets formés par les 20 octets de la *graine masquée* et les 107 octets du *bloc masqué* (cf. fig. 33). Ce dernier est calculé à partir du bloc clair qui a également de longueur 107 octets et qui est constitué constitués par 20 octets constants (*lHash*), un certain nombre [5] d'octets *nuls* (PS), un octet `0x01` et enfin le tableau `m` initial : ce dernier est de taille variable ; mais il ne devra jamais comporter plus de 86 octets.

Question 1. La première partie du bourrage consiste à construire le *bloc* à partir du tableau `m`. Le bloc est un tableau d'octets formé par la constante `0xDA39A3EE5E6B4B0D3255BFEF95601890AFD80709` (qui est le résumé SHA-1 de la chaîne vide), notée *lHash*, suivie d'une suite d'octets nuls, notée PS et éventuellement vide, suivie de l'octet `0x01`, et terminé par les octets du tableau `m`. Par exemple, si `m` est formé des six octets `0x416C66726564` alors le tableau d'octets bloc vaudra

`0xDA39A3EE5E6B4B0D3255BFEF95601890AFD8070900...0001416C66726564`

Ajoutez au programme la fabrique et l'affichage du bloc à partir du tableau `m` donné.

La commande « `echo -n "" | sha1sum` » produira la constante *lHash* utilisée, si vous en avez besoin.

Question 2. Le bourrage OAEP est non-déterministe, c'est-à-dire que le contenu du tableau d'octets `em` dépend d'une valeur aléatoire g choisie au moment du chiffrement ; cette *graine* g est un tableau de 20 octets. La fonction MGF(g, k) permet de produire un *masque* de k octets à partir de la graine g. Ce tableau d'octets MGF(g, k) est obtenu en accolant, les uns derrière les autres, les résumés SHA-1 des tableaux de 24 octets $g\|$`0x00000000`, $g\|$`0x00000001`, $g\|$`0x00000002`, etc. Par exemple, si la graine g est formée de 20 octets nuls, alors $g\|$`0x00000000` est formé de 24 octets nuls et son résumé SHA-1 vaut :

SHA-1($g\|$`0x00000000`)=`0xD3399B7262FB56CB9ED053D68DB9291C410839C4`

D'autre part,

SHA-1($g\|$`0x00000001`)=`0xA4FA647DEFA221E1F720C856FD893973D239D227`

Par conséquent, le tableau de 107 octets MGF$(g, 107)$ commencera par les octets

`0xD3399B7262FB56CB9ED053D68DB9291C410839C4A4FA647DEFA221E1F7...`

Noter ici que 107 n'étant pas un multiple de 20, le dernier résumé SHA-1 utilisé lors de la fabrication du tableau d'octets MGF$(g, 107)$ sera *tronqué*. Ajoutez au programme le calcul et l'affichage des 107 octets du masque MGF$(g, 107)$, où g est la graine formée de 20 octets nuls.

Le code en C et en Java pour le calcul de résumés SHA-1 est donné dans l'archive `RSA.zip`.

Question 3. Le *bloc masqué* est simplement le XOR bit-à-bit des 107 octets du bloc calculé à la première question et des 107 octets du masque calculé à la question précédente. Ajoutez au programme le calcul et l'affichage des 107 octets du bloc masqué.

Question 4. Le *masque de la graine* est obtenu par la formule MGF(bloc masqué,20) ; puisque les résumés SHA-1 font 20 octets précisément, le masque de la graine est simplement le résumé SHA-1 du tableau obtenu à la question précédente auquel il faut accoler 4 octets nuls. Comme indiqué sur la figure 33, la graine masquée est simplement le XOR bit-à-bit de la graine g choisie et du masque de la graine MGF(bloc masqué,20). Ajoutez au programme le calcul et l'affichage des 20 octets de la graine masquée.

Question 5. Conservez dans un fichier binaire les 128 octets du chiffré obtenu lors d'une exécution : vous pourrez vérifier sa validité lors d'un prochain TP à l'aide de la JCE.

5. La longueur de la suite d'octets nuls PS est variable et éventuellement nulle.

```java
import java.math.BigInteger;
public class RSA_PKCS1 {
    public static void main(String[] args) throws Exception {
        //---------------------------------------------------------------
        //  Construction et affichage de la clef
        //---------------------------------------------------------------
        BigInteger n = new BigInteger("00af7958cb96d7af4c2e6448089362"+
                       "31cc56e011f340c730b582a7704e559e3d797c2b697c4eec07ca5a903983"+
                       "4c0566064d11121f1586829ef6900d003ef414487ec492af7a12c34332e5"+
                       "20fa7a0d79bf4566266bcf77c2e0072a491dbafa7f93175aa9edbf3a7442"+
                       "f83a75d78da5422baa4921e2e0df1c50d6ab2ae44140af2b", 16);
        BigInteger e = BigInteger.valueOf(0x10001);
        BigInteger d = new BigInteger("35c854adf9eadbc0d6cb47c4d11f9c"+
                       "b1cbc2dbdd99f2337cbeb2015b1124f224a5294d289babfe6b483cc253fa"+
                       "de00ba57aeaec6363bc7175fed20fefd4ca4565e0f185ca684bb72c12746"+
                       "96079cded2e006d577cad2458a50150c18a32f343051e8023b8cedd49598"+
                       "73abef69574dc9049a18821e606b0d0d611894eb434a59", 16);
        System.out.println("Module          (n): " + n + " ("+n.bitLength()+" bits)");
        System.out.println("Exposant public (e): " + e + " ("+e.bitLength()+" bits)");
        System.out.println("Exposant privé  (d): " + d + " ("+d.bitLength()+" bits)");
        //---------------------------------------------------------------
        //  Construction et affichage du message clair
        //---------------------------------------------------------------
        byte[] m = { 0x4B, 0x59, 0x4F, 0x54, 0x4F } ;
        System.out.println("Message clair      : " + toHex(m) );
        //---------------------------------------------------------------
        //  Du message m à l'entier représentatif x (partie à modifier)
        //---------------------------------------------------------------
        BigInteger x = new BigInteger(1, m);          // Encodage du message
        System.out.println("x = " +  x  + " (en décimal)");
        System.out.println("x = 0x" + String.format("%X", x) + " (en hexadécimal)");
        //---------------------------------------------------------------
        //  Chiffrement de l'entier représentatif
        //---------------------------------------------------------------
        BigInteger c = x.modPow(e, n);
        System.out.println("x^e mod n = " + c + " ("+c.bitLength()+" bits)");
        //---------------------------------------------------------------
        //  Décodage de l'entier représentatif
        //---------------------------------------------------------------
        byte[] chiffré = c.toByteArray();
        System.out.println("Message chiffré    : " + toHex(chiffré) );
    }
    public static String toHex(byte[] données) {
        StringBuffer sb = new StringBuffer();
        for(byte k: données) sb.append(String.format("0x%02X ", k));
        sb.append(" (" + données.length + " octets)");
        return sb.toString();
    }
}
/*
  $ make
  javac *.java
  $ java RSA_PKCS1
  Module          (n): 12322204109610601400...299   (1024 bits)
  Exposant public (e): 65537 (17 bits)
  Exposant privé  (d): 37767385438721355925...209   (1022 bits)
  Message clair      : 0x4B 0x59 0x4F 0x54 0x4F     (5 octets)
  x = 323620918351 (en décimal)
  x = 0x4B594F544F (en hexadécimal)
  x^e mod n = 65891982980551359715048403549...638   (1023 bits)
  Message chiffré    : 0x5D 0xD5 0x53 0x0B ... 0x26 (128 octets)
*/
```

FIGURE 34 – Programme `RSA_PKCS1.java` à compléter

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "gmp.h"
typedef unsigned char uchar;

int main(void) {
  //----------------------------------------------------------------
  //  Construction et affichage de la clef
  //----------------------------------------------------------------
  mpz_t e, n, d;
  mpz_inits(n, e, d, (void *)NULL);
  mpz_set_str(n,
              "00af7958cb96d7af4c2e644808936231cc56e011f340c730b582a7704e55\
              9e3d797c2b697c4eec07ca5a9039834c0566064d11121f1586829ef6900d\
              003ef414487ec492af7a12c34332e520fa7a0d79bf4566266bcf77c2e007\
              2a491dbafa7f93175aa9edbf3a7442f83a75d78da5422baa4921e2e0df1c\
              50d6ab2ae44140af2b", 16);
  mpz_set_str(e, "10001", 16);
  mpz_set_str(d,
              "35c854adf9eadbc0d6cb47c4d11f9cb1cbc2dbdd99f2337cbeb2015b1124\
              f224a5294d289babfe6b483cc253fade00ba57aeaec6363bc7175fed20fe\
              fd4ca4565e0f185ca684bb72c1274696079cded2e006d577cad2458a5015\
              0c18a32f343051e8023b8cedd4959873abef69574dc9049a18821e606b0d\
              0d611894eb434a59", 16);
  gmp_printf("Module          (n): %Zd (%d bits)\n", n, mpz_sizeinbase(n, 2));
  gmp_printf("Exposant public (e): %Zd (%d bits)\n", e, mpz_sizeinbase(e, 2));
  gmp_printf("Exposant privé  (d): %Zd (%d bits)\n", d, mpz_sizeinbase(d, 2));
  //----------------------------------------------------------------
  //  Construction et affichage du message clair
  //----------------------------------------------------------------
  uchar m[5] = { 0x4B, 0x59, 0x4F, 0x54, 0x4F } ;
  printf("Message clair      : ");
  for (int i = 0 ; i<sizeof(m) ; i++) printf("0x%02X ", m[i]);
  printf(" (%lu octets)\n", sizeof(m));
  //----------------------------------------------------------------
  //  Du message m à l'entier représentatif x (partie à modifier)
  //----------------------------------------------------------------
  mpz_t x;
  mpz_init(x);
  mpz_import (x, sizeof(m), 1, sizeof(m[0]), 0, 0, m);
  gmp_printf("x = %Zd (en décimal)\n", x);
  gmp_printf("x = 0x%ZX (en hexadécimal)\n", x);
  //----------------------------------------------------------------
  //  Chiffrement de l'entier représentatif
  //----------------------------------------------------------------
  mpz_t c;
  mpz_init(c);
  mpz_powm(c, x, e, n);
  gmp_printf("m^e mod n = %Zd (%d bits)\n", c, mpz_sizeinbase(c, 2));
  //----------------------------------------------------------------
  //  Décodage de l'entier représentatif
  //----------------------------------------------------------------
  uchar chiffre[256];
  size_t taille;
  mpz_export(chiffre, &taille, 1, 1, 1, 0, c);
  printf("Message chiffré    : ");
  for (size_t i = 0; i < taille; i++) printf("0x%02X ", chiffre[i]);
  printf(" (%lu octets)\n", taille);

  mpz_clears (x, c, n, e, d, (void *) NULL);
  exit(EXIT_SUCCESS);
}
```

FIGURE 35 – Programme `rsa_pkcs1.c` à compléter

```c
#include <stdio.h>
#include <stdlib.h>
#include <openssl/sha.h>

int main() {
  unsigned char resume_sha1[SHA_DIGEST_LENGTH];
  unsigned char message[] = "Alain Turin";
  printf("Message à hacher: \"%s\" (%lu octets)\n", message , sizeof(message)-1);

  SHA1(message, sizeof(message)-1, resume_sha1);

  printf("Le résumé SHA1 de cette chaîne est: 0x");
  for(int i = 0; i < SHA_DIGEST_LENGTH; i++) printf("%02X", resume_sha1[i]);
  printf("\n");
  exit(EXIT_SUCCESS);
}
/*
  $ make
  gcc -o resume_sha1 -I/usr/local/include -I/usr/include resume_sha1.c -L/usr/local/lib
  -L/usr/lib -lm -lssl -lcrypto -g -Wall -std=c99
  $ ./resume_sha1
  Message à hacher: "Alain Turin" (11 octets)
  Le résumé SHA1 de cette chaîne est: 0x9B682F2CA6F44CB60493288A686DE5D81ECA6B6D
  $ echo -n "Alain Turin" | sha1sum
  9b682f2ca6f44cb60493288a686de5d81eca6b6d  -
  $
*/
```

FIGURE 36 – Calcul du résumé SHA1 d'une chaîne de caractères en C

```java
  import java.io.*;
  import java.security.*;

  public class ResumeSHA1 {
    public static void main(String[] args) throws Exception {
      byte[] resumeSHA1;
      String message = "Alain Turin";
      System.out.println("Message à hacher: \"" + message +"\"");

      MessageDigest hâcheur = MessageDigest.getInstance("SHA1");
      resumeSHA1 = hâcheur.digest(message.getBytes());

      System.out.print("Le résumé SHA1 de cette chaîne est: 0x");
      for(byte octet: resumeSHA1) System.out.printf("%02X", octet);
      System.out.println();
    }
  }

  /*
    $ javac ResumeSHA1.java
    $ java ResumeSHA1
    Message à hacher: "Alain Turin"
    Le résumé SHA1 de cette chaîne est: 0x9B682F2CA6F44CB60493288A686DE5D81ECA6B6D
  */
```

FIGURE 37 – Calcul du résumé SHA1 d'une chaîne de caractères en Java