Série modulaire sur la

Sécurité informatique

Fascicule 1:

Introduction à la cryptographie

Frédéric Mailhot

Université de Sherbrooke

Avril 2021

Note : En vue d'alléger le texte, le masculin est utilisé pour désigner les femmes et les hommes.

Document fascicule01.pdf
Version 3.14, Avril 2021

Département de génie électrique et de génie informatique, Université de Sherbrooke.
Réalisé à l'aide de LaTeX et TeXstudio.

Table des matières

Liste des figures

Liste des tableaux

Liste des algorithmes

Chapitre

1

Systèmes de cryptographie à clés symétriques

Les systèmes d'encryptage à clés symétriques existent depuis très longtemps. Leur principe général de fonctionnement est d'effectuer une transformation sur un texte[1], transformation pour laquelle il existe une inverse. On désire évidemment que le texte transformé soit incompréhensible à moins de détenir l'information nécessaire à la transformation inverse. Quoique de très nombreuses méthodes aient été proposées depuis les premières tentatives d'encryptage il y a plus de vingt siècles, peu de ces méthodes sont vraiment sécuritaires. Par exemple, toute transformation qui associe une lettre transformée unique à chacune des lettres de l'alphabet utilisé est susceptible de se faire découvrir par une analyse de la fréquence des caractères. Ainsi, la lettre "e" est celle qui apparaît la plus fréquemment dans les textes écrits en anglais ou en français. En nous basant sur la fréquence des lettres apparaissant dans le message crypté, nous pouvons déduire plusieurs des lettres les plus utilisées, et déchiffrer en partie le message transformé [Sin00, Spi05].

L'histoire moderne de la cryptographie à clés symétriques commence dans les années 1940 avec l'analyse mathématique par Claude Shannon d'une technique de cryptage proposée à la fin de la première guerre mondiale par Gilbert Vernam. Dans ce qui suit, nous parlerons donc brièvement de la méthode de Vernam et de l'analyse de Shannon. Nous verrons ensuite deux familles de chiffres à clé symétrique:

[1]Ici, on parle de texte au sens général. En effet, un texte est une série de mots composés de lettres appartenant à un alphabet prédéfini. L'alphabet peut être composé de tout ensemble fini de symboles. Il est toujours possible d'établir une relation entre ces symboles et un sous-ensemble des nombres entiers positifs, ceci impliquant qu'un mot (ou suite de symboles) peut être représenté par une suite de nombres. Par exemple, les symboles {e,i,l} peuvent être représentés respectivement par les nombres binaires {00, 01, 11}. Les mots {ile, le, elle, il} peuvent être représentés par {011100, 1100, 00111100, 0111}, qu'on pourrait ensuite utiliser pour échanger des messages basés sur un dictionnaire comprenant 4 mots.

1. Le chiffrement par flux, où une clé d'encryptage aussi longue que le message est utilisée et les lettres du message sont cryptées une à la fois

2. Le chiffrement par bloc, où la clé d'encryptage est de longueur finie et le texte est crypté par blocs de même longueur que la clé d'encryptage

1.1 Chiffre de Vernam et analyse de Shannon

Gilbert Vernam, un ingénieur qui travaillait à l'époque pour la compagnie ATT, a proposé en 1917 une méthode d'encryptage de texte (envoyé par télétype) basée sur l'utilisation de l'opération OU-exclusif entre le texte à crypter et une série de nombres pré-définis et enregistrés sur des rubans de papier[2]. Le ruban de papier était utilisé en boucle circulaire, à la fois pour l'encryptage et le décryptage. Il faut noter qu'à l'époque, les lettres étaient encodées à l'aide du code de Baudot, un système utilisant 5 chiffres binaires, précurseur du code ASCII moderne. Le système a été adapté par la suite par Joseph Mauborgne, un major général de l'armée américaine, pour devenir le système de clés symétriques à utilisations uniques [Ver26]. Dans ce cas, la clé est formée de nombres aléatoires et sa longueur est identique à la longueur du message à envoyer. Les nombres aléatoires utilisés doivent être connus à la fois de l'envoyeur et du receveur du message, ce qui implique un échange sécurisé préalable entre les deux interlocuteurs. À l'époque, cet échange était typiquement fait par un messager apportant physiquement une copie des clés aux deux parties.

Claude Shannon, lui aussi employé de la compagnie ATT, est reconnu comme le père de la théorie de l'information. En parallèle avec son travail sur l'information [Sha48], il a démontré dans les années 1940 que l'utilisation d'une clé à usage unique et aussi longue que le message garantit la sécurité d'un message [Sha49]. En étudiant les probabilités des messages en clair (non cryptés), des clés et des messages cryptés, Shannon a prouvé que l'utilisation unique d'une séquence aléatoire non répétée de nombres (telle que proposée par Vernam et Mauborgne) est parfaitement inattaquable [Sha49, TW06, Sti06, Spi05]. Cette méthode est malheureusement peu pratique, puisqu'elle requiert l'envoi sécurisé préalable de clés aussi longues que le message à transmettre, une nouvelle clé étant transmise pour chaque nouveau message. Dans des systèmes réels, on prend des clés plus courtes, qui sont utilisées de façon répétitive pour couvrir le message. Le message ainsi crypté n'est plus parfaitement sécuritaire, mais si la clé utilisée est assez longue, il est impossible de percer le système en pratique. Ainsi, l'utilisation d'une clé de 128 bits implique l'existence de 2^{128} clés potentielles. Ce nombre étant excessivement grand, nul ne peut énumérer toutes les clés possibles pour ensuite décrypter le message sécurisé. Il faut malgré tout prendre des précautions lors de l'encryptage symétrique, car certains types d'analyse

[2]Il a obtenu le brevet américain 1 310 719 le 22 juillet 1919 pour cette invention [USP06]

sont possibles même avec des clés de cette taille. Nous aborderons le sujet dans la section 1.3.6.

Shannon a étudié les conditions nécessaires à l'obtention d'un niveau parfait de secret. En se basant sur le calcul des probabilités, il a défini qu'un message chiffré est parfaitement secret lorsqu'il ne dévoile aucune information à propos du message original. Il a établi en particulier que le niveau d'incertitude face à un message est lié au nombre de messages possibles, et qu'un message chiffré atteint un niveau parfait de secret lorsqu'il n'affecte pas le niveau d'incertitude face au message original. En d'autres termes, ceci veut dire qu'intercepter un message chiffré parfaitement secret n'aide aucunement l'intercepteur à découvrir le message original [Sha49, TW06, Sti06]. Shannon a aussi proposé de combiner plusieurs systèmes de chiffrement pour obtenir plus de sécurité. En particulier, Shannon a établi que deux types d'opérations combinées pouvaient permettre d'atteindre un niveau parfait de secret: la diffusion et la confusion. Le principe de la diffusion est qu'un petit changement dans le message en clair devrait produire un grand changement dans le message crypté. En d'autres termes, un petit changement doit "diffuser" dans tout le message chiffré. En ce qui concerne la confusion, il s'agit de mélanger les lettres de telle sorte qu'on ne puisse les identifier (par exemple par leur fréquence) dans le message chiffré.

1.2 Chiffrement par flux

Le chiffrement par flux (*stream cipher* en anglais) utilise une clé très longue, théoriquement de même longueur que le message [Spi05, Sta03]. Le principe de ce type de chiffrement est le suivant: chaque lettre du message original est codé sous forme de séquence de 0 et de 1 (par exemple, à l'aide de code ASCII, ou de Unicode). Le chiffrement est obtenu en effectuant l'opération OU-exclusif (EXOR) entre les bits du messages et les bits correspondants de la clé. La figure 1.1 représente le chiffrement d'un message de 16 bits à l'aide d'une clé de 16 bits.

L'opération OU-exclusif entre le message et la clé est rapide et facile à effectuer. L'élément important du chiffrement par flux est donc la clé d'encryptage. Elle doit être assez longue, et doit être connue à la fois de l'envoyeur et du receveur du message. Il existe de nombreuses méthodes pour générer des séquences de nombres pseudo-aléatoires qui peuvent être utilisées à cet effet. En supposant que deux interlocuteurs partagent le nombre de départ d'une séquence très longue de nombres pseudo-aléatoires, il peuvent ensuite utiliser le générateur de nombres pour obtenir la clé d'encryptage par flux.

Il faut noter que le déchiffrement du message chiffré est obtenu en effectuant exactement la même opération que pour l'encryptage, cette fois entre les bits du message chiffré et ceux de la clé.

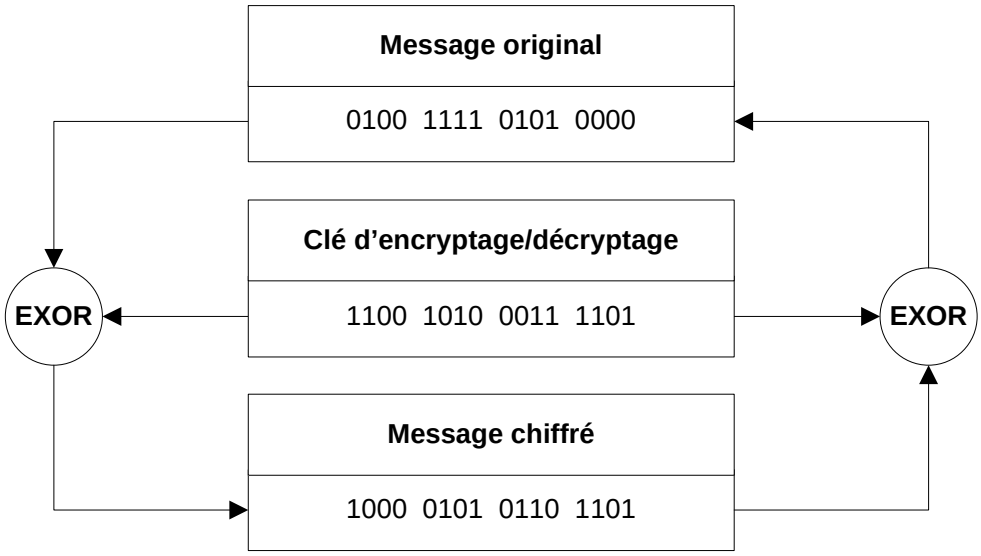


Figure 1.1: Exemple de chiffrement/déchiffrement par flux

1.2.1 Utilisation de LFSR

L'utilisation de registres à décalage à retour linéaire (Linear Feedback Shift Register, LFSR) est l'une des méthodes utilisées pour produire la clé d'encryptage/décryptage dans les systèmes de chiffrement par flux. Le principe de cette méthode est le suivant: on utilise un registre à décalage dont certains des bits sont combinés (à l'aide d'un OU-exclusif) pour produire la prochaine entrée du registre à décalage. Typiquement, un nombre initial est entré en parallèle dans le registre, qui est ensuite utilisé en mode décalage série pour produire une séquence de valeurs en sortie. La figure 1.2 présente un LFSR avec chargement/sortie parallèles et retour linéaire des bits 1 et 4 [Spi05].

L'intérêt des LFSR est qu'ils sont faciles à réaliser à l'aide de portes logiques matérielles. La réalisation logicielle de LFSR est aussi relativement simple et efficace. Un point important à remarquer est qu'après un certain nombre d'itérations, ils reviennent nécessairement à leur point de départ, démarrant un nouveau cycle de nombres apparaissant dans le même ordre que lors de la suite initiale. Pour un LFSR qui comporte n bits, ce retour au nombre de départ se fait au maximum après 2^n itérations. Cependant, la période d'un cycle peut être beaucoup plus courte que ce nombre maximal. Cette période dépend des bits qui sont utilisés comme entrées du OU-exclusif. L'étude de la longueur des cycles en fonction des bits utilisés pour produire le retour dépasse le cadre de ce texte, mais le lecteur intéressé peut consulter [DOF+90, MvOV97].

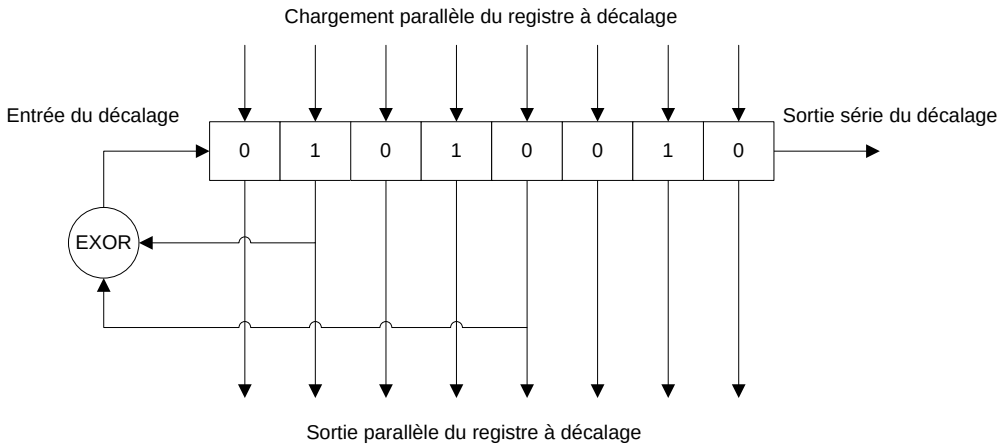


Figure 1.2: Exemple de registre à décalage avec retour linéaire (LFSR)

1.2.2 Chiffre RC4

La méthode de chiffrement par flux RC4 a été développée par Ronald Rivest chez RSA Security en 1987. Cette méthode non publiée est demeurée secrète jusqu'en 1994, alors qu'une soumission anonyme sur le site d'échange *cypherpunks* en a rendu public le code source [Sch96]. Comme le fonctionnement de RC4 n'a jamais été officiellement dévoilé par son propriétaire légitime (RSA Security), sa teneur exacte ainsi que son analyse demeurent sujets à caution. Depuis 1994, plusieurs auteurs ont publié des articles portant sur la méthode RC4 *alléguée* (Alleged RC4) [KMP$^+$98, FM01]. L'intérêt de RC4 est qu'il a été utilisé par plusieurs protocoles courants de sécurisation (SSL, TLS, WEP, WPA).

Tout comme les autres méthodes de chiffrement par flux, RC4 repose sur le calcul d'un OU-exclusif entre le message à transmettre et une série de bits pseudo-aléatoires. L'élément distinctif de RC4 est la méthode utilisée pour générer la séquence pseudo-aléatoire, qui se résume aux étapes décrites par l'algorithme1.1.

```
RC4_init(cle[taille]) {
  for (i = 0; i++; i <= 255) {
    tableau_RC4[i] = i
  }

  for (i = 0; i++; i <= 255) {
    j = (i + tableau_RC4[i] + cle[i modulo taille]) modulo 256
    tmp = tableau_RC4[i]
    tableau_RC4[i] = tableau_RC4[j]
    tableau_RC4[j] = tmp
  }

  return(tableau_RC4)
}
```

Algorithme 1.1: Méthode de chiffrement par flux RC4 - Initialisation

```
RC4(tableau_RC4[256],message[taille]) {
  i = 0
  j = 0
  for (k = 0; k< taille; k++) {
    i = (i + 1) mod 256
    j = (j + tableau_RC4[i]) mod 256
    tmp = tableau_RC4[i]
    tableau_RC4[i] = tableau_RC4[j]
    tableau_RC4[j] = tmp
    nb_alea = (tableau_RC4[i] + tableau_RC4[j]) mod 256
    message_crypt[k] = message[k] OU-Exclusif nb_alea
  }
}
```

Algorithme 1.2: Méthode de chiffrement par flux RC4 - Génération et utilisation de nombres pseudo-aléatoires

1. Débuter avec un tableau de 256 octets (256 nombres compris entre 0 et 255)
2. Initialiser le tableau en écrivant le numéro de l'octet à la position correspondante (0 dans l'octet 0, 1 dans l'octet 1, ... 255 dans l'octet 255)
3. Faire une permutation initiale des octets, de la façon suivante:
   (a) Pour chaque valeur d'un compteur comprise entre 0 et 255:
   (b) Additionner la valeur du compteur,
   (c) *À compléter...*

1.2.3 Chiffre A5

À compléter...

1.3 Chiffrement par bloc

Le chiffrement par bloc (*block cipher* en anglais) est une extension du chiffrement par flux. Alors que le chiffrement par flux crypte les bits l'un à la fois à l'aide d'une opération ou-exclusif, le chiffrement par bloc crypte des groupes de bits d'une taille prédéterminée (où la taille est plus grande que 1). Le chiffrement d'un bloc (ou groupe de bits) se fait à l'aide d'une transformation réversible (biunivoque) qui associe à chaque groupe de bits de départ un groupe de bits chiffré unique. Habituellement, la taille des blocs est un multiple de 8 assez grand[3], typiquement 64 ou plus. En ce moment, il est courant d'utiliser des blocs d'au moins 128 bits[4].

Il existe de nombreuses méthodes modernes de chiffrement par bloc, l'ancêtre de toutes étant la méthode de Feistel. Dans ce qui suit, nous étudierons d'abord la structure générale des réseaux de substitution/permutation. Puis nous étudierons les principes de base de la méthode de Feistel, pour ensuite couvrir les méthodes DES et 3-DES. Nous enchaînerons avec les méthodes récentes telles AES (Rijndael) et ses concurrentes. Nous terminerons cette section par une analyse des modes d'utilisation des chiffrements par blocs.

1.3.1 Réseaux de substitution/permutation

Les réseaux de substitution/permutation sont en quelque sorte la réalisation des idées de Shannon sur la combinaison de méthodes de chiffrement pour obtenir la confusion et la diffusion. Les systèmes de chiffrement par blocs sont dérivés de ce modèle: on effectue une série d'itérations, chaque itération comprenant une étape de substitution (phénomène de confusion) et une étape de permutation (phénomène de diffusion). De plus, on intègre à chaque itération une opération ou-exclusif, en utilisant une clé spécifique à cette étape. Enfin, dans certains systèmes de chiffrement par blocs, on ajoute aussi une transformation linéaire réversible comme étape supplémentaire de chaque itération.

Nous étudierons maintenant un système simple qui démontre comment on peut agencer les différentes étapes pour effectuer le chiffrement [Sti06]. Dans cet exemple, on chiffrera des messages de 16 bits. Il est important de noter qu'un tel système ne serait pas sécuritaire (les blocs sont beaucoup trop petits), mais

[3] Les ordinateurs modernes utilisent l'octet de 8 bits comme élément de base du processeur, de sa mémoire et de ses disques

[4] AES permet l'utilisation de blocs de 128, 160, 192 et 256 bits.

qu'il permet de comprendre les principes derrière les systèmes de chiffrement par blocs réels. Cet exemple est présenté sous forme schématique à la figure 1.3. Les éléments importants de ce système sont les boîtes S_{xy} de substitution, les boîtes P de permutation et les boîtes EXOR qui effectuent les opérations ou-exclusif. Il n'y a pas de transformation linéaire. Les opérations ou-exclusifs se font sur 16 bits, avec des clés K_i différentes dans chaque cas.

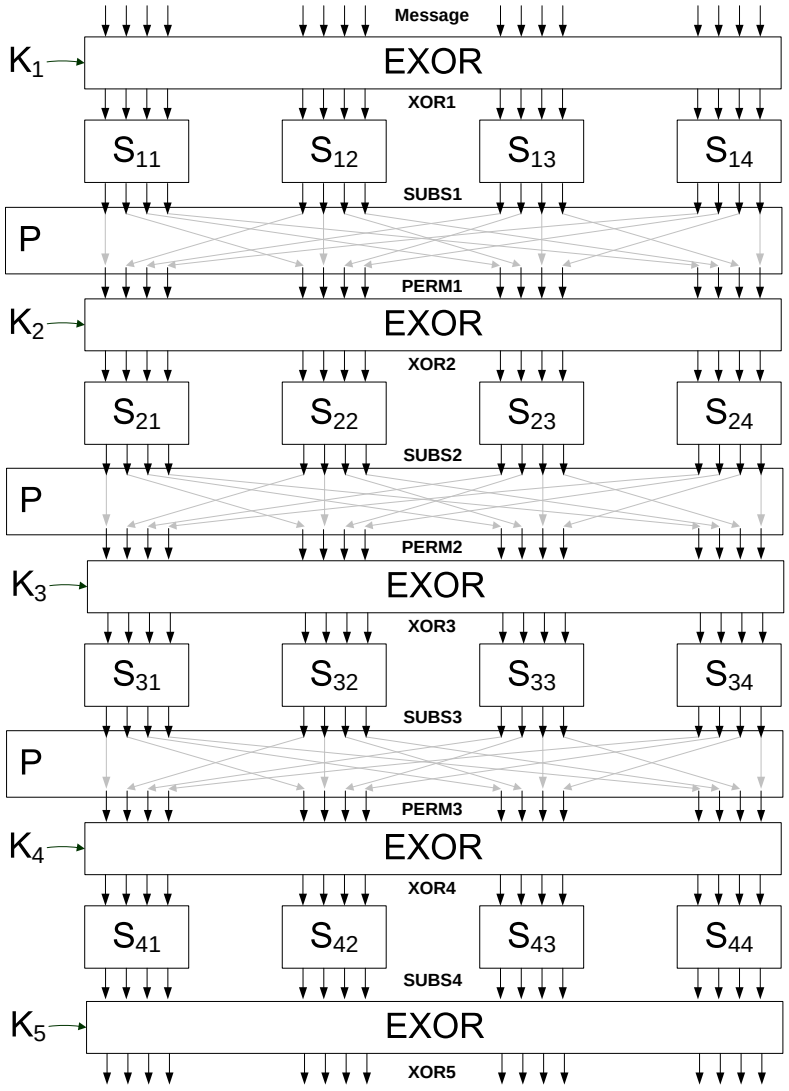


Figure 1.3: Schéma d'un réseau de Substitution-Permutation

Les boîtes de substitution font correspondre à chaque entrée possible une sortie unique. Ici, il y a $2^4 = 16$ entrées possibles pour une boîte de substitution donnée, puisque les entrées sont sur 4 bits. Le message à chiffrer étant sur 16 bits, il est donc nécessaire d'utiliser 4 boîtes de substitution pour chaque itération.

Dans un cas général, chacune des boîtes S d'une itération peut être différente. On peut aussi changer les boîtes à chaque itération. Pour cet exemple, nous utiliserons une unique boîte de substitution S ($S_{11} = S_{12} = \cdots = S_{44} = S$). Les valeurs définissant la boite de substitution S apparaissent à la figure 1.4.

	S															
Entrée	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Sortie	**1110**	**0100**	**1101**	**0001**	**0010**	**1111**	**1011**	**1000**	**0011**	**1010**	**0110**	**1100**	**0101**	**1001**	**0000**	**0111**

Figure 1.4: Boîte de substitution pour l'exemple de la figure 1.3

Les boîtes de permutations redistribuent les bits d'entrées vers les sorties. Ce sont en quelque sorte des stations d'aiguillage. Puisque le but de cette opération est de faire "diffuser" les bits du message, le nombre d'entrées des boîtes de permutation est égal à la taille des blocs traités. Dans cet exemple, les blocs ont 16 bits, alors la boîte de permutation doit faire correspondre 16 sorties à 16 entrées. Puisque cette opération doit être réversible, il doit y avoir une correspondance unique entre chaque entrée et chaque sortie. Dans l'exemple, nous utilisons la boîte de permutation P de la figure 1.5.

	P															
Entrée	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
Sortie	**16**	**12**	**8**	**4**	**15**	**11**	**7**	**3**	**14**	**10**	**6**	**2**	**13**	**9**	**5**	**1**

Figure 1.5: Boîte de permutation pour l'exemple de la figure 1.3

Afin de compléter le système de chiffrement de la figure 1.3, nous devons établir la taille de la clé de chiffrement, et la méthode à utiliser pour déterminer les clés spécifiques à chaque itération. Pour que le système soit simple, nous utiliserons une clé de 32 bits. Pour chaque itération, nous utiliserons les 16 bits les plus faibles de la clé globale. De plus, après chaque itération, nous décalerons la clé globale de 4 bits vers le bas. Ainsi, les portions suivantes de la clé de 32 bits seront utilisées lors des itérations correspondantes:

$$\begin{aligned}
\text{Soit } K &= k_{31}, k_{30}, k_{29}, \cdots, k_0 \\
\text{EXOR 1: } K_1 &= k_{15}, k_{14}, k_{13}, \cdots, k_0 \\
\text{EXOR 2: } K_2 &= k_{19}, k_{18}, k_{17}, \cdots, k_4 \\
\text{EXOR 3: } K_3 &= k_{23}, k_{22}, k_{21}, \cdots, k_8 \\
\text{EXOR 4: } K_4 &= k_{27}, k_{26}, k_{25}, \cdots, k_{12} \\
\text{EXOR 5: } K_5 &= k_{31}, k_{30}, k_{29}, \cdots, k_{16}
\end{aligned}$$

Ayant tous les éléments nécessaires, nous pouvons maintenant suivre le déroulement du chiffrement alors qu'il se propage dans le système. Dans la figure 1.3, les annotations `Message`, `XOR1`, `SUBS1`, `PERM1`, `XOR2`, `SUBS2`, `PERM2`, `XOR3`, `SUBS3`, `PERM3`, `XOR4`, `SUBS4`, `XOR5` correspondent aux différentes étapes identifiées ici. Supposons que le message à chiffrer et la clé de chiffrement sont les suivants:

$$\begin{aligned} \text{Message} &= \texttt{0110 1000 1101 0001} \\ K &= \texttt{0111 1111 0011 0000 1010 1001 0110 0001} \end{aligned}$$

Alors les étapes intermédiaires du chiffrement seront (les bits sont ordonnés de gauche à droite, du plus fort au plus faible):

$$\begin{aligned} \text{Message} &= \texttt{0110 1000 1101 0001} \\ K_1 &= \texttt{1010 1001 0110 0001} \\ \text{XOR1} &= \texttt{1100 0001 1011 0000} \\ \text{SUBS1} &= \texttt{0101 0100 1100 1110} \\ \text{PERM1} &= \texttt{0011 1111 0001 1000} \\ K_2 &= \texttt{0000 1010 1001 0110} \\ \text{XOR2} &= \texttt{0011 0101 1000 1110} \\ \text{SUBS2} &= \texttt{0001 1111 0011 0000} \\ \text{PERM2} &= \texttt{0100 0100 0110 1110} \\ K_3 &= \texttt{0011 0000 1010 1001} \\ \text{XOR3} &= \texttt{0111 0100 1100 0111} \\ \text{SUBS3} &= \texttt{1000 0010 0101 1000} \\ \text{PERM3} &= \texttt{1001 0010 0100 0010} \\ K_4 &= \texttt{1111 0011 0000 1010} \\ \text{XOR4} &= \texttt{0110 0001 0100 1000} \\ \text{SUBS4} &= \texttt{1011 0100 0010 0011} \\ K_5 &= \texttt{0111 1111 0011 0000} \\ \text{XOR5} &= \texttt{1100 1011 0001 0011} \end{aligned}$$

On peut se demander si les réseaux de substitution/permutation sont vraiment utiles. En effet, dans l'exemple qui précède, on pourrait imaginer établir une table de lookup qui aurait la même fonctionnalité que le système proposé tout en étant beaucoup plus performant. Puisqu'on utilise des blocs de 16 bits, on pourrait créer un tableau de 2^{16} entrées, chaque entrée détenant le code de 16 bits de la transformation. Au total, ce tableau exigerait $2^{16} \times \frac{16}{8} = 128$K octets, ce qui est relativement petit de nos jours. Pour ce qui est de notre exemple, l'utilisation d'un tableau de valeurs serait donc beaucoup plus performant que l'ensemble de calculs proposés. Cependant, pour des systèmes réels, où les blocs ont des tailles d'au moins 128 bits, la quantité de mémoire requise serait prohibitive: même pour les plus petits blocs (128 bits), la quantité de mémoire requise serait de $2^{128} \times \frac{128}{8} = 2^{132}$ octets.

Dans l'exemple précédent, la série de clés utilisées (lors des opérations ou-exclusif) est obtenue de manière très simple. Dans un système réel, les clés de calcul obtenues à partir de la clé principale sont générées par des moyens plus sophistiqués.

1.3.2 Méthode de Feistel

Jusqu'au début des années 1970, la cryptographie était utilisée principalement par des organismes gouvernementaux, pour la transmission de messages secrets destinés à des entités distantes. Avec l'avènement des ordinateurs, les débuts de la réseautique [MB76] et leur utilisation de plus en plus répandue, il est apparu souhaitable de définir un standard d'encryptage d'utilisation publique. En 1973, le National Bureau of Standards (NBS) américain a lancé un appel d'offre pour la mise au point d'un tel système. Chez IBM, Horst Feistel et Don Coppersmith firent partie de l'équipe qui mit au point le système LUCIFER en 1974, qui fut modifié par le NBS et devint le DES (Data Encryption System) [Fei73]. Quoique DES soit maintenant dépassé, il demeure important parce que son principe de base, attribué à Feistel, est encore utilisé de nos jours dans plusieurs systèmes d'encryptage modernes [TW06, VGM04].

La méthode de Feistel a été l'une des premières méthodes modernes de cryptographie à mettre en oeuvre les principes généraux énoncés par Shannon. Ce système est basé sur l'utilisation répétitive d'opérations réversibles destinées à produire la diffusion et la confusion. Le principe est le suivant: lors de chacune des itérations, la moitié du message à crypter est modifié, alors que l'autre moitié est utilisée pour former la clé d'encryptage et qu'elle demeure intacte. Lors du décryptage, on peut donc récupérer la clé d'encryptage d'une itération donnée en se servant de la moitié non cryptée du message. Chaque itération échange les moitiés de message, de sorte que l'ensemble du message est crypté à la fin du processus. Sous forme schématique, une itération de la méthode de Feistel apparaît à la figure 1.6.

Nous étudierons maintenant plus en détail la méthode DES, qui est la version officielle de Lucifer, le système de chiffrement mis au point chez IBM par Feistel.

1.3.3 Standards DES et 3-DES

DES est une implémentation particulière d'un système de Feistel, qui utilise une clé de 56 bits pour encrypter des blocs de messages de 64 bits. Les éléments importants du système sont la production de la séquence de clés et les différentes transformations réalisées lors d'une itération du système: l'expansion d'un demi-message, le ou-exclusif avec la clé, la substitution, la permutation et le ou-exclusif avec l'autre demi-message. Nous aborderons brièvement chacun de ces éléments, qui apparaissent aux figures 1.7 et 1.8.

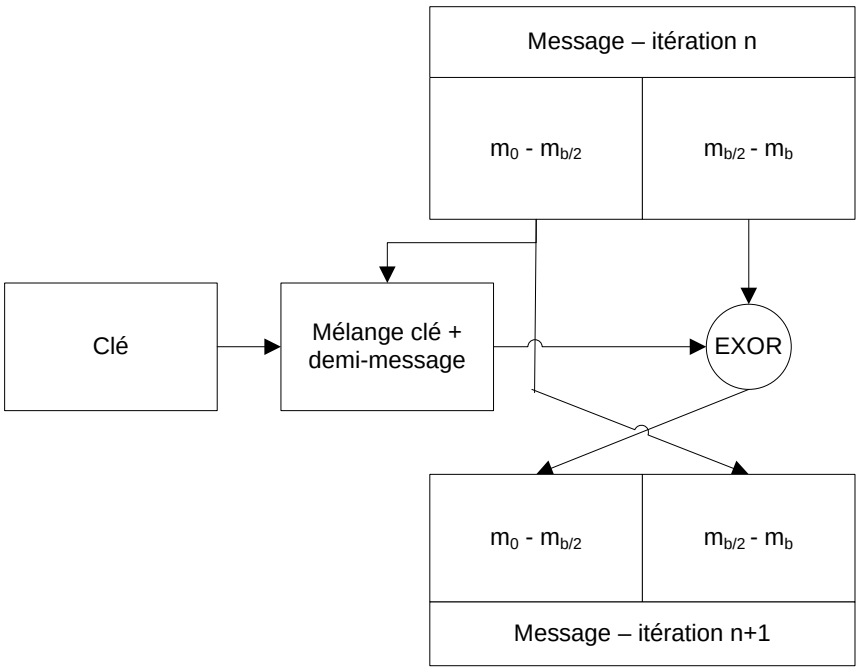


Figure 1.6: Schématique d'une itération de la méthode de Feistel

DES est une séquence de 16 itérations d'un réseau de substitution/permutation. Pour chaque itération, une clé particulière est obtenue à partir de la clé générale. Cette dernière est longue de 64 bits, dont 8 sont utilisés comme bits de parité: une clé DES est formée de 56 bits effectifs, soit 8 blocs de 7 bits. Après chaque bloc de 7 bits, on ajoute un huitième bit de parité, de sorte que chaque octet de la clé générale a une parité impaire (chaque octet contient un nombre impair de 1). Dans la clé générale DES les bits 7, 15, 23, 31, 39, 47, 55 et 63 sont les bits de parité, chacun étant combiné avec les 7 bits de plus faible poids qui le précèdent (on suppose ici que les bits de la clé DES sont numérotés de 0 à 63).

À chacune des 16 itérations, la clé particulière utilisée occupe 48 bits. Pour obtenir chacune des clés d'itération, on fait des transformations successives de la clé originale de 64 bits. La séquence de clés d'itérations commence par l'extraction des 56 bits effectifs de la clé de 64 bits. Puis un premier choix de permutation (Permutation Choice 1) est fait, à l'aide de la boîte PC-1, qui réordonne les 56 bits selon l'ordre indiqué à la figure 1.9. Le résultat de cette permutation est divisé en deux groupes de 28 bits, auxquels on applique un décalage circulaire vers la gauche de 1 bit. Le résultat de ce décalage est utilisé comme entrée du deuxième choix de permutation (Permutation Choice 2), qui produit une clé de 48 bits en sortie. Pour obtenir les clés suivantes, on fait subir un décalage circulaire au résultat du décalage circulaire précédent, et le résultat de cette opération devient à la fois l'entrée du PC-2 de la clé suivante et l'entrée du prochain décalage circulaire (voir figure 1.7). Le bloc de permu-

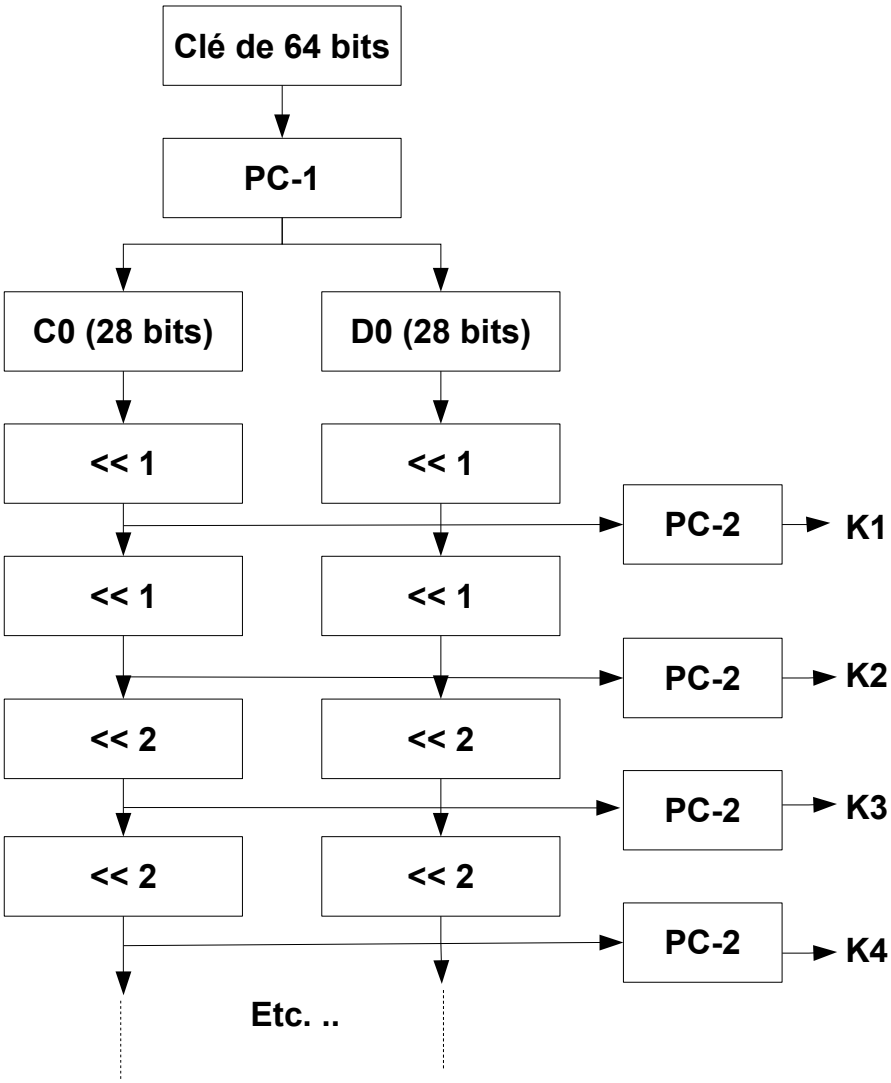


Figure 1.7: Production des clés DES

tation PC-2, qui produit les clés successives à partir des résultats des décalages successifs, est défini à la figure 1.9. Les décalages successifs utilisés pour obtenir les clés d'itérations sont de 1 ou 2 bits, selon une séquence qui dépend de l'itération en question. La figure 1.10 indique les décalages utilisés pour chacune des itérations.

Pour chaque itération, les clés définies plus haut sont utilisées pour transformer le texte entrant, suivant la structure de la figure 1.8. Dans une itération, la moitié du bloc de 64 bits est passé tel quel, alors que l'autre moitié est transformée, selon la méthode de Feistel. Dans le système DES, la partie transformée traverse 5 opérations. La première opération est une expansion, qui produit un bloc de 48 bits à partir d'une entrée de 32 bits. Cette opération apparaît à la figure 1.11.

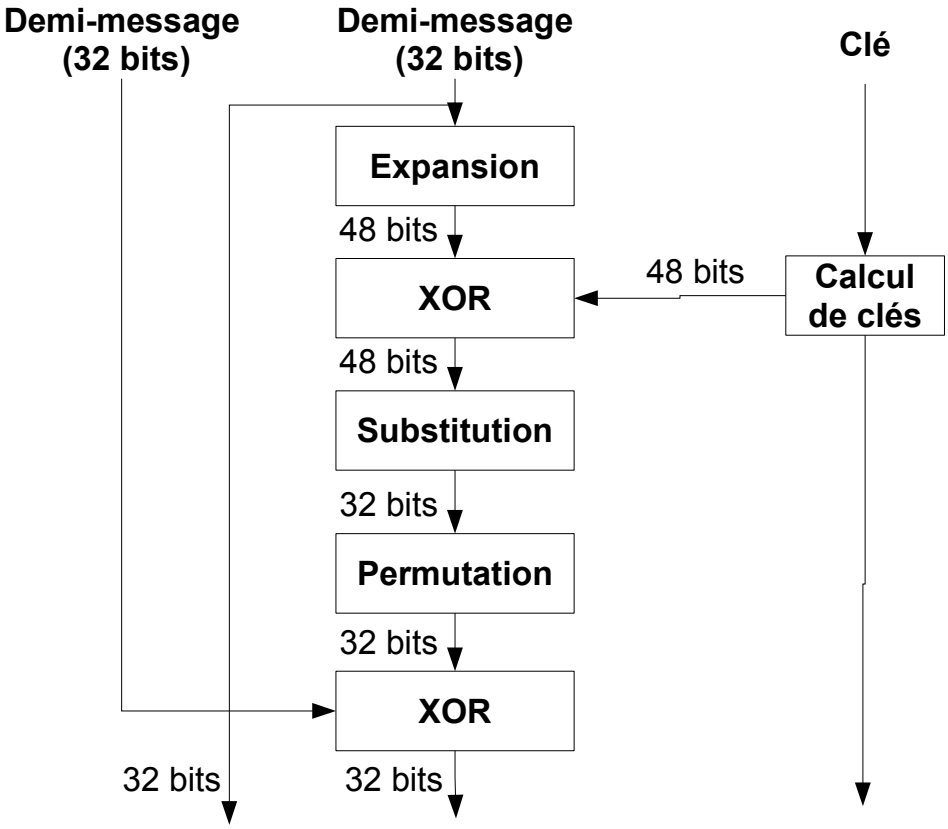


Figure 1.8: Une itération DES

La sortie du module d'expansion est utilisée comme entrée d'un ou-exclusif, où l'autre entrée est la clé (de 48 bits) de cette itération. Cette transformation produit l'entrée de l'étage de substitution. Ici, les 48 bits sont séparés en 8 blocs de 6 bits, lesquels sont utilisés comme entrées des 8 boîtes de substitution. Dans le système DES, chacune des boîtes de substitution est différente. Les 6 entrées d'une boîte de substitution sont utilisées de la façon suivante: le bit le plus faible et le bit le plus fort sont combinés pour sélectionner l'une des 4 lignes de la boîte de substitution. Les 4 bits du milieu de l'entrée sont utilisés pour sélectionner la colonne appropriée (l'une des 16) de la ligne sélectionnée. Le résultat de cette sélection est la sortie de la boîte de substitution donnée. Les 8 boîtes de substitution sont détaillées dans les figures 1.12 et 1.13.

Les sorties des 8 boîtes de substitution sont combinées et deviennent les entrées de la boîte de permutation. Puisque chacune des 8 boîtes de substitution a une sortie de 4 bits, l'entrée de l'unique boîte de permutation a donc 32 bits. La boîte de permutation procède à l'aiguillage des différentes entrées et les redistribue sur les 32 sorties. La fonction d'aiguillage est définie dans la figure 1.14. Enfin,

PC-1						
57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

PC-2							
14	17	11	24	1	5	3	28
15	6	21	10	23	19	12	4
26	8	16	7	27	20	13	2
41	52	31	37	47	55	30	40
51	45	33	48	44	49	39	56
34	53	46	42	50	36	29	32

Figure 1.9: Boîtes de choix de permutation 1 et 2 de DES

	Décalages gauches circulaires															
Itération	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Décalage	**1**	**1**	**2**	**2**	**2**	**2**	**2**	**2**	**1**	**2**	**2**	**2**	**2**	**2**	**2**	**1**

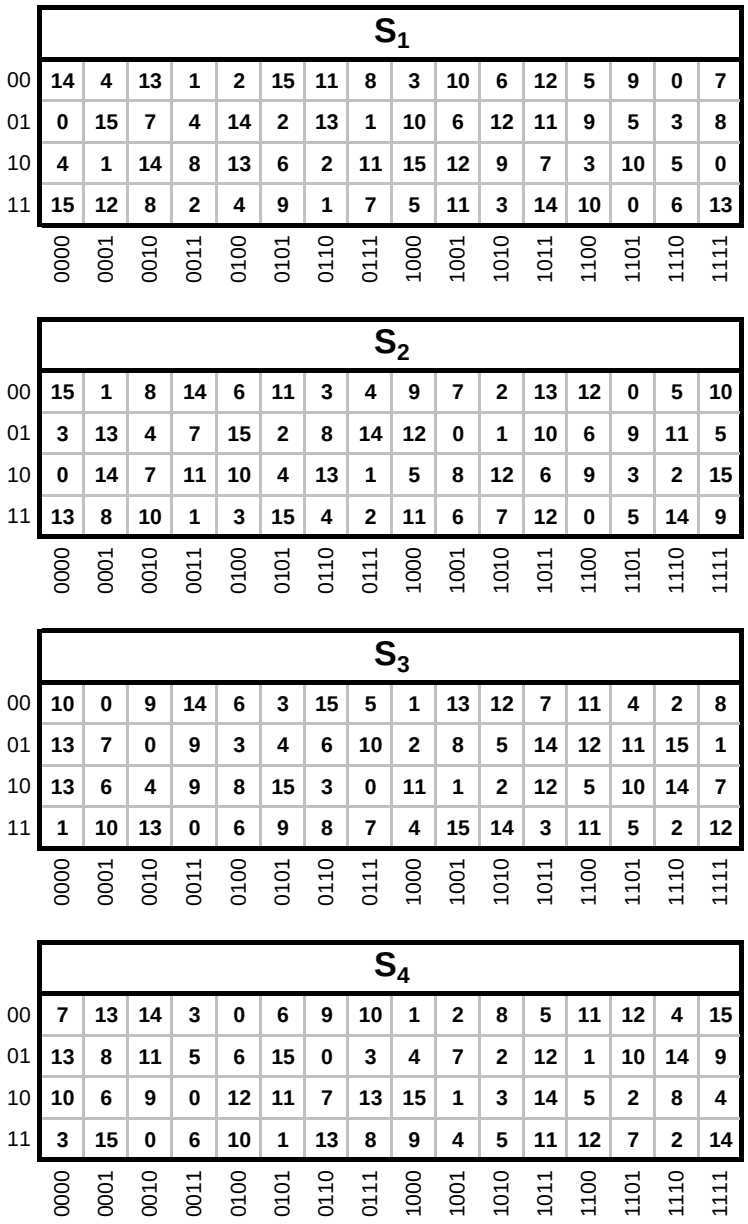
Figure 1.10: Séquence de décalages gauches circulaires de DES

les 32 bits sortant de cette opération sont utilisés comme entrées d'un deuxième ou-exclusif dont l'autre entrée est le demi-message non transformé.

Comme mentionné plus haut, le système de chiffrement DES procède à 16 itérations de calcul pour transformer un bloc de texte de 64 bits. Il faut noter avant de terminer que dans le système DES, il existe une étape qui précède la séquence d'itérations, et une étape (inverse de la première) qui suit la séquence d'itérations. On appelle ces deux transformations la Permutation Initiale (Initial Permutation, IP) et son inverse. Ces deux opérations sont définies par les permutations apparaissant à la figure 1.15.

	$E(R_{i-1})$					
1 - 6	**32**	**1**	**2**	**3**	**4**	**5**
7 - 12	**4**	**5**	**6**	**7**	**8**	**9**
13 - 18	**8**	**9**	**10**	**11**	**12**	**13**
19 - 24	**12**	**13**	**14**	**15**	**16**	**17**
25 - 30	**16**	**17**	**18**	**19**	**20**	**21**
31 - 36	**20**	**21**	**22**	**23**	**24**	**25**
37 - 42	**24**	**25**	**26**	**27**	**28**	**29**
43 - 48	**28**	**29**	**30**	**31**	**32**	**1**

Figure 1.11: E: Fonction d'expansion DES



S_1

	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
01	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
10	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
11	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

S_2

	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
01	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
10	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
11	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

S_3

	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
01	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
10	13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
11	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12

S_4

	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
01	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
10	10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
11	3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

Figure 1.12: S_1 à S_4: Quatre premières boites de substitution DES

S_5

	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
01	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
10	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
11	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

S_6

	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
01	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
10	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
11	4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13

S_7

	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
01	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
10	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
11	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12

S_8

	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
01	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
10	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
11	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Figure 1.13: S_5 à S_8: Quatre dernières boites de substitution DES

	P															
1 - 16	**16**	**7**	**20**	**21**	**29**	**12**	**28**	**17**	**1**	**15**	**23**	**26**	**5**	**18**	**31**	**10**
17 - 32	**2**	**8**	**24**	**14**	**32**	**27**	**3**	**9**	**19**	**13**	**30**	**6**	**22**	**11**	**4**	**25**
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32

Figure 1.14: P: Fonction de permutation DES

IP							
58	**50**	**42**	**34**	**26**	**18**	**10**	**2**
60	**52**	**44**	**36**	**28**	**20**	**12**	**4**
62	**54**	**46**	**38**	**30**	**22**	**14**	**6**
64	**56**	**48**	**40**	**32**	**24**	**16**	**8**
57	**49**	**41**	**33**	**25**	**17**	**9**	**1**
59	**51**	**43**	**35**	**27**	**19**	**11**	**3**
61	**53**	**45**	**37**	**29**	**21**	**13**	**5**
63	**55**	**47**	**39**	**31**	**23**	**15**	**7**

IP^{-1}							
40	**8**	**48**	**16**	**56**	**24**	**64**	**32**
39	**7**	**47**	**15**	**55**	**23**	**63**	**31**
38	**6**	**46**	**14**	**54**	**22**	**62**	**30**
37	**5**	**45**	**13**	**53**	**21**	**61**	**29**
36	**4**	**44**	**12**	**52**	**20**	**60**	**28**
35	**3**	**43**	**11**	**51**	**19**	**59**	**27**
34	**2**	**42**	**10**	**50**	**18**	**58**	**26**
33	**1**	**41**	**9**	**49**	**17**	**57**	**25**

Figure 1.15: Permutation initiale et inverse DES

1.3.4 Méthode IDEA

À compléter...

1.3.5 Standard AES

[IAI06] *À compléter...*

1.3.5.1 Méthode Rijndael

[DR02, DKR97, TW06] *À compléter...*

X_L \ X_H	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Figure 1.16: La boîte de substitution de AES (Rijndael)

1.3.5.2 Méthodes concurrentes (Twofish, RC6, Serpent, MARS)

À compléter...

1.3.6 Mode d'opération des chiffrements par blocs

Par définition, le chiffrement par bloc opère sur des messages de taille prédéfinie (la taille du bloc traité). Pour permettre d'encrypter des messages de tailles arbitraires, il faut donc les découper en séquences de blocs, où chaque bloc peut être traité par l'algorithme de chiffrement choisi. Plusieurs questions doivent être résolues pour effectuer ce découpage de façon reproduisible et sécuritaire:

1. Que faire avec les messages dont la taille n'est pas un multiple de celle du bloc?
2. Comment traiter les différents blocs qui composent le message à encrypter? Doit-on les transformer en isolation ou bien en les regroupant?

Pour résoudre le premier problème, il est usuel d'utiliser du rembourrage (padding en anglais). Typiquement, on ajoutera au message un tampon de taille appropriée, de telle sorte que la taille totale soit un multiple de la taille du bloc de chiffrement. Cependant, on doit noter que cette transformation doit être réversible. Cela implique qu'après décryptage du message, le receveur doit pouvoir retrancher les bits du tampon pour retrouver exactement le message original. Plusieurs types de rembourrages sont possibles. Nous en présentons deux tirés de [FS03]:

- On ajoute au message un octet contenant la valeur 128 ($0x80$), suivi d'autant d'octets nuls (valeur: $0x00$) que nécessaire pour que le message total comporte un nombre entier de blocs. Il est ensuite facile de retirer le tampon à l'arrivée.
- On détermine le nombre d'octets requis dans le tampon. Ce nombre N sera compris entre $1 \leq N \leq B$, où B est la taille des blocs. On ajoute alors ce nombre d'octets au message, chaque octet prenant la valeur N. Cette méthode suppose que la taille des blocs de chiffrement ne dépasse jamais 256, la valeur maximum exprimable par un octet[5], hypothèse raisonnable avec les méthodes modernes de chiffrement par blocs.

Le deuxième problème est important pour la sécurité du système de chiffrement. En effet, puisque le message est encrypté par blocs de taille fixe, un cryptanalyste pourrait potentiellement détecter des groupes de symboles qui se répètent. Puisque la taille des blocs varie entre 128 et 256 bits, des séquences répétées de 7 ou 8 caractères qui seraient alignés sur les blocs pourraient être identifiés,

[5]En réalité, les nombres exprimés par un octet sont compris entre 0 et 255. Cependant, puisque le tampon a une taille minimum de 1, on utilisera l'octet 0 pour indiquer un tampon de 1 octet, jusqu'à l'octet 255 pour indiquer un tampon de 256 octets. On remarque que pour un message original qui est un multiple de la taille du bloc, on doit ajouter un bloc complet de rembourrage.

permettant une analyse statistique partielle du message chiffré. Par exemple, supposons que le message contienne plusieurs instances du mot "cryptographie". Puisque ce mot est plus grand que 8 caractères, il pourrait être à l'origine de blocs chiffrés identiques.

Dans ce qui suit, nous étudierons différentes techniques pour coupler les blocs de chiffrement afin de réduire l'importance de ce problème. Dans les descriptions de chacune des techniques présentées, nous supposons que M est le message à encrypter, décomposé en blocs de taille fixe $M = (M_1, M_2, \cdots)$. De plus, nous appelons $CB(K, b)$ la fonction de chiffrement d'un bloc b à l'aide de la clé K et C_i le bloc chiffré correspondant au bloc M_i du message initial.

1.3.6.1 Mode "Electronic Codebook" (ECB)

La méthode la plus simple de chiffrement par blocs est de découper le message et d'encrypter les blocs indépendemment. On appelle cette façon de procéder le mode "Electronic Codebook", qu'on représente mathématiquement ainsi:

$$C_i \quad = \quad CB(K, M_i) \quad \text{pour} \quad i = 1, 2, \cdots$$

Nous avons vu plus haut que cette façon de chiffrer un message est sujette à des attaques statistiques. Ces attaques seront d'autant plus efficaces que le message chiffré est long, puisque la possibilité d'observer des blocs identiques augmentera. Quoique simple d'utilisation, cette méthode est donc à proscrire pour les longs messages, puisqu'elle est alors particulièrement insécure. Cependant, le mode ECB peut être très efficace pour transmettre des messages courts dont le contenu est non-répétitif. Par exemple, ce mode peut être utile pour transmettre des clés d'encryption[6].

1.3.6.2 Mode "Cipher Block Chaining" (CBC)

Une façon courante d'éliminer les problèmes liés au mode ECB est de lier le chiffrement d'un bloc à celui du bloc qui le précède. Le "Cipher Block Chaining" réalise ce lien en exécutant un ou-exclusif entre le bloc à chiffrer et le chiffre du bloc précédent. Le bloc ainsi obtenu est ensuite chiffré. Le résultat de cette opération est utilisé sur le bloc suivant, et ainsi de suite. Alors:

$$C_i \quad = \quad CB(K, M_i \bigoplus C_{i-1}) \quad \text{pour} \quad i = 1, 2, \cdots$$

[6]Il faut toutefois remarquer que pour utiliser le mode ECB dans un système de chiffrement par blocs, il faut que la clé symétrique ait été échangée auparavant. Le mode ECB peut tout de même être utile, par exemple si un serveur central distribue des clés symétriques utilisées pour des échanges entre pairs.

1.3.6.2.1 Vecteur d'Initialisation (VI) En utilisant le CBC, on élimine le problème des blocs répétés, chiffrés de la même façon. Cependant, il reste encore à déterminer la première valeur à utiliser pour démarrer le système (C_0). On appelle cette valeur le Vecteur d'Initialisation (Initialization Vector, ou IV, en anglais) [FS03, Sta03]. On peut être tenté d'utiliser un VI fixe pour amorcer la chaîne. Cependant, l'utilisation d'un VI fixe et connu de tous réintroduit le problème du ECB, pour le premier bloc des messages cette fois. Or il arrive fréquemment que les messages débutent avec la même séquence de caractères (par exemple, formule de politesse dans une lettre ou codes prédéterminés pour indiquer le type de message). Pour cette raison, on doit éviter d'utiliser des VI fixes. Il existe cependant plusieurs alternatives:

- VI par compteur: On peut imaginer remplacer le VI fixe par un compteur, qui est augmenté à chaque nouveau message. Ceci malheureusement n'est pas totalement sécuritaire: lorsqu'on incrémente un compteur, les bits des nombres successifs vont souvent être semblables. Par exemple, lorsqu'on incrémente un nombre pair, le nombre suivant n'a que le bit 0 qui diffère. Si les premiers blocs de deux messages successifs ne diffèrent que d'un bit, l'utilisation du compteur comme paramètre du ou-exclusif viendrait éliminer la différence, et l'encryptage du premier bloc de chacun de ces deux messages serait identique.
- VI aléatoires: Une autre solution est d'utiliser un VI aléatoire. Ceci permet d'éviter qu'un tiers puisse savoir quel est le VI utilisé. Par contre, cela implique qu'on doive transmettre le VI au destinataire, pour qu'il puisse amorcer le décryptate. Pour ce faire, on peut transmettre le VI aléatoire comme premier paquet (chiffré) du message:

$$\begin{array}{rcll} C_0 & = & CB(K, \text{VI}) & \text{où VI est un nombre aléatoire} \\ C_i & = & CB(K, M_i \bigoplus C_{i-1}) & \text{pour} \quad i = 1, 2, \cdots \end{array}$$

  Quoiqu'intéressante, l'idée introduit quelques problèmes supplémentaires:

  1. Il est important d'utiliser un générateur de nombres aléatoires efficace et digne de confiance, ce qui exige un soin particulier [Knu98, DOF+90].
  2. Le message chiffré doit contenir un bloc de plus que le message original. Pour des courts messages envoyés en grand nombre, ceci peut devenir un frein à la performance.

- VI "nuniques": Un "nunique", ou nombre unique (en anglais nonce, pour Number used Once) est un nombre qui n'est utilisé qu'une seule fois pour une clé de chiffrement donnée. On associe un nunique à chaque message à encrypter. Pour ce faire, la méthode typique est de numéroter les messages, puis de déterminer le VI en encryptant ce numéro du message:

$$\begin{array}{lcll} \text{VI} & = & CB(K, N_{un}) & \text{où } N_{un} \text{ est le nunique associé au message} \\ C_0 & = & \text{VI} & \text{le VI est l'amorce du CBC} \\ C_i & = & CB(K, M_i \bigoplus C_{i-1}) & \text{pour} \quad i = 1, 2, \cdots \end{array}$$

Pour que le receveur puisse déchiffrer le message, il doit pouvoir retrouver le VI. Ce dernier étant dérivé du numéro de message, il est donc suffisant d'inclure le numéro de message au début de l'envoi. Cette façon de faire a l'avantage de requérir moins d'espace (le N_{un} sera typiquement beaucoup plus petit (32 à 48 bits sont suffisants) que le VI (qui aura la taille des blocs de chiffrement, soit entre 128 et 256 bits)). De plus, cela permet au receveur de vérifier l'ordre et le nombre de messages. Il est important de remarquer que pour être sécuritaire, cette méthode doit s'assurer que les nuniques sont vraiment uniques et donc qu'un numéro de message n'est jamais réutilisé avec une clé K donnée. On remarque aussi que C_0 (VI) n'a pas besoin d'être transmis, puisque le receveur du message peut le recalculer à partir du numéro de message et de la clé K.

1.3.6.3 Mode "Output Feedback" (OFB)

Dans le mode CBC, la perte d'un bloc intermédiaire empêche le déchiffrement du bloc qui suit (perte de deux blocs au total). Ce problème n'existe pas dans le chiffrement par flux (section 1.2), puisque le système fonctionne à l'aide d'une séquence de clés variables. Or il est possible de transformer un système de chiffrement par bloc en un système de chiffrement par flux . Le principe est de générer une séquence de nombres avec un système de chiffrement par bloc et ensuite de les utiliser comme clés variables d'un système de chiffrement par flux:

$$\begin{array}{lcll} K_0 & = & \text{VI} & \\ K_i & = & CB(K, K_{i-1}) & \text{pour} \quad i = 1, 2, \cdots \\ C_i & = & M_i \bigoplus K_i & \text{pour} \quad i = 1, 2, \cdots \end{array}$$

où K est la clé d'encryptage symétrique, $(K_0, K_1, K_2, \cdots)$ est la séquence de clés utilisées pour le chiffrement par flux (via l'opération ou-exclusif entre le bloc de message M_i et la clé de chiffrement par flux K_i) et VI est le Vecteur d'Initialisation. L'intérêt du OFB est que l'encryptage et le décryptage sont des opérations identiques. En effet, pour décrypter le message chiffré, on doit regénérer la suite de clés $(K_0, K_1, K_2, \cdots)$, et effectuer de nouveau les opérations ou-exclusif sur les éléments chiffrés. Un système unique est donc suffisant pour le chiffrement et le déchiffrement. De plus, puisque l'opération ou-exclusif

peut être appliquée sur un octet unique, il n'est plus nécessaire de faire de rembourrage si la taille du message n'est pas un multiple de la taille des blocs. Il suffit de tronquer la dernière clé, et de n'effectuer le ou-exclusif que sur le nombre d'octets restants dans le message à chiffrer. Pour de petits messages, c'est un net avantage, puisque le message chiffré a la même taille que le message d'origine.

Malheureusement, le mode OFB est aussi sujet à certains problèmes. Puisque c'est un système de chiffrement par flux, il est très important que les clés utilisées pour faire les opérations ou-exclusif ne soient pas répétées. Si les clés sont effectivement répétées, il est alors possible pour un cryptanalyste de déchiffrer le message en tout ou en partie. Or si aucune précaution n'est prise, cela est malheureusement possible:

- Si un VI est réutilisé pour une même clé K, alors toute la séquence de clés sera identique (elle ne dépend que de VI et de K). Cela permet de faire une analyse différentielle de messages. En effet, soient deux messages $M = \{M_1, M_2, \cdots\}$ et $M' = \{M'_1, M'_2, \cdots\}$ encryptés à l'aide de la même séquence de clés $\{K_1, K_2, \cdots\}$. En effet:

$$\begin{array}{rlcll} & K_0 & = & \text{VI} & \\ & K_i & = & CB(K, K_{i-1}) & \text{pour} \quad i = 1, 2, \cdots \\ & C_i & = & M_i \bigoplus K_i & \\ & C'_i & = & M'_i \bigoplus K_i & \\ \text{d'où:} & C_i \bigoplus C'_i & = & M_i \bigoplus K_i \bigoplus M'_i \bigoplus K_i & \\ & & = & M_i \bigoplus M'_i & \end{array}$$

- Si une séquence de clés boucle sur elle-même rapidement (à la limite produisant une séquence de clés identiques), alors l'analyse statistique des blocs devient possible, et le message chiffré n'est plus en sécurité.

1.3.6.4 Mode "Counter" (CTR)

Le mode "Compteur", ou CTR, a été proposé dès 1979 par Diffie et Hellman [DH79]. Cependant, il n'était pas mentionné dans les méthodes standards associées à DES[NIoST80], et son utilisation a été peu fréquente jusqu'au début des années 2000. C'est toutefois un mode d'opération de systèmes de chiffrement par blocs qui présente de nombreux avantages[LRW00] et qui est maintenant un standard du NIST[Dwo01]. Ce mode d'opération produit un système de chiffrement par flux , tout comme le mode OFB. Ici, on utilise un nunique et un compteur pour produire la séquence de clés de chiffrement par flux:

$$\begin{array}{rcll} K_i & = & CB(K, N_{un} \| \, i) & \text{pour} \quad i = 1, 2, \cdots \\ C_i & = & M_i \bigoplus K_i & \end{array}$$

où $\|$ représente la concaténation de deux séquences de bits, N_{un} (un nunique) et i, le numéro du bloc à traiter. Ceci implique que la somme des tailles de ces deux éléments doit être plus petite que la taille des blocs, qui est d'au moins 128 bits dans les systèmes modernes tels AES. Comme auparavant, K est la clé de chiffrement symétrique, K_i est la clé de chiffrement par flux du bloc de message M_i et le résultat chiffré est C_i. Ici encore, il est impératif de n'utiliser chaque nunique qu'une seule fois pour une clé k donnée.

Chapitre

2

Concepts de base de la théorie des nombres

Les algorithmes à clés publiques (tels DH, Elgamal et RSA que nous verrons au chapitre 4) utilisent la multiplication, la division et l'élévation à une puissance de très grands nombres premiers et composés, le tout en présence de l'opérateur modulo[1]. Pour maîtriser ces méthodes d'encryptage, il est donc essentiel de bien comprendre les caractéristiques de ces opérations et les façons de les effectuer, en particulier avec de très grands nombres.

Nous allons tout d'abord étudier l'opérateur modulo et les propriétés des nombres qu'il génère. Nous verrons ensuite comment se fait le calcul efficace de puissances en présence de l'opérateur modulo. Par la suite, nous parlerons des nombres premiers, de la décomposition des nombres composés en produits de nombres premiers et du Plus Grand Commun Diviseur (PGCD), calculé à l'aide de l'algorithme d'Euclide. Puis nous verrons comment, à l'aide d'une version modifiée de l'algorithme d'Euclide, il est possible de trouver l'inverse multiplicatif d'un nombre en présence du modulo, ce qui permet d'effectuer la division de deux nombres modulo un troisième. Nous étudierons aussi certaines caractéristiques des nombres entiers qui sont fondamentales pour le fonctionnement des méthodes de cryptage à clés publiques/privées, en discutant du petit théorème de Fermat, de l'indicatrice d'Euler et des nombres de Carmichael. Nous poursuivrons ce chapitre en voyant comment vérifier qu'un nombre est premier, au moyen du crible d'Ératosthène et du test de Miller-Rabin, et comment utiliser ces méthodes de vérification pour découvrir de très grands nombres premiers. Nous terminerons le chapitre 2 par une présentation du théorème du reste chinois, qui permet entres autres de vérifier la validité de RSA, et qui dans certains cas permet aussi de réduire significativement les temps de calcul.

[1]Notons que la division en présence de l'opération modulo ne correspond pas à la division standard entre les nombres entiers ou réels. On l'appelle "division", mais c'est simplement l'opération qui est l'inverse de la multiplication en présence de l'opérateur modulo. Elle se calcule très différemment de la division standard.

Avant de commencer, il est intéressant de noter deux choses:

1. Les concepts mathématiques dont on parlera existent pour la grande majorité depuis fort longtemps : on mentionnera entre autres Euclide et Ératosthène, qui vivaient en Grèce aux 3^{e} et 2^{e} siècles avant JC, Sun Zi, un mathématicien chinois du 3^{e} siècle après JC, Fermat, Euler et Galois, qui vécurent aux 17^{e}, 18^{e} et 19^{e} siècles.

2. La théorie des nombres a longtemps été considérée comme une branche "pure" des mathématiques, c'est-à-dire sans application pratique [Kob07, Har40]. Cependant, depuis 1976 et 1977, avec l'introduction par Diffie et Hellman, puis Rivest, Shamir et Adleman, de concepts de la théorie des nombres pour la cryptographie, cette discipline est entrée de plein pied dans le domaine pratique...

2.1 Introduction à DH et RSA

Avant d'étudier les concepts mathématiques, nous présenterons brièvement les méthodes Diffie-Hellman (DH) et Rivest-Shamir-Adleman (RSA), pour illustrer l'usage des opérations mathématiques qui seront étudiées par la suite avec plus de détails. Les méthodes DH et RSA reposent sur un ensemble de concepts mathématiques qu'on doit bien posséder pour utiliser correctement ces techniques d'encryptage. Cette courte présentation initiale est nécessairement incomplète. Les concepts mathématiques sous-jacents seront présentés plus loin dans ce chapitre. Nous reverrons DH et RSA plus en profondeur au chapitre 4.

La méthode de Diffie et Hellman, publiée en 1976[2], a eu un impact considérable dans le domaine de la cryptographie. Ces deux chercheurs ont en effet proposé un nouveau paradigme, où il n'est plus nécessaire de faire l'échange préalable sécurisé d'une clé privée pour participer à un échange d'information protégé.

Dans la méthode DH, deux interlocuteurs échangent deux nombres utilisés par la suite par chacun pour produire un troisième nombre qui ne peut être connu que d'eux seuls. Soient Alice et Bob, les deux interlocuteurs, qui s'entendent au préalable sur deux nombres de départ, g et p . Ces nombres n'ont pas à être secrets. Alice produit le nombre $G_A = g^x \bmod p$ en choisissant elle-même un nombre x, qu'elle prend soin de conserver en lieu sûr. De même, Bob produit le nombre $G_B = g^y \bmod p$, conservant lui aussi le nombre y. Ensuite, Alice envoie G_A à Bob, et Bob envoie G_B à Alice, cet envoi pouvant se faire par un canal de communication non sécurisé. Par la suite, chacun peut obtenir un nouveau nombre G_{AB} en utilisant l'exposant initial (x pour Alice, y pour Bob):

[2]Un troisième chercheur, Ralph Merkle, a aussi contribué à l'élaboration de cette méthode. Il fait d'ailleurs partie des co-inventeurs du brevet américain 4 200 770, déposé le 6 septembre 1977 pour protéger la méthode DH [USP06].

$$\begin{aligned} G_{AB} &= (G_A)^y \bmod p \\ &= (G_B)^x \bmod p \\ &= g^{xy} \bmod p \end{aligned}$$

Ce nombre G_{AB}, qui n'est connu que d'Alice et Bob, peut ensuite être utilisé pour faire de l'encryptage symétrique à clé unique (ce type d'encryptage est essentiellement ce qui était utilisé avant 1976).

En 1978, soit 2 ans après la publication de l'algorithme de Diffie-Hellman, Rivest, Shamir et Adleman ont proposé une nouvelle méthode d'encryptage à clé publique[3] [RSA78]. Cette méthode utilise l'opération modulo, tout comme la méthode DH. Cependant, à la différence de cette dernière, la méthode RSA permet directement l'encryptage d'un message.

Dans la méthode RSA, il s'agit de trouver deux nombres e et d qui, lorsque utilisés comme puissances successives d'un nombre m en présence du modulo n , redonnent le nombre initial m. Mathématiquement, en supposant un message de départ représenté par le nombre m, la méthode RSA utilise la propriété suivante : $m^{e\times d} = m \bmod n$. Ici, Alice envoie les nombres e et n à Bob, qui calcule $c = m^e \bmod n$ et l'envoie à Alice. Celle-ci calcule $c^d \bmod n = (m^e)^d \bmod n$ et récupère ainsi le message m. En supposant que personne à part Alice ne peut connaître d, elle est la seule qui puisse déchiffrer le message de Bob.

Il faut remarquer que les nombres g et p utilisés par DH ainsi que les nombres n et d utilisés par RSA doivent être très grands. Si ce n'est pas le cas, il est alors très facile de découvrir l'information qui devait être sécurisée.

2.2 Modulos, résidus et congruences

L'opérateur modulo (mod) est bien connu. Pour effectuer cette opération, il s'agit de diviser un nombre entier a par un nombre entier n (la base), et de conserver le reste r. Ainsi, $a = ns + r$ (où s est un nombre entier) et on définit $a \bmod n = r$, avec $0 \leq r < n$. On appelle aussi r "le résidu de a modulo n".

L'utilisation de l'opérateur mod dans les démonstrations mathématiques peut parfois être une source de confusion. Ainsi, les mathématiciens écrivent habituellement $a \equiv r \pmod n$, indiquant que r et a appartiennent à la même classe d'équivalence "modulo n". En effet, si l'on considère l'ensemble $\mathbb{Z}$ des nombres entiers, l'opération "modulo n" les sépare en n classes distinctes (par exemple, 2, 9, 16 sont dans la même classe modulo 7 alors que 3, 10, 17 sont aussi dans une même classe modulo 7, mais différente de la première). Cette façon mathéma-

[3]Rivest, Shamir et Adleman avaient inventé leur méthode dès 1977. Elle avait même été présentée la même année par Martin Gardner dans sa chronique de septembre du Scientific American. La publication officielle est venue plus tard, en partie à cause de démêlés avec la National Security Agency américaine. En parallèle, les trois chercheurs ont aussi déposé la demande de brevet américain 4 405 829 le 14 décembre 1977 [USP06].

tique d'écrire la relation modulo peut paraître surprenante à première vue pour un habitué de la programmation, où la ligne de code[4] $a = r \bmod n$ indiquerait qu'on divise r par n, et que le reste est ensuite écrit dans la variable a. Dans la suite de ce texte, nous utiliserons les deux formes d'écriture ($a \bmod n = r$ et $a \equiv r \pmod{n}$), la deuxième forme étant toujours utilisée selon l'usage des mathématiciens, jamais selon celui des informaticiens.

On dit que deux nombres a et b sont "congruents" modulo n si $a \equiv b \pmod{n}$.

Soit	$a \equiv r \pmod{n}$	
et	$b \equiv r \pmod{n}$,	
alors	$a - b = kn$	($k \in \mathbb{Z}$, i.e. k est un nombre entier)
En effet,	$a \equiv r \pmod{n} \Rightarrow a = ns + r$	$(s \in \mathbb{Z})$
De même,	$b \equiv r \pmod{n} \Rightarrow b = nt + r$	$(t \in \mathbb{Z})$
D'où	$a - b = (ns + r) - (nt + r)$	
	$a - b = n(s - t)$	
$\therefore$	$a - b = nk$	$(k \in \mathbb{Z})$

L'opération modulo a des caractéristiques particulières, qui découlent de sa construction. Pour chaque base n , il existe un nombre fini de résidus possibles : $\{0, 1, 2, \cdots, n-1\}$. On appelle cet ensemble Z_n, l'ensemble des résidus de la base n. On peut vérifier que l'addition et la multiplication modulo n de deux membres de Z_n donnent des résultats qui sont aussi dans Z_n.

Par exemple, utilisons la base 7 et multiplions ensemble les nombres 5 et 6 dans cette base:

$$Z_7 = \{0, 1, 2, 3, 4, 5, 6\}$$

$$\begin{aligned} 5 \times 6 &= 30 \\ &= 28 + 2 \\ &= 4 \times 7 + 2 \\ &\equiv 2 \pmod{7} \qquad \text{(le résultat, 2, est encore dans } Z_7\text{)} \end{aligned}$$

L'addition et la multiplication ont les caractéristiques suivantes dans Z_n:

0 est l'élément neutre de l'addition
1 est l'élément neutre de la multiplication
$\forall a \in Z_n$ il existe un inverse additif $-a$
tel que $a + (-a) \equiv 0 \pmod{n}$
si p est un nombre premier, $\forall a \in Z_p$, $(a \neq 0)$ il existe un inverse multiplicatif a^{-1}
tel que $aa^{-1} \equiv 1 \pmod{p}$
0 n'a pas d'inverse multiplicatif

[4]en C, C++, C# et Java, cette ligne de code apparaîtrait sous la forme : "$a = r \,\%\, n$;"

Par exemple, pour l'addition dans Z_7:

0 est son propre inverse	
1 et 6 sont inverses	$(1 + 6 = 7 \equiv 0 \pmod 7)$
2 et 5 sont inverses	$(2 + 5 = 7 \equiv 0 \pmod 7)$
3 et 4 sont inverses	$(3 + 4 = 7 \equiv 0 \pmod 7)$

De même, pour la multiplication dans Z_7:

1 est son propre inverse	
2 et 4 sont inverses	$(2 \times 4 = 8 \equiv 1 \pmod 7)$
3 et 5 sont inverses	$(3 \times 5 = 15 \equiv 1 \pmod 7)$
6 est son propre inverse	$(6 \times 6 = 36 \equiv 1 \pmod 7)$

En fait, ces caractéristiques de l'opérateur modulo font que les éléments

$$\{1, 2, 3, \cdots, n-1\} \in Z_n$$

c'est-à-dire Z_n duquel on a enlevé l'élément 0 et qu'on représente par Z_n^*, forment un corps de Galois (en anglais, "Galois Field"'). Dans de nombreux livres et articles de cryptographie de langue anglaise, on parle d'ailleurs de GF(n), pour "Galois Field" de n. Voir l'annexe A pour plus d'information sur les corps, les anneaux et les groupes.

Il est intéressant d'étudier ce qui se passe lorsqu'on multiplie tous les éléments de Z_n^* par une même constante c (modulo n). Pour certaines valeurs de n et c, on obtiendra de nouveau tous les éléments de Z_n^*. Par contre, il existe des valeurs de n et c pour lesquelles on ne retrouve pas tous les éléments de Z_n^*. Par exemple, considérons $n = 8$, où $Z_n^* = \{1, 2, 3, 4, 5, 6, 7\}$. La figure 2.1 indique le résultat du produit (modulo n) de la constante c par l'élément $Z_n^*(i)$ de Z_n^*, pour différentes valeur de c.

c Z(i)	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7
2	2	4	6	0	2	4	6
3	3	6	1	4	7	2	5
4	4	0	4	0	4	0	4
5	5	2	7	4	1	6	3
6	6	4	2	0	6	4	2
7	7	6	5	4	3	2	1

Figure 2.1: Résultats du produit dans Z_8^*

On s'aperçoit que pour certaines valeurs de c, les résultats qu'on obtient lors des multiplications modulo n ne représentent pas tous les éléments possibles de Z_n^*. Dans ces cas, on observe des séries de valeurs qui se répètent, représentées dans le tableau par des zones grisées foncées. On voit par contre qu'il existe des valeurs de c pour lesquelles tous les éléments de Z_n^* apparaissent dans les résultats, comme indiqué dans les zones grisées pâles du tableau. On dit alors que ces valeurs de c "génèrent" Z_n^*, ou encore que ces valeurs de c sont des "génératrices" de Z_n^*. Ainsi, on s'aperçoit dans le tableau précédent que 1, 3, 5 et 7 génèrent Z_8^*. On note que les valeurs de c qui génèrent Z_8^* sont celles qui n'ont aucun facteur commun avec 8 (autre que 1). On dit de ces nombres sans facteur commun qu'ils sont relativement premiers. On peut généraliser l'observation précédente et prouver que seules les valeurs de c qui sont relativement premières avec n vont générer Z_n^*. Dans le cas où n est un nombre premier, alors toutes les valeurs de c qui ne sont pas des multiples de n vont être relativement premières avec n, et donc toutes les valeurs de $c \in Z_n^*$ généreront Z_n^*. Par exemple, pour $n = 7$, on obtient la figure 2.2, où toutes les valeurs de c génèrent Z_7^*.

c Z(i)	1	2	3	4	5	6
1	1	2	3	4	5	6
2	2	4	6	1	3	5
3	3	6	2	5	1	4
4	4	1	5	2	6	3
5	5	3	1	6	4	2
6	6	5	4	3	2	1
7	0	0	0	0	0	0

Figure 2.2: Résultats du produit dans Z_7^*

On note que la multiplication de $Z_n^*(i)$ par n ne va donner que des 0. Ce résultat est évident lorsqu'on y réfléchit quelques instants : par définition, l'opération modulo a comme résultat le reste de la division d'un nombre par la base. Si ce nombre est un multiple de la base, alors il n'y aura pas de reste, et le résidu sera 0.

L'existence de valeurs génératrices de l'opérateur modulo est essentielle pour les systèmes de cryptographie. En effet, toute opération d'encryptage consiste à remplacer un groupe de caractères $XY \ldots Z$ par un groupe de même taille de caractères différents $RS \ldots T$. L'opération de décryptage revient alors à faire l'opération inverse, pour retrouver les caractères originaux. Si deux groupes de caractères initiaux différents (par exemple $XY \ldots Z$ et $AB \ldots C$) sont cryptés par le même groupe de caractères résultants (par exemple $RS \ldots T$), alors il n'est plus possible de faire le décryptage : Le même groupe de caractères cryptés ($RS \ldots T$) doit produire deux séquences différentes (soit $XY \ldots Z$ ou $AB \ldots C$) pour retrouver le message original, ce qui est impossible à faire. Par exemple, supposons que nous voulions crypter le message suivant, qui comporte deux

nombres : (3,5). Supposons que notre méthode d'encryptage soit de multiplier chacun des nombres par 4, et d'effectuer ensuite le modulo 8. On obtient alors : $3 \times 4 \bmod 8 = 4$, et $5 \times 4 \bmod 8 = 4$. Le message crypté est donc : (4,4), qu'il est ensuite impossible de décrypter correctement (comment savoir que le premier 4 doit être décrypté en 3, alors que le deuxième 4 doit produire un 5 ?). Cependant, si on avait utilisé la base 7 au lieu de la base 8, on aurait obtenu $3 \times 4 \bmod 7 = 5$ et $5 \times 4 \bmod 7 = 6$. Le message crypté obtenu aurait été (5,6), qu'on peut décrypter en observant la figure 2.2 à la ligne 4: les valeurs 5 et 6 apparaissent aux colonnes 3 et 5, les deux nombres de départ utilisés pour notre technique minimaliste d'encryptage.

2.2.1 Calcul du modulo de produits et de puissances

Dans le domaine de la cryptographie, l'opération modulo est habituellement utilisée sur de très grands nombres. De plus, ces grands nombres sont souvent le résultat de la multiplication ou de la mise à une puissance de nombres eux-mêmes très grands. Le plus souvent, ces très grands nombres sont beaucoup plus grands que ceux qui peuvent être calculés directement sur un processeur 32 ou même 64 bits. Dans ce qui suit, nous verrons comment on peut théoriquement effectuer les opérations modulo sur des nombres qui dépassent la taille de l'unité de calcul du processeur sur laquelle on effectue les calculs.

2.2.2 Méthode binaire des exposants

Soient trois très grands nombres entiers a, w, n avec lesquels on doit calculer $a^w \bmod n$. En règle générale, dans les calculs de modulos utilisés en cryptographie, l'exposant w sera très grand et il sera impossible de calculer directement a^w pour ensuite lui appliquer l'opération modulo n.

Par exemple, supposons que a et w sont des nombres de 256 bits (ce qui en ferait des nombres relativement petits selon les standards modernes de cryptographie). Le nombre de bits requis pour écrire le résultat serait:

$$\begin{aligned} \log_2 a^w &= w \times \log_2 a \\ &\approx 2^{256} \times 256 \\ &\approx 2^{264} \\ &\approx 3 \times 10^{79}, \end{aligned}$$

nombre qui correspond à peu près à la quantité de particules dans l'univers. En cryptographie, chacun des bits est important (on ne peut faire d'arrondis sous peine de perdre le message). Même en n'utilisant qu'une particule élémentaire par bit, nous aurions tout juste assez de l'univers entier pour mettre ce nombre en mémoire...

On réalise rapidement qu'il est nécessaire d'effectuer l'opération modulo pendant les calculs intermédiaires de a^w pour conserver des nombres de taille raisonnable. Pour ce faire, observons comment nous pouvons décomposer la mise à une puissance en une série d'étapes simples, chacune des étapes pouvant être couplée à l'opération modulo pour réduire la taille des calculs intermédiaires. Remarquons tout d'abord que nous pouvons utiliser la représentation binaire (base 2) de w pour déterminer une série de mises au carré et de multiplications par a. En effet, chacun des bits de w correspond à une puissance de deux à laquelle on peut arriver par mises au carré successives.

Voyons tout d'abord un exemple sans opération modulo, avec $a = 3$, $w = 13$:

$$
\begin{array}{rcll}
w & = & 13_{10} & \text{(13 est en base 10)} \\
 & = & 1101_2 & \text{(1101 est en base 2)} \\
w[3] & = & 1 & (w[i] \text{ représente le bit } i \text{ de } w) \\
w[2] & = & 1 & \\
w[1] & = & 0 & \\
w[0] & = & 1 & \\
w & = & 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 & \\
 & = & ((1 \times 2 + 1) \times 2 + 0) \times 2 + 1 & \\
 & = & w[3] \times 2^3 + w[2] \times 2^2 + w[1] \times 2^1 + w[0] \times 2^0 & \\
 & = & ((w[3] \times 2 + w[2]) \times 2 + w[1]) \times 2 + w[0] &
\end{array}
$$

d'où:

$$
\begin{array}{rcl}
a^w & = & a^{2^3+2^2+2^0} \\
 & = & a^{((1\times2+1)\times2+0)\times2+1} \\
 & = & (((a)^2 \times a)^2 \times 1)^2 \times a \\
 & = & a^{((w[3]\times2+w[2])\times2+w[1])\times2+w[0]}
\end{array}
$$

De l'exemple précédent, on peut tirer une méthode pour faire le calcul d'une puissance: chaque multiplication par 2 dans l'exposant correspond à une mise au carré de la puissance à calculer, et chaque addition de 1 dans l'exposant correspond à une multiplication par a. On commence par le bit de plus fort poids de l'exposant (en anglais, Most Significant Bit, ou MSB), pour produire le résultat par itérations successives (les résultats successifs seront appelés $r_0, r_1, r_2, r_3, \cdots, r_F$ où r_F est le résultat final et $r_0 = 1$).

Par exemple, avec $a = 3, w = 13 = 1101_2$, on obtient :

$$
\begin{array}{lllll}
\text{Début:} & r_0 & = & 1 \\
1^{\text{ère}} \text{ itération :} & w[3] & = & 1, \\
& r_1 & = & r_0^2 \times a^1 \\
& & = & (1^2 \times 3^1) \\
& & = & 3 \\
2^{\text{e}} \text{ itération :} & w[2] & = & 1, \\
& r_2 & = & (r_1^2 \times a^1) \\
& & = & (3^2 \times 3^1) \\
& & = & 27 \\
3^{\text{e}} \text{ itération :} & w[1] & = & 0, \\
& r_3 & = & (r_2^2 \times a^0) \\
& & = & (27^2 \times 3^0) \\
& & = & 729 \\
4^{\text{e}} \text{ itération :} & w[0] & = & 1, \\
& r_4 & = & (r_3^2 \times a^1) \\
& & = & (729^2 \times 3^1) \\
& & = & 1\,594\,323
\end{array}
$$

On peut modifier la séquence d'opérations précédente pour effectuer une opération modulo après chaque multiplication, pour s'assurer que les calculs intermédiaires ne produisent jamais de nombres de taille exagérée. Ainsi, pour chaque itération de la séquence ci-haut, on doit faire un modulo après la mise au carré, de même qu'après la multiplication par a (lorsque le bit correspondant de l'exposant w n'est pas nul).

Reprenons maintenant l'exemple précédent, en calculant le modulo. Soit $a = 3, w = 13, n = 7$. Alors on peut calculer $a^w \bmod n$ de la façon suivante:

$$
\begin{array}{lllll}
1^{\text{ère}} \text{ itération :} & w[3] & = & 1, \\
& r_1 & = & 3^1 \bmod 7 \\
& & = & 3 \\
2^{\text{e}} \text{ itération :} & w[2] & = & 1, \\
& r_2 & = & (r_1^2 \bmod 7 \times 3^1) \bmod 7 \\
& & = & (3^2 \bmod 7 \times 3) \bmod 7 \\
& & = & (2 \times 3) \bmod 7 \\
& & = & 6 \\
3^{\text{e}} \text{ itération :} & w[1] & = & 0, \\
& r_3 & = & (r_2^2 \bmod 7 \times 3^0) \bmod 7 \\
& & = & (6^2 \bmod 7) \\
& & = & (36 \bmod 7) \\
& & = & 1 \\
4^{\text{e}} \text{ itération :} & w[0] & = & 1, \\
& r_4 & = & (r_3^2 \bmod 7 \times 3^1) \bmod 7 \\
& & = & (1^2 \bmod 7 \times 3) \bmod 7 \\
& & = & 3
\end{array}
$$

On vérifie dans l'exemple précédent que

$$3^{13} \bmod 7 = 1\,594\,323 \bmod 7 = 3.$$

En reformulant la méthode précédente sous forme récursive, on obtient l'algorithme 2.1.

```
puissance_mod ( a, w, n ) {
  if (w == 0) return ( 1 ) ;          //   fin de recursion
  r = puissance_mod ( a, w/2, n );    //   w / 2 : division entiere
  r = (r x r) % n;                    //   % est l'operateur modulo
  if ( w % 2 ) {                      //   bit 0 de w non-nul
    r = ( r x a ) % n ;
  }
  return( r );
}
```

Algorithme 2.1: Élévation à une puissance modulo la base

Note : on suppose ici et pour l'ensemble des algorithmes décrits dans ce texte qu'on utilise un langage de programmation qui permet directement les opérations arithmétiques sur de très grands nombres. Cette hypothèse est vraie pour les langages Python et Perl, de même que pour le logiciel GNU bc. Cependant, pour les langages usuels tels C, C++, Java et C#, on ne peut utiliser directement de très grands nombres. On doit auparavant définir un ensemble de classes qui supportent la représentation et les opérations arithmétiques élémentaires (+,-,/,*) sur les très grands nombres, ou bien utiliser des librairies spécialisées telles GMP [Gra10] ou PARI/GP [CB05] (voir Annexe C pour une courte description de librairies connues).

Il est important de noter que pour un exposant s'étalant sur k bits, l'algorithme 2.1 prendra au maximum $2k$ multiplications et $2k$ extractions de modulos. Outre la méthode binaire, il existe de nombreuses méthodes pour effectuer plus efficacement le calcul de $a^w \bmod n$ (en utilisant moins de $2k$ multiplications et extractions de modulos), par exemple les méthodes quaternaires, octales, maires et adaptatives. Pour une couverture intéressante de ces méthodes, voir [Knu98] et [Koc94]. On peut aussi utiliser la méthode de Montgomery, qui ne demande que deux extractions de modulo au total. On note que pour toutes ces méthodes, la quantité de calculs à effectuer est proportionnelle au nombre de bits de l'exposant.

2.3 Nombres premiers

Nous verrons au chapitre 4 que les nombres premiers sont au coeur des méthodes d'encryptage à clés publiques/privées. Dans la suite de cette section, nous explorerons différents algorithmes et propriétés liés aux nombres premiers qui s'avéreront utiles pour comprendre et utiliser les méthodes d'encryptage à clés publiques telles DH, RSA, Elgamal et à courbes elliptiques.

Les nombres premiers forment un sous-ensemble des nombres entiers. Ils ont la caractéristique de n'être divisibles sans reste que par 1 et eux-mêmes. Euclide a prouvé par contradiction qu'il y a une infinité de nombres premiers:

Supposons qu'il y ait un nombre fini de nombres premiers, qui forment l'ensemble $\{p_1, p_2, \ldots, p_k\}$. Construisons le nombre $N = p_1 \times p_2 \times \cdots \times p_k + 1$. Par construction, N ne peut avoir aucun des nombres $\{p_1, p_2, \ldots, p_k\}$ comme facteur. Alors soit N est lui-même un nouveau nombre premier, soit il existe un ensemble d'autres nombres premiers $\{p_x, p_y, \cdots\}$, facteurs de N , qui ne se trouvent pas dans $\{p_1, p_2, \ldots, p_k\}$. Dans les deux cas, l'hypothèse de départ est fausse, et donc il y ne peut y avoir un nombre fini de nombres premiers.

2.3.1 PGCD et algorithme d'Euclide

On peut représenter tout nombre entier positif n par un produit de nombres premiers : $n = p_1^{w_1} \times p_2^{w_2} \times \cdots \times p_k^{w_k}$ avec $\{p_1, p_2, \ldots, p_k\}$ des nombres premiers et $\{w_1, w_2, \ldots, w_k\}$ des nombres entiers positifs, les puissances respectives des nombres premiers. Par exemple : $16 = 2^4$, $32 = 2^5$, $33 = 3^1 \times 11^1$, $36 = 2^2 \times 3^2$. On définit le Plus Grand Commun Diviseur (PGCD) de deux nombres n_1 et n_2 comme étant le produit des facteurs premiers communs $\{p_i, p_j, \ldots, p_l\}$ de n_1 et n_2, où chaque nombre p_i apparaît à la puissance correspondante w_i la plus petite du facteur p_i de chacun des deux nombres n_1 et n_2.

Par exemple, PGCD($n_1 = 32$, $n_2 = 36$) = PGCD(2^5, $2^2 \times 3^2$) = 2^2.

Euclide a été le premier à proposer un algorithme pour trouver le PGCD, algorithme qui est encore utilisé à l'heure actuelle. Soient deux nombres entiers positifs, a et n, dont le PGCD est μ. Ceci implique que $a = s \times \mu$ et $n = t \times \mu$, où s et t sont deux entiers positifs. Supposons que $n \geq a$ (si ce n'est pas le cas, on intervertit a et n). Définissons $\lfloor x \rfloor$ comme le nombre entier qui précède immédiatement ou est égal au nombre réel x (par exemple $\lfloor \pi \rfloor = 3$,$\lfloor 2.0 \rfloor = 2$). Alors:

$$\begin{aligned} r &= n \bmod a \\ &= n - a \times \lfloor n/a \rfloor \\ &= t \times \mu - s \times \mu \times \lfloor t/s \rfloor \\ &= \mu \times (t - s \times \lfloor t/s \rfloor) \end{aligned}$$

De cette relation découlent deux propriétés importantes :

1. r est un multiple de μ, le PGCD de a et n.
2. r étant un résidu modulo a , il est par définition plus petit que a.

L'algorithme d'Euclide utilise ces deux propriétés pour effectuer le calcul du PGCD de façon itérative, avec la garantie que le nombre d'itérations soit fini. Étant donné une paire de nombres (a, n) , on effectue le calcul $r = n \bmod a$, on remplace (a, n) par (r, a), et on recommence, tant que r n'est pas nul. Puisque $r = n \bmod a$ est strictement plus petit que a et que r remplace a à chaque itération, il est assuré qu'éventuellement r sera nul. Lorsque c'est le cas, alors la valeur actuelle de a représente le PGCD de la paire de nombres originale (a, n), puisque tout au long de la séquence de calculs, les résidus r successifs sont toujours des multiples du PGCD(a, n) des nombres initiaux (a, n). Sous forme de pseudo code on obtient l'algorithme 2.2.

```
pgcd (a, n) {
  if (n < a) {  t = n;     n = a;     a = t;  }
  r = n % a;
  while ( r != 0) {
    n = a;     a = r;     r = n % a;
  }
  return (a);
}
```

Algorithme 2.2: Calcul du PGCD de Euclide

2.3.2 Algorithme étendu d'Euclide et inverses multiplicatifs modulo

Soit μ, le PGCD de a et n. Alors il existe deux nombres entiers u et v tels que $n \times u + a \times v = \mu$, avec $0 \leq u < a$ et $0 \leq v < n$. Nous verrons plus loin l'intérêt de cette relation pour trouver l'inverse multiplicatif d'un nombre modulo n. Dans ce qui suit, nous nous concentrerons sur une méthode qui permet de calculer u et v à partir de a et n [Bog05, Knu98].

Pour trouver u et v, remarquons tout d'abord qu'ils représentent les coefficients d'une combinaison linéaire de a et n dont le résultat est μ. Pour calculer u et v, nous procéderons de façon itérative, en nous basant sur l'algorithme d'Euclide. Notre méthode sera de mettre à jour les valeurs relatives des coefficients entrant dans la combinaison linéaire de a et n à mesure que progresse l'algorithme d'Euclide. Nous supposerons que $n \geq a$ et nous débuterons avec les deux combinaisons linéaires suivantes :

$$\begin{array}{rcrcll} n & = & 1 \times n & + & 0 \times a & \text{(I)} \\ a & = & 0 \times n & + & 1 \times a & \text{(II)} \end{array}$$

Nous supposerons que $n = a \times t + r$, où $r = n \bmod a$ et $t \geq 0, t \in \mathbb{N}$. Alors, on peut multiplier l'équation (II) par t, et soustraire de l'équation (I) l'équation obtenue par cette multiplication. Ce faisant, on obtient l'équation (III):

$$\begin{array}{rcrcrrcl} n & = & 1 \times n & + & 0 \times a & 1 \times & \text{(I)} & \\ (-t) \times a & = & (-t) \times 0 \times n & + & (-t) \times 1 \times a & (-t) \times & \text{(II)} & \\ r & = & 1 \times n & + & (-t) \times a & \text{(I)} \; -t \times & \text{(II)} & = \quad \text{(III)} \end{array}$$

On remarque que t représente le nombre de fois (entier) que a, le terme de gauche de l'équation (II), entre dans n, le terme de gauche de l'équation (I). Pour obtenir u et v, il s'agit de continuer et d'effectuer la même opération entre les équations (II) et (III) que celle effectuée entre les équations (I) et (II). On obtiendra ainsi l'équation (IV), qui sera ensuite utilisée de concert avec l'équation (III) pour obtenir l'équation (V), et ainsi de suite jusqu'à ce que les termes de gauches soient divisibles sans reste. Par exemple, prenons $n = 210$ et $a = 66$. On obtient alors la séquence d'équations suivante :

$$\begin{array}{rcrcrll} 210 & = & 1 \times 210 & + & 0 \times 66 & & \text{(I)} \\ 66 & = & 0 \times 210 & + & 1 \times 66 & (t = 210/66 = 3, r = 12) & \text{(II)} \\ 12 & = & 1 \times 210 & + & (-3) \times 66 & (t = 66/12 = 5, r = 6) & \text{(III)} \\ 6 & = & (-5) \times 210 & + & 16 \times 66 & (t = 12/6 = 2, r = 0) & \text{(IV)} \\ 0 & = & 11 \times 210 & + & (-35) \times 66 & & \text{(V)} \end{array}$$

Notons que le terme de gauche de l'équation (IV) (c'est-à-dire 6) représente le PCGD de 210 et 66. L'équation suivante (V) a un terme de gauche nul, ce qui indique la fin des itérations. Les valeurs de u et v sont les coefficients du terme de droite de l'équation (IV). Dans l'exemple précédent, $u = -5$ et $v = 16$.

Nous pouvons maintenant bâtir un algorithme qui effectue ce calcul, en observant comment on obtient les différentes équations successives. Points à remarquer :

- On n'utilise que 2 équations à la fois, celle du haut (h) et celle du bas (b)
- Les termes de gauche varient, celui du bas d'une itération devenant celui du haut de l'itération suivante. Le terme de gauche de l'équation du bas de l'itération suivante est le reste de la division entière des deux termes de gauche précédents
- Dans les termes de droite, seuls les coefficients des deux nombres du départ sont variables.
- Le calcul se termine lorsque le terme de gauche de l'équation du bas est nul. Le PGCD et les valeurs de u et v sont alors tirés de l'équation du haut.

Définissons h_g et b_g les termes de gauche de l'équation du haut et de l'équation du bas. Définissons aussi h_u, h_v, b_u, b_v les coefficients des deux nombres du

départ qui apparaissent dans les termes de droite des équations du haut et du bas. Finalement, définissons t, le résultat de la division entière de h_g par b_g. On obtient alors l'algorithme 2.3, qui calcule le PGCD de deux nombres a et n, et qui au même moment calcule u et v (noter que trois variables temporaires, tg, tu et tv, doivent aussi être utilisées pour la mise à jour des valeurs de h_g, h_u et h_v).

```
Euclide_etendu(a, n){
  if ( a == 0) {return (ERREUR);}
  h_g = n; h_u = 1; h_v = 0;
  b_g = a; b_u = 0; b_v = 1;
  while ( b_g != 0) {
    t = h_g / b_g ;   // Division entiere
    t_g = b_g ; b_g = h_g - t * b_g ; h_g = t_g;
    t_u = b_u ; b_u = h_u - t * b_u ; h_u = t_u;
    t_v = b_v ; b_v = h_v - t * b_v ; h_v = t_v;
  }
  return (PGCD = h_g , U = h_u , V = h_v);
}
```

Algorithme 2.3: Calcul étendu du PGCD de Euclide

Maintenant que nous savons comment effectuer l'algorithme étendu d'Euclide, voyons en quoi il peut nous être utile. Dans ce qui précède, nous avons utilisé des nombres a et n arbitraires. Que se passe-t-il si les deux nombres sont relativement premiers ? Dans ce cas, le PGCD de a et n est nécessairement 1 (les deux nombres n'ont pas de facteur commun). L'algorithme étendu d'Euclide produira alors les valeurs u et v telles que $u \times n + v \times a = 1$. Si on effectue le modulo (base n) de chaque côté de cette équation, on obtient: $(v \times a) \bmod n = 1$, ou si l'on veut, $v \times a \equiv 1 \pmod{n}$. On reconnaît ici que a et v sont des inverses multiplicatifs modulo n, tel que présenté à la section 2.4. Par exemple, calculons l'inverse multiplicatif de 45, modulo 56:

$$
\begin{array}{rcrcrll}
56 & = & 1 \times 56 & + & 0 \times 45 & & \text{(I)} \\
45 & = & 0 \times 56 & + & 1 \times 45 & (t = 56/45 = 1, r = 11) & \text{(II)} \\
11 & = & 1 \times 56 & + & (-1) \times 45 & (t = 45/11 = 4, r = 1) & \text{(III)} \\
1 & = & (-4) \times 56 & + & 5 \times 45 & (t = 11/1 = 11, r = 0) & \text{(IV)} \\
0 & = & 45 \times 56 & + & (-56) \times 45 & & \text{(V)}
\end{array}
$$

L'équation (IV) nous donne :

PGCD $= u \times 56 + v \times 45$, c'est-à-dire $1 = (-4) \times 56 + (5) \times 45$.

On obtient donc que $v = 5$ est l'inverse multiplicatif de 45 mod 56. On peut vérifier que $5 \times 45 = 225 = 224 + 1 = 4 \times 56 + 1$.

Il est important de noter que cet algorithme peut produire un coefficient v négatif. On doit alors y appliquer l'opération modulo (ou y additionner la base,

ce qui est équivalent dans ce cas). Par exemple, nous avons vu dans la section 2.2 que 5 et 3 sont des inverses multiplicatifs modulo 7. En partant de 3 et 7, on fait le calcul suivant:

$$\begin{array}{rcrcrll} 7 & = & 1\times 7 & + & 0\times 3 & & \text{(I)} \\ 3 & = & 0\times 7 & + & 1\times 3 & (t=7/3=2, r=1) & \text{(II)} \\ 1 & = & 1\times 7 & + & (-2)\times 3 & (t=3/1=3, r=0) & \text{(III)} \\ 0 & = & (-3)\times 7 & + & 7\times 3 & & \text{(IV)} \end{array}$$

L'équation (III) nous donne :

PGCD $= u \times 7 + v \times 3$, c'est-à-dire $1 = 1 \times 7 + (-2) \times 3$

Ici $v = -2$ est l'inverse multiplicatif de 3 (mod 7). Or $(-2) \bmod 7 = 7 - 2 = 5$. Dans Z_7^*, l'inverse multiplicatif de 3 est donc 5, et on retrouve le résultat présenté à la section 2.2.

On remarque que lorsqu'on recherche l'inverse multiplicatif d'un nombre a pour une certaine base n, la valeur du coefficient u est superflue. On peut ainsi simplifier l'algorithme étendu d'Euclide et obtenir l'algorithme 2.4.

```
inverse_multiplicatif(a, n){
  if ( a == 0) {return (ERREUR);}
  h_g = n; h_v = 0;
  b_g = a; b_v = 1;
  while ( b_g != 0) {
    t = h_g / b_g ;   //Division entiere
    t_g = b_g ; b_g = h_g - t * b_g ; h_g = t_g;
    t_v = b_v ; b_v = h_v - t * b_v ; h_v = t_v;
  }
  return (PGCD = h_g , V = h_v);
}
```

Algorithme 2.4: Calcul de l'inverse multiplicatif modulo la base

2.3.3 Théorème de Fermat

Pierre de Fermat fut avocat et magistrat à Toulouse au 17^{e} siècle et s'est distingué par sa grande facilité à identifier et solutionner (en partie) des problèmes mathématiques complexes. Il a été en contact (principalement épistolaire) avec de nombreux mathématiciens et scientifiques de son temps (entre autres Pascal, Descartes, Huygens, Mersenne, Carcavi) [Wei83]. On connaît le dernier théorème de Fermat, qui indique qu'il n'y a pas de solution entière à l'équation $a^n = b^n + c^n$ pour $n > 2$, et qui n'a été définitivement prouvé qu'en 1994

[Wil95, TW95]. Le théorème dont il est question ici, qu'on appelle aussi le "petit" théorème de Fermat, se rapporte à une propriété des résidus d'un nombre premier [Dic05]: Soit p, un nombre premier, et a, un nombre entier positif quelconque. Alors:

$$a^p \equiv a \pmod{p}$$

Ce théorème a été prouvé pour $a = 2$ par Fermat, preuve généralisée à tous les nombres entiers par Euler en 1742. Dans ce qui suit, nous utiliserons la preuve d'Euler, qui procède par induction :

1. Le théorème est trivialement vrai pour $a = 0$ (puisque $0^p = 0 \equiv 0 \pmod{p}$)

2. En supposant le théorème vrai pour a^p, il doit être vrai pour $(a+1)^p$. Or :

$$(a+1)^p = a^p + \binom{p}{1}a^{p-1} + \cdots + \binom{p}{p-1}a + 1 \qquad \text{(Binôme de Newton)},$$

   où $\binom{n}{m} = \frac{n!}{m!(n-m)!}$ est le nombre de combinaisons de n objets pris m à la fois (voir Annexe B). Puisque p est un nombre premier, alors tous les facteurs $\binom{p}{1} \cdots \binom{p}{p-1}$ sont des multiples de p (p n'apparaît pas au dénominateur, et étant premier, p ne peut être simplifié par d'autres nombres au dénominateur: aucun de ces nombres n'est un multiple de p). Ces facteurs ont tous des résidus nuls modulo p. D'où : $(a+1)^p \equiv (a^p + 1) \pmod{p}$. Le théorème étant vrai pour a^p, on obtient finalement: $(a+1)^p \equiv a^p \pmod{p} + 1 \pmod{p} \equiv (a+1) \pmod{p}$

3. Puisque le théorème est vrai pour $a = 0$, et qu'il est vrai pour $(a+1)$ lorsqu'il est vrai pour a , alors il vrai pour tous les a entiers.

Lorsque a n'est pas un multiple de p, on peut obtenir une formule simplifiée :

$$a^{(p-1)} \equiv 1 \pmod{p}$$

Quoique cette dernière relation puisse être dérivée de la preuve d'Euler, on peut aussi la prouver à l'aide d'un raisonnement tout à fait différent: considérons que pour tout nombre premier p, tous les éléments de Z_p^* génèrent Z_p^*. Établissons tout d'abord le produit de tous les éléments de Z_p^*:

$$\Pi = 1 \times 2 \times 3 \times \cdots \times (p-1)$$

Ensuite, prenons chaque élément de Z_p^*, et multiplions-le par a. Faisons maintenant le produitΠ_a de tous ces nombres :

$$\Pi_a = (a) \times (2a) \times (3a) \times \cdots \times ((p-1)a) = \Pi \times a^{(p-1)}$$

Supposons tout d'abord que a est un élément de Z_p^* (c'est-à-dire, $0 < a \leq p-1$). Alors nous savons que a génère Z_p^*, donc que :

$$\{a \bmod p, 2a \bmod p, 3a \bmod p, \ldots, (p-1)a \bmod p\} \equiv \{1, 2, 3, \ldots, p-1\} = Z_p^*$$

(l'ordre des éléments ne sera pas le même, mais ils seront tous présents et formeront Z_p^*). Alors en appliquant le modulo p aux équations qui définissent Π et Π_a, on obtiendra :

$$\Pi_a \bmod p = \Pi \bmod p$$

Or $\Pi_a = \Pi \times a^{(p-1)}$, d'où :

$$\begin{array}{lcl} \Pi \times a^{(p-1)} \bmod p & = & \Pi \bmod p \\ (\Pi \bmod p \times a^{(p-1)} \bmod p) \bmod p & = & \Pi \bmod p \\ a^{(p-1)} \bmod p & = & 1 \end{array}$$

Cette forme du petit théorème de Fermat est vraie si a est un élément de Z_p^*. Qu'en est-il si a est plus grand que p (sans pour autant être un multiple de p)? Dans ce cas, on peut toujours exprimer a sous la forme : $a = k \times p + r$, où r est le résidu de a modulo p. Lors de la multiplication (modulo p) de a par un élément $z_i \in Z_p^*$, on obtiendra alors:

$$\begin{array}{rcl} a \times z_i \bmod p & \equiv & ((k \times p) + r) \times z_i \pmod p \\ & \equiv & ((k \times p \times z_i) \bmod p + r \times z_i \bmod p) \bmod p \\ & \equiv & r \times z_i \pmod p \end{array}$$

puisque $(k \times p \times z_i) \bmod p$ est nul ($k \times p \times z_i$ est un multiple de p, et son résidu est nul). En conséquence, la preuve du petit théorème de Fermat est encore valable, car la multiplication par $a > p$ est équivalente à une multiplication par $r \in Z_p^*$, où $r \equiv a \pmod p$.

Plus loin dans ce document, pour distinguer les deux formes du petit théorème de Fermat, on indiquera la forme " originale " du théorème de Fermat lorsqu'on mentionnera :

$$a^p \equiv a \pmod p$$

et on indiquera la forme " simplifiée " du théorème de Fermat lorsqu'on mentionnera :

$$a^{(p-1)} \equiv 1 \pmod{p}$$

Il est important de se rappeler que dans sa forme originale, le petit théorème de Fermat n'impose aucune condition sur a autre qu'il soit entier et positif. Par contre, dans sa forme simplifiée, le petit théorème de Fermat suppose que a est relativement premier avec le nombre premier p.

Nous verrons plus loin que le petit théorème de Fermat est à la base des algorithmes d'encryptage DH et RSA. De plus, puisque ce théorème s'applique à tous les nombres premiers sans exception, il peut être utilisé pour vérifier si un nombre est effectivement premier. En effet, pour l'utilisation sécuritaire de DH et RSA, nous verrons qu'il est important d'utiliser de très grands nombres premiers. Les questions suivantes se posent alors : Comment produire de très grands nombres premiers, et comment vérifier que les très grands nombres qu'on produit sont vraiment premiers ? Le petit théorème de Fermat peut nous aider à répondre à la deuxième question. En effet, si on produit un nombre premier p, le théorème est vrai pour tous les a. Si au contraire p n'est pas un nombre premier, alors on peut supposer que la relation sera fausse pour certaines valeurs de a.

Ainsi, pour vérifier la primalité d'un nombre n, on pourrait effectuer le calcul

$$a^n \bmod n$$

pour plusieurs valeurs de a, et s'assurer que le résultat est bien a dans chaque cas. En effectuant un nombre suffisant d'essais fructueux, on aurait alors une grande confiance que le nombre est effectivement premier. Pour utiliser efficacement cette façon de faire, il serait préférable d'utiliser des nombres a qui n'ont pas de facteurs communs entre eux, de sorte que chaque nouveau calcul $a^n \bmod n$ apporte une réponse indépendante des autres. Ainsi, la meilleure méthode serait d'utiliser pour a une série de nombres premiers, qui par définition n'ont pas de facteurs communs. Soit $P_x = \{2, 3, 5, 7, 11, 13, \ldots, K\}$, qui contient les x nombres premiers les plus petits ($P_1 = \{2\}$, $P_2 = \{2, 3\}$, $P_3 = \{2, 3, 5\}$, etc.). Alors le système pourrait par exemple utiliser tous les éléments de $P_{64} = \{2, 3, 5, 7, \cdots, 307, 311\}$ et vérifier à tour de rôle que

$$a^n \equiv a \pmod{n},\ \forall a \in P_{64}$$

Si la vérification réussit pour tous les nombres de P_{64}, on pourrait supposer que le nombre n est fort probablement premier. Nous verrons à la section 2.3.6 que cette méthode fonctionne en général mais qu'elle n'est malheureusement pas à toutes épreuves.

2.3.4 Théorème de l'indicatrice d'Euler

Au 18^{e} siècle, Euler a établi une extension intéressante au petit théorème de Fermat. Nous allons maintenant nous pencher sur ce théorème, qui est essentiel au fonctionnement de RSA. Avant de commencer, nous devons tout d'abord introduire la fonction indicatrice d'Euler, $\phi(n)$ (appelée "totient function" ou "phi function" dans les publications en anglais): $\phi(n)$ représente le nombre d'entiers positifs plus petits que n et relativement premiers avec celui-ci.

Par exemple, déterminons $\phi(23)$ et $\phi(22)$. Puisque 23 est un nombre premier, alors $\phi(23) = 22$ (tous les nombres entiers positifs plus petits que 23 sont relativement premiers avec celui-ci). Pour ce qui est de $\phi(22)$, pour le calculer on doit prendre tous les nombres qui sont inférieurs à 22 et retirer les nombres qui ont des facteurs communs avec 22:

$1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21$		(21 nombres plus petits que 22)
dont on enlève :	$2, 4, 6, 8, 10, 12, 14, 16, 18, 20$	(10 multiples de 2 plus petits que 22)
et	11	(1 multiple de 11 plus petit que 22)

D'où : $\phi(22) = 21 - 10 - 1 = 10$

Si $n = p \times q$, où p et q sont deux nombres premiers, alors $\phi(n)$ se calcule en utilisant la formule suivante:

$$\begin{aligned} \phi(n) &= \phi(p \times q) \\ &= \phi(p) \times \phi(q) \\ &= (p-1) \times (q-1) \end{aligned}$$

Pour se convaincre que cette formule est valable, il suffit de réaliser que l'ensemble des nombres positifs plus petits que n est $\{1, 2, \cdots, (pq-1)\}$, et que parmi ceux-ci, les seuls nombres qui ne sont pas relativement premiers avec n sont les multiples de p et q: $\{p, 2p, \cdots, (q-1)p\}$ et $\{q, 2q, \cdots, (p-1)q\}$ [Sta03].

On obtient donc:

$$\begin{aligned} \phi(n) &= (pq-1) - (q-1) - (p-1) \\ &= pq - (p+q) + 1 \\ &= (p-1) \times (q-1) \end{aligned}$$

Par exemple,

$$\begin{aligned} \phi(22) &= \phi(2 \times 11) \\ &= \phi(2) \times \phi(11) \\ &= 1 \times 10 \\ &= 10, \end{aligned}$$

résultat qu'on avait obtenu auparavant par énumération.

On peut généraliser le calcul de $\phi(n)$ pour des nombres composés ayant un nombre arbitraire de facteurs premiers. Nous n'en ferons pas la preuve ici, mais la forme générale de l'indicatrice d'Euler est [Yan04, CP05]:

$$\phi(n) = n \times \prod_{p|n} \left(\frac{p-1}{p} \right),$$

où $p|n$ indique l'ensemble des nombres premiers p qui divisent n.

Par exemple, on peut vérifier que

$$\begin{aligned} 60 &= 2^2 \times 3^1 \times 5^1 \\ \phi(60) &= 60 \times \tfrac{2-1}{2} \times \tfrac{3-1}{3} \times \tfrac{5-1}{5} \\ &= 60 \times \tfrac{1}{2} \times \tfrac{2}{3} \times \tfrac{4}{5} \\ &= 16 \end{aligned}$$

Il y a effectivement 16 nombres compris entre 0 et 60 qui y sont relativement premiers. Ces nombres sont: $1, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 49, 53$ et 59.

Euler a démontré que la fonction indicatrice $\phi(n)$ permet de généraliser le petit théorème de Fermat. Ainsi, si a et n sont relativement premiers, alors :

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

Preuve : Lorsque n est premier, $\phi(n) = n-1$, et alors on retrouve exactement la forme simplifiée du petit théorème de Fermat. Cependant, si n n'est pas premier, on peut prouver que la relation est encore vraie en utilisant un raisonnement analogue à celui employé pour la preuve du théorème de Fermat simplifié. Nous savons que $\phi(n)$ représente l'ensemble des nombres entiers positifs plus petits que n et relativement premiers avec n. Représentons l'ensemble de tous ces nombres par:

$$R = \{r_1, r_2, \cdots, r_{\phi(n)}\}$$

Multiplions maintenant chacun des éléments r_i de R par a (un nombre entier qui est relativement premier avec n), et pour chaque produit calculons le résidu modulo n. On obtient alors le nouvel ensemble:

$$S = \{a \times r_1 \bmod n, a \times r_2 \bmod n, \cdots, a \times r_{\phi(n)} \bmod n\}$$

Or, les deux ensembles contiennent exactement les mêmes éléments (possiblement permutés): puisque a et les éléments de R sont relativement premiers

avec n, alors tous les produits $a \times r_i$ sont aussi relativement premiers avec n, de même que leurs résidus $a \times r_i \bmod n$. De plus, tous les résidus $a \times r_i \bmod n$ sont différents (si ce n'était pas le cas, cela impliquerait qu'il existe deux nombres r_i et r_j qui sont identiques, ce qui va à l'encontre de notre définition de R). Comme pour la preuve du petit théorème de Fermat, il s'agit ensuite d'effectuer le produit (modulo n) de tous les éléments de R, et de réaliser que ce produit est identique au produit (modulo n) de tous les éléments de S:

Soient Π_R et Π_S ces deux produits.

Alors:	$\Pi_S \bmod n$	$=$	$\Pi_R \bmod n.$
Or,	Π_S	$=$	$\Pi_R \times a^{\phi(n)},$
d'où:	$\Pi_R \times a^{\phi(n)} \bmod n$	$=$	$\Pi_R \bmod n$
	$a^{\phi(n)} \bmod n$	$=$	1

L'équation qui précède est la forme courante du théorème de l'indicatrice d'Euler. Si on élève à une puissance entière et positive arbitraire k un côté et l'autre de la relation, on obtiendra le même résultat (puisque $1^k = 1$, alors $a^{k\times\phi(n)} \bmod n = 1^k = 1$). Ensuite, si on multiplie le résultat par a de part et d'autre de la relation, on obtient une autre forme du théorème de l'indicatrice d'Euler qui s'avérera utile à la section 4.3.1:

$$a^{k\times\phi(n)+1} \equiv a \pmod{n}$$

2.3.5 Inverse multiplicatif - autre méthode de calcul

Nous avons vu à la section 2.3.2 qu'on peut utiliser l'algorithme étendu d'Euclide pour calculer l'inverse multiplicatif d'un nombre modulo un autre nombre avec lequel il est relativement premier. Les théorèmes de Fermat et de l'indicatrice d'Euler permettent une méthode complètement différente de trouver cet inverse multiplicatif. En effet, selon le théorème de l'indicatrice d'Euler:

Nous savons que:	$a^{\phi(n)}$	$\equiv$	$1 \pmod{n}$
Ceci implique la relation suivante:	$a \times a^{\phi(n)-1}$	$\equiv$	$1 \pmod{n}$
Puisque par définition,	$a \times a^{-1}$	$\equiv$	$1 \pmod{n}$
L'inverse multiplicatif peut se calculer ainsi:	a^{-1}	$\equiv$	$a^{\phi(n)-1} \pmod{n}$

Lorsqu'on connaît l'indicatrice d'Euler du nombre n, on peut donc trouver l'inverse multiplicatif $a^{-1} \pmod{n}$ d'un nombre a, en utilisant par exemple l'algorithme d'élévation à une puissance modulo une base (voir algorithme 2.1). Puisque l'indicatrice d'Euler du nombre n est essentielle pour calculer l'inverse multiplicatif à l'aide de cette méthode, elle n'est utilisable que lorsqu'on connait les facteurs premiers de n. Autrement, la méthode de la section 2.3.2 est utilisée.

2.3.6 Nombres de Carmichael

La méthode de vérification des nombres premiers qui utilise le petit théorème de Fermat ébauchée à la section 2.3.3 est valide s'il est vrai que les nombres composés (non premiers) vont toujours échouer le test pour certaines valeurs de a. Pour la grande majorité des nombres composés, c'est effectivement le cas. Malheureusement, il existe une classe de nombres composés, décrits d'un point de vue théorique en 1899 par Korselt [CP05], dont des exemples ont été découverts par R.D. Carmichael en 1910-12 [Car10, Car12] et qui répondent à l'équation de Fermat pour toutes les valeurs possibles de a. On appelle maintenant ces nombres des nombres de Carmichael. Le critère de Korselt établit que ces nombres particuliers ont les caractéristiques suivantes:

1. Aucun de leurs facteurs premiers ne sont des carrés parfaits (c'est-à-dire, $N_C \neq k \times p^2$)
2. Ils comportent au moins trois facteurs premiers ($N_C = p_1 \times p_2 \times p_3 \times \cdots$)
3. Pour chacun des facteurs premiers p de N_C, $p-1$ est un facteur de $N_C - 1$

Par exemple, le nombre $561 = 3 \times 11 \times 17$ et certains nombres de la forme $N = (6k+1)(12k+1)(18k+1)$ ont cette propriété. Ainsi, la recherche d'un nombre a tel que $a^{561} \bmod 561 \neq a$ sera nécessairement vaine, puisque 561 satisfait la version originale du théorème de Fermat pour tous les nombres a qui sont des entiers positifs. Il est donc impossible de distinguer un nombre de Carmichael d'un nombre premier à l'aide de la version originale du théorème de Fermat. Cependant, la version simplifiée du théorème de Fermat ($a^{p-1} \equiv 1 \pmod{p}$), qui exige que a et p soient relativement premiers, pourra détecter des nombres de Carmichael N_C lorsque a est un multiple de l'un des facteurs de N_C. Malheureusement, même cette méthode est incertaine. Par exemple, on peut vérifier que 216 821 881 satisfait l'équation simplifiée de Fermat:

$$a^{p-1} \equiv 1 \pmod{p}, \forall a \in P_{64} = \{2, 3, 5, 7, \cdots, 307, 311\}$$

alors que c'est un nombre composé. En effet:

$$\begin{aligned} 216\,821\,881 &= 331 \times 661 \times 991 \\ &= (6 \times 55 + 1)(12 \times 55 + 1)(18 \times 55 + 1). \end{aligned}$$

Les nombres faisant partie de P_{64} sont tous plus petits que le plus petit des facteurs (331) du nombre 216 821 881. Pour que l'équation simplifiée de Fermat puisse détecter qu'un nombre de Carmichael est composé, il faut utiliser un nombre a qui soit un multiple de l'un des facteurs de N_C. Lorsque n est un nombre de Carmichael, il y a donc $(n - \phi(n))$ nombres pour lesquels le test basé

sur le théorème simplifié de Fermat échouera avec certitude. Pour qu'il soit possible de détecter un nombre de Carmichael, on doit prendre comme valeur a des nombres aléatoires entre 2 et n, pour qu'il soit possible d'obtenir un nombre a qui ne soit pas relativement premier avec n. Dans ce cas, la probabilité de succès est de $(1-\frac{\phi(n)}{n})$, soit dans l'exemple précédent environ 0.5 de 1 pour cent. Or plus le nombre de Carmichael est grand, plus cette probabilité diminue.

L'utilisation du petit théorème simplifié de Fermat n'est donc pas suffisante pour avoir la certitude qu'un nombre est effectivement premier, puisque pour de très grands nombres de Carmichael, il est très peu probable qu'on utilise par hasard un nombre a qui ne passe pas le test basé sur le théorème simplifié de Fermat. Pour conclure cette réflexion sur les nombres de Carmichael, on peut se demander s'il existe de tels nombres parmi les nombres très grands utilisés en cryptographie. On sait depuis 1992 qu'il y a une infinité de nombres de Carmichael [AGP94], et donc qu'il existe des nombres de Carmichael même parmi les très grands nombres. Il faut noter cependant que la densité des nombres de Carmichael est assez faible, le nombre $C(n)$ de nombres de Carmichael plus petits qu'un nombre n étant $n^{2/7} < C(n) < n^{1-\ln\ln\ln n/\ln\ln n}$ [AGP94, EP86]. Par exemple, il a été établi empiriquement qu'il n'y a que 585 355 nombres de Carmichael entre 0 et 10^{17}, soit moins de un millionième de un pour cent des nombres dans cet intervalle [Pin93]. En définitive, on peut tirer comme conclusion que lorsqu'on génère un nombre aléatoire très grand, la probabilité de tomber par hasard sur un nombre de Carmichael est très petite, mais que dans ce cas, il est presque assuré qu'on le croira premier si on ne se base que sur le théorème simplifié de Fermat.

2.3.7 Calcul et vérification des nombres premiers

Les algorithmes d'encryptage à clés publiques que nous allons étudier, tels DH, RSA, Elgamal et les courbes elliptiques, sont basés sur l'utilisation de très grands nombres premiers. Comment fait-on pour les obtenir? En théorie, il existe deux façons de faire: soit on crée un nombre premier dont la primalité est assurée par notre méthode de construction, soit on crée un nombre de façon aléatoire, et on vérifie ensuite si ce nombre est premier ou non.

Depuis Euclide, les mathématiciens ont tenté d'établir une fonction f qui produirait les nombres premiers, de telle sorte que $f(k)$ produise le $k^{\text{ième}}$ nombre premier. Malheureusement, les nombres premiers ont une distribution très irrégulière et aucune fonction f n'a été établie pour les calculer de cette façon. On connaît la fonction $\zeta(s)$ (zêta) de Riemann, qui établit une relation dans le plan complexe (puisque $s = a + bi$) entre une somme infinie de puissances de nombres ordinaux n et un produit infini de fonctions de nombres premiers p:

$$\zeta(s) = \sum_{n=1}^{\infty} n^{-s} = \prod_{p} (1 - p^{-s})^{-1}$$

Cependant, la fonction $\zeta(s)$ ne permet pas d'obtenir une suite structurée de nombres premiers. Toutefois, on sait depuis la fin du 18^{e} siècle (conjecture de Gauss et Legendre, prouvée indépendamment en 1896 par Hadamard et Vallée-Poussin) que la distribution moyenne des nombres premiers est proportionnelle au logarithme naturel de leur taille. C'est-à-dire que dans le voisinage d'un nombre n, il y aura (en moyenne) un nombre premier à tous les $(\ln n)$ nombres.

Ceci ouvre la porte à la mise au point d'une méthode efficace de génération de nombres premiers de grande taille. Supposons qu'on désire obtenir un nombre premier d'une certaine taille, disons un nombre de 600 chiffres décimaux (c'est la taille approximative des nombres de 2 048 bits utilisés pour RSA). On peut générer un nombre aléatoire N de 600 chiffres, et vérifier s'il est premier. S'il ne l'est pas, on vérifie le nombre suivant $(N + 1)$. Si celui-ci n'est pas premier, on continue, et ainsi de suite jusqu'à ce qu'on ait découvert un nombre qui soit premier. Cette recherche, qui est conceptuellement simple à réaliser, pourrait a priori exiger l'examen d'une très grande quantité de nombres. Toutefois, puisque la distribution des nombres premiers est proportionnelle à $(\ln n)$, on sait qu'en moyenne, il devrait y avoir $(\ln n)$ nombres composés entre deux nombres premiers contigus. En supposant qu'une recherche aléatoire produirait un nombre de départ avec une répartition uniforme dans l'intervalle entre deux nombres premiers, en moyenne on devrait faire $(\ln n)/2$ essais avant de trouver le nombre premier suivant le nombre de départ. Dans l'exemple précédent (pour un nombre de 600 chiffres décimaux), il devrait y avoir un nombre premier à tous les $\ln(10^{600}) = 1\,381,6 \approx 1\,400$ nombres. Or de ces 1 400 nombres, il est facile d'éliminer les multiples de 2, 3 et 5, qui se reconnaissent aisément. Ce qui fait qu'en définitive, on ne devra vérifier en moyenne que $\frac{1\,400}{2} \times \frac{\phi(2\times3\times5)}{2\times3\times5} \approx 185$ nombres. Le problème de la génération de nombres premiers revient donc à un problème de *vérification* de nombres premiers : comment s'assure-t-on qu'un nombre est premier ? C'est ce que nous verrons dans les deux sections suivantes.

2.3.7.1 Le crible d'Ératosthène

L'un des premiers algorithmes utilisés pour évaluer si un nombre est premier est attribué à Ératosthène, au 2^{e} siècle avant JC. L'idée derrière le crible d'Ératosthène (en anglais : sieve of Eratosthenes) est la suivante : si un nombre est composé, alors il est un multiple d'au moins un nombre premier plus petit ou égal à sa racine carrée. Alors en commençant par le nombre 2, on marque tous ses multiples plus petits ou égaux au nombre dont on veut établir la primalité. Puis on prend le nombre suivant qui n'est pas encore marqué (c'est-à-dire 3), et on refait l'opération de marquage. On recommence ainsi jusqu'à ce que le nombre n soit marqué (et à ce moment on sait que n n'est pas premier) ou bien jusqu'à ce que le prochain nombre non marqué soit plus grand que la racine carrée de n. Dans ce cas, on sait avec certitude que n est un nombre premier.

Par exemple, prenons $n = 41$. Alors $\sqrt{41} = 6$, et les itérations du crible procèdent comme suit :

$1^{\text{ère}}$ itération	:	2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40
2^{e} itération	:	3 9 15 21 27 33 39
3^{e} itération	:	5 25 35

Puisque 6 est déjà marqué, on arrête après la troisième itération, sachant avec certitude que $n = 41$ est un nombre premier.

Autre exemple : $n = 65$. Alors $\sqrt{65} = 8$, et on obtient :

$1^{\text{ère}}$ itération	:	2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64
2^{e} itération	:	3 9 15 21 27 33 39 45 51 57 63
3^{e} itération	:	5 25 35 55 65

Et on s'aperçoit que 65 est un nombre composé, multiple de 5.

L'algorithme d'Ératosthène est utile pour établir la primalité de nombres relativement petits. En effet, il énumère une quantité de nombres qui est de l'ordre de grandeur du nombre à vérifier, ce qui devient impossible à faire pour de très grands nombres. En pratique, on utilise le crible d'Ératosthène pour trouver les nombres premiers plus petits que 1 000 ou 2 000, mais on utilise des méthodes plus efficaces pour les nombres plus grands.

```
Eratosthene(max) {
  for (i = 2; i < max; i = i + 1) { premier[i] = 1; }
  for (i = 2; i ≤ √max ; i = i + 1) {
    if (premier[i] == 1) {
      for (j = 2 * i ; j ≤ max ; j = j + i) {
        premier[j] = 0 ;
      }
    }
  }
  for (i = 2; i < max; i = i + 1) {
    if (premier[i] == 1) {print premier[i];}}
}
```

Algorithme 2.5: Méthode d'Eratosthène

2.3.7.2 Test de Selfridge et Miller-Rabin

Le test qui suit a d'abord été proposé par Artjuhov [Art67, CP05], puis a été redécouvert par Selfridge au début des années 1970 [BLS75, CP05]. Il est basé à la fois sur le petit théorème de Fermat et sur le fait que pour un nombre premier p la racine carrée de 1 modulo p ne peut être que 1 ou -1 (on pourrait

aussi dire 1 ou $p-1$, puisque $-1 \equiv (p-1) \pmod{p}$). L'adaptation de ce test de primalité pour en faire un algorithme utilisable est l'oeuvre de Miller et Rabin [Mil75, Rab80], à qui beaucoup d'auteurs attribuent la paternité entière de cette méthode. L'apport important de Miller-Rabin a été de déterminer les probabilités que ce test donne de faux résultats et de proposer une méthode pour diminuer les possibilités d'erreur. Dans ce qui suit, nous parlerons d'abord du test de Selfridge, puis nous enchaînerons avec l'évaluation des probabilités d'erreur de ce test, concluant avec un algorithme qui les réduit autant qu'on le désire.

Soit un nombre n dont on veut vérifier la primalité. On effectue le test de la façon suivante. Puisque n est potentiellement premier, c'est nécessairement un nombre impair (on ignore le cas trivial $n = 2$). Donc $(n-1)$ est un nombre pair, et on peut l'exprimer sous forme d'un produit d'un nombre impair par une puissance de 2: $(n-1) = 2^R \times T$ où T est un nombre impair, et $R \geq 1$ est un nombre entier positif. On prend ensuite un nombre entier positif a et on calcule la séquence suivante de puissances de a modulo n:

$$\begin{aligned} a^T \bmod n \\ a^{2T} \bmod n \\ a^{4T} \bmod n \\ \dots \\ a^{2^R T} \bmod n \end{aligned}$$

Par définition, l'exposant du dernier terme, $2^R \times T = (n-1)$. Si n est effectivement un nombre premier, alors le théorème de Fermat s'applique et $a^{2^R T} \bmod n = a^{(n-1)} \bmod n = 1$. Si ce n'est pas le cas, alors n n'est pas premier. Si le théorème de Fermat s'applique, alors soit la séquence ne sera composée que d'une suite de 1, soit la séquence ne contiendra que des 1 à partir d'une certaine puissance $2^i \times T$ de a. Si la séquence ne comporte que des 1, alors la seule conclusion est que n est possiblement premier. Sinon, on observe le résidu $a^{2^{i-1} \times T} \bmod n$. C'est le résidu de l'élément de la séquence qui précède l'élément qui débute la série de 1. Ce résidu est donc la racine carrée du nombre 1 qui débute la séquence. Puisque la racine carrée de 1 modulo un nombre premier ne peut qu'être 1 ou -1, alors le résidu qui précède la séquence de 1 doit nécessairement être $(n-1)$ (puisque ce n'est pas 1). Si ce nombre n'est pas $(n-1)$, alors n n'est pas premier. Sinon, on vient d'établir que n est possiblement un nombre premier.

Prenons par exemple $n = 73$, et utilisons $a = 46$. Alors $(n-1) = 72 = 2^3 \times 9$. On obtient la séquence:

$$\begin{aligned} 46^{1 \times 9} \bmod 73 &= 46 \\ 46^{2 \times 9} \bmod 73 &= 72 \\ 46^{4 \times 9} \bmod 73 &= 1 \\ 46^{8 \times 9} \bmod 73 &= 1 \end{aligned}$$

On s'aperçoit que le résidu qui précède la série de 1 est effectivement 72.

Utilisons le test de Selfridge sur le premier nombre de Carmichael (561) , de nouveau avec $a = 46$. Alors $(n-1) = 560 = 16 \times 35$, et on obtient la séquence:

$$\begin{aligned} 46^{1\times 35} \bmod 561 &= 538 \\ 46^{2\times 35} \bmod 561 &= 529 \\ 46^{4\times 35} \bmod 561 &= 463 \\ 46^{8\times 35} \bmod 561 &= 67 \\ 46^{16\times 35} \bmod 561 &= 1 \end{aligned}$$

Le résidu qui précède la série de 1 est $67 \neq 560$, et donc le nombre 561 n'est pas un nombre premier, malgré le fait qu'il passe le test de Fermat ($46^{560} \bmod 561 = 1$). On remarque que dans le test de Selfridge, ce qui devient important est la vérification de la valeur du résidu qui précède la série de 1. C'est ce que nous retiendrons pour bâtir un algorithme qui utilise cette méthode.

Avant d'en arriver à l'algorithme de vérification de primalité, on doit savoir que Miller et Rabin ont établi théoriquement la probabilité qu'un nombre a produise un résultat positif alors que n n'est pas premier (on appelle ce résultat un faux positif) [Mil75, Rab80]. Ce calcul théorique établit qu'il y a au maximum $(1/4)$ des résidus a qui produisent de faux positifs. Pour améliorer la probabilité de détection de nombres composés, il s'agit donc de refaire la même opération avec plusieurs résidus a différents et de vérifier que le résultat du test soit toujours positif. En appliquant le test de Selfridge k fois successives, chaque fois avec une valeur de a différente, il ne restera qu'une probabilité de $(1/4)^k$ que le nombre n soit en fait un nombre composé. En effectuant un nombre suffisant d'itérations, on peut donc diminuer la probabilité d'un mauvais choix autant qu'on le désire. En pratique, en appliquant le test de Selfridge 64 fois, la méthode de Miller-Rabin s'assure que la probabilité d'une mauvaise décision n'est que de 2^{-128} [FS03]. Dans ce cas, considérant qu'il y a eu environ 2^{88} nanosecondes depuis le Big Bang, il faudrait que 1 000 milliards d'ordinateurs produisant un résultat par nanoseconde aient fait des calculs depuis le début de l'univers pour que la méthode de Miller-Rabin se soit trompée une unique fois. Donc 2^{-128} semble acceptable comme probabilité d'échec de cette méthode de vérification. Nous avons maintenant tous les éléments nécessaires à la construction de l'algorithme 2.6 de vérification de primalité.

```
Miller_Rabin(n) {
  if (n == 2) { return ( TRUE ); }
  if ((n < 3) || (n % 2 == 0)) {
    return ( FALSE ); }                    // n est trop petit ou pair
  w = n-1;      k = 0;
  while (w % 2 == 0) {
    w = w / 2; k = k + 1 ;}                // 2^k est facteur de n-1
  essais = 0 ;
  while (essais < 64) {                    // 64 succes = prob. 2^-128 d'erreur
    a = ⌊2 + (n - 3) * random( )⌋ ;  // 2 ≤ a < n-1
    temp = puissance_mod(a, w,  n) ;// Calcul de a^w (mod n)
    if ( temp != 1) {                      // Serie de 1 ne commence pas
      r = 0 ;                              //    au tout debut
      while (temp != n - 1) {              // Recherche du residu (n-1)
        if ( r == k - 1) {                 // Avant-dernier element
          return(FALSE) ;}
        else {
          temp = puissance_mod( temp, 2, n);  // Mise au carre (mod n)
          r = r + 1 ;
        }
      }
    }
    essais = essais + 1 ;
  }
  return ( TRUE ) ;
}
```

Algorithme 2.6: Vérification de primalité de Miller-Rabin

Note : Le nombre a est choisi aléatoirement dans l'algorithme 2.6, mais il peut aussi être déterminé à l'avance, par exemple en inscrivant une série de nombres dans un tableau. Certaines versions de l'algorithme de Miller-Rabin utilisent ainsi un tableau pré-établi de 64 nombres premiers (calculés au préalable par exemple à l'aide du crible d'Ératosthène) pour déterminer les valeurs successives de a. Dans ce cas, l'ensemble de nombres premiers $P_{64} = \{2, 3, 5, 7, \cdots, 307, 311\}$ est souvent utilisé.

2.3.7.3 Algorithme AKS

Jusqu'en 2002, on ne savait pas s'il était possible de vérifier la primalité d'un nombre en utilisant un algorithme polynomial par rapport à la quantité de chiffres de ce nombre. Ceci a changé avec la publication des résultats de Agrawal, Kayal et Saxena [AKS04]. Ceux-ci ont démontré un algorithme polynomial qui permet d'établir avec certitude la primalité d'un nombre. Pour le moment, la méthode reste théorique, car elle est encore beaucoup plus lente que la méthode de Miller-Rabin qui demeure la méthode de choix dans la pratique. Cependant,

il y a eu de très nombreux développements récents, et il est possible qu'un jour cette méthode ou une méthode dérivée soit utilisée pour vérifier la primalité des grands nombres premiers, de façon exacte plutôt que probabiliste [Gra04, Die04].

2.3.7.4 Découverte de nombres premiers de taille arbitraire

Nous avons maintenant tous les éléments requis pour permettre la découverte de nombres premiers aussi grands qu'on le désire: nous connaissons une méthode de recherche efficace ainsi qu'une méthode de vérification efficace. Pour rechercher les nombres premiers, nous débuterons avec un nombre aléatoire. Nous pouvons éliminer aisément les multiples de 2, 3 et 5 en ne conservant que les nombres relativement premiers avec $30 = 2 \times 3 \times 5$, c'est-à-dire les nombres dont les résidus sont 1, 7, 11, 13, 17, 19, 23 ou 29 modulo 30. Pour ce faire, on vérifie si le nombre aléatoire de départ a un résidu modulo 30 qui correspond à l'un des résidus acceptables. Sinon, on augmente la valeur du nombre aléatoire pour qu'elle corresponde au prochain résidu permis.

Par exemple, soit le nombre aléatoire $A = 92$. Alors $A \bmod 30 = 2$, qui se trouve entre 1 et 7, les deux premiers résidus acceptables. On utilisera donc $A' = A + 7 - A \bmod 30 = 97$ comme nombre de départ, qu'on augmentera par la suite de telle sorte que les résidus des nombres successifs soient toujours acceptables.

L'algorithme 2.7 découvre un nombre premier aléatoire d'une taille donnée.

```
trouver_premier(taille) {
  TAILLE_RESIDUS = 8 ;
  BASE = 30 ;
  residu [TAILLE_RESIDUS] = {1,7,11,13,17,19,23,29};
  for (i = 0;
       i < TAILLE_RESIDUS;
       i++) {                          // delta[] = {6,4,2,4,2,4,6,2}
    delta [i] = (residu [(i+1) % TAILLE_RESIDUS]
                 - residu [i]) % BASE;
  }
  temp = nombre_aleatoire(taille);     // Nombre d'une certaine taille
  temp_residu = temp % BASE ;
  for (i = 0 ;
       (i < TAILLE_RESIDUS) && (residu[i] < temp_residu);
       i++) ;
  if (residu [i % TAILLE_RESIDUS] != temp_residu) {
    temp = temp
    + residu[i % TAILLE_RESIDUS]   // On demarre avec un nombre dont
    - temp_residu;                 // le residu est acceptable
  }
  if (i >= TAILLE_RESIDUS) {
    temp = temp + BASE;
  }
  while ( Miller_Rabin(temp) ! = TRUE) {
    temp = temp + delta [i];  // Utilisation des residus acceptables
    i = (i+1) % TAILLE_RESIDUS;
  }
  return(temp) ;
}
```

Algorithme 2.7: Découverte de nombres premiers

2.4 Théorème du reste chinois

Le "théorème du reste chinois" (TRC) a été exposé pour la première fois dans un livre écrit par le mathématicien Sun Zi, entre le 3^{e} et le 5^{e} siècle après JC [Mar88]. L'intérêt du TRC tient au fait qu'il explique certaines caractéristiques de la méthode RSA, et qu'il permet de réduire environ des trois quarts le temps de calcul requis par la méthode RSA lors du décryptage.

Dans le livre de Sun Zi, on retrouve ce problème[5] :

[5]Texte traduit du chinois tiré de [Mar88]

"Soit des objets en nombre inconnu:
si on les compte par 3*, il en reste* 2*;*
par 5*, il en reste* 3
et par 7*, il en reste* 2*.*
Combien y a-t-il d'objets ?"

La solution proposée par Sun Zi est la suivante :

"Règle :
En comptant par 3*, il en reste* 2*: poser* 140*;*
en comptant par 5*, il en reste* 3*: poser* 63*;*
en comptant par 7*, il en reste* 2*: poser* 30*.*

Faire la somme de ces trois nombres, obtenir 233*.*

Soustraire 210 *de ce total, d'où la réponse.*

En général,
pour chaque unité restante d'un décompte par 3*, poser* 70*;*
pour chaque unité restante d'un décompte par 5*, poser* 21*,*
pour chaque unité restante d'un décompte par 7*, poser* 15*.*

Si (la somme ainsi obtenue) vaut 106 *ou plus, ôter* 105 *pour trouver la réponse."*

En utilisant une notation moderne, voyons pourquoi ce calcul fonctionne. On remarque que les nombres 70, 21 et 15 ont les caractéristiques suivantes :

$$\begin{aligned} 70 &\equiv 1 \pmod 3 \equiv 0 \pmod 5 \equiv 0 \pmod 7 \\ 21 &\equiv 0 \pmod 3 \equiv 1 \pmod 5 \equiv 0 \pmod 7 \\ 15 &\equiv 0 \pmod 3 \equiv 0 \pmod 5 \equiv 1 \pmod 7 \end{aligned}$$

Donc:

$$\begin{aligned} 2 \times 70 + 3 \times 21 + 2 \times 15 &= 233 \\ &\equiv 2 \pmod 3 \\ &\equiv 3 \pmod 5 \\ &\equiv 2 \pmod 7, \end{aligned}$$

ce qui répond aux exigences du problème. Vérifions cette dernière affirmation en considérant le résultat modulo 3. Puisque 21 et 15 ont des résidus nuls modulo 3, les multiples de ces deux nombres ne contribueront pas au résultat final modulo 3. Puisque 70 a un résidu de 1 modulo 3, alors en le multipliant par 2, on obtiendra un résidu de 2 mod 3 . Ainsi, on obtiendra un résidu de 2 mod 3 pour

la somme de $2 \times 70 + j \times 21 + k \times 15$, où j et k sont des entiers arbitraires. Un raisonnement similaire s'applique pour les résidus modulo 5 et 7. Finalement, pour obtenir la plus petite solution, on effectue $233 \pmod{3 \times 5 \times 7} = 23$, puisque tout multiple de $3 \times 5 \times 7$ ne contribue aucun reste modulo 3, 5 et 7.

Avant de généraliser la méthode de Sun Zi, il est intéressant de comprendre comment le nombre 70 a été obtenu (les nombres 21 et 15 sont calculés de façon analogue). D'abord, 70 est un multiple du produit de tous les nombres premiers du problème autres que 3, c'est-à-dire un multiple de 5×7. Ensuite, on a déjà remarqué que $70 = 2 \times 35 \equiv 1 \pmod 3$. Cette dernière égalité nous indique que 2 et 35 sont des inverses multiplicatifs modulo 3. Pour trouver le nombre 70, il faut donc effectuer le calcul suivant: $70 = 2 \times 35 = ((5 \times 7)^{-1} \bmod 3) \times (5 \times 7)$.

Nous pouvons maintenant formaliser la méthode de Sun Zi de la façon suivante: Soit

$$n = \prod_{i=1}^{k} p_i$$

un nombre composé, formé du produit de k nombres premiers $p_1, p_2, \cdots, p_k$, et soit m, un nombre entier positif tel que $0 \leq m < n$. Nous pouvons représenter de façon unique le nombre m par l'ensemble des résidus de m modulo les facteurs premiers de n. Représentons cette équivalence par le symbole $\overset{n}{\longleftrightarrow}$. Alors $m \overset{n}{\longleftrightarrow} (a_1, a_2, \cdots, a_k)$ où $a_i = m \bmod p_i$. Par exemple, dans le problème de Sun Zi, on recherche $m \overset{n}{\longleftrightarrow} (2, 3, 2)$, avec $n = p_1 \times p_2 \times p_3 = 3 \times 5 \times 7 = 105$.

La transformation entre le nombre m et le vecteur de résidus peut s'effectuer dans les deux sens. À partir des résidus obtenus, on calcule m en utilisant l'équation suivante :

$$m = \left(\sum_{i=1}^{k} a_i \times \left(\frac{n}{p_i} \right) \times \left[\left(\frac{n}{p_i} \right)^{-1} \bmod p_i \right] \right) \bmod n$$

Cette équation représente de façon plus formelle les opérations faites pour solutionner le problème de Sun Zi.

On démontre que l'équation précédente est valable en vérifiant qu'on retrouve la série de $a_i = m \bmod p_i$ en remplaçant m par cette équation. Pour un p_i donné, on sait que tous les résidus a_j tels que $j \neq i$ seront multipliés par un multiple de p_i (puisque $\frac{n}{p_j}$ a nécessairement p_i comme facteur), et donc qu'aucun d'entre eux ne contribuera au résidu modulo p_i (tout multiple de la base a un résidu nul). De la somme qui définit m, il ne reste donc que le terme

$$a_i \times \left(\frac{n}{p_i} \right) \times \left[\left(\frac{n}{p_i} \right)^{-1} \bmod p_i \right]$$

Or par construction, le produit des deux termes à droite de a_i est égal à 1 mod p_i . En définitive, on obtient donc : $m \bmod p_i = a_i$, ce que nous voulions démontrer.

L'intérêt du TRC réside dans le fait qu'il permet de simplifier les calculs impliquant des résidus de nombres composés. En effet, soient deux nombres m_1 et m_2, sur lesquels on désire faire une certaine opération arithmétique (+, -, ×) modulo une base n. On peut effectuer les opérations sur les résidus respectifs de m_1 et m_2 modulo les facteurs premiers de n, puis obtenir le résultat recherché en combinant l'ensemble des résidus selon l'équation présentée plus haut. Ce faisant, on effectue les opérations sur des nombres plus petits, ce qui diminue le temps de calcul. Ceci est particulièrement vrai pour la multiplication et la division de très grands nombres dont la taille dépasse celle du CPU.

Comme exemple de l'utilisation du TRC, calculons

$$\begin{array}{rrcl} & m_3 & = & m_1 \times m_2 \bmod n, \\ \text{pour} & m_1 & = & 125, \\ & m_2 & = & 153 \\ \text{et} & n & = & 187 \\ & & = & 11 \times 17. \end{array}$$

On obtient:

$$\begin{array}{rl} & 125 \overset{11\times17}{\longleftrightarrow} (4,6) \\ \text{et} & 153 \overset{11\times17}{\longleftrightarrow} (10,0). \end{array}$$

On peut ensuite faire la multiplication indépendamment sur les résidus modulo 11 et ceux modulo 17:

$(4,6) \times (10,0) = (4 \times 10 \bmod 11, 6 \times 0 \bmod 17)$,

d'où on tire:

$$\begin{array}{rcl} 125 \times 153 \bmod (11 \times 17) & = & (40 \bmod 11, 0 \bmod 17) \\ & = & (7,0). \end{array}$$

On peut ensuite retrouver le nombre initial, en effectuant le calcul:

$7 \times 17 \times (17^{-1} \bmod 11) + 0 \times 11 \times (11^{-1} \bmod 17)$,

d'où $m_3 = 7 \times 17 \times 2 \bmod (11 \times 17) = 51$.

On vérifie que $125 \times 153 = 19125 = 102 \times 11 \times 17 + 51$.

Chapitre

3

Concepts complémentaires de la théorie des nombres

Dans ce chapitre, nous aborderons certains aspects plus complexes de la théorie des nombres qui sont utiles pour la cryptographie. En particulier, nous étudierons les résidus quadratiques, les courbes elliptiques et les méthodes de factorisation et d'extraction du logarithme discret des grands nombres. Il est important de connaître les méthodes de pointe de factorisation et celles d'extraction du logarithme discret, parce que les techniques modernes de cryptographie reposent sur la difficulté de l'un ou l'autre de ces deux problèmes. Connaître les algorithmes de factorisation et d'extraction du logarithme discret permet de bien identifier les limites de sécurité des techniques de cryptographie et de déterminer les paramètres qui les rendent sécuritaires. Par exemple, la capacité des méthodes de factorisation modernes est ce qui détermine la taille minimale sécuritaire des nombres utilisés pour définir les clés RSA. Quant aux courbes elliptiques, elles sont utiles pour effectuer efficacement certaines méthodes de cryptage et de décryptage. Elles sont aussi utiles pour factoriser certains types de nombres.

Nous débuterons ce chapitre par une présentation des résidus quadratiques, qui sont utiles à la fois dans les calculs impliquant des courbes elliptiques et pour certaines méthodes de factorisation.

3.1 Calcul de résidus quadratiques

Les résidus quadratiques (carrés de nombres en présence de l'opérateur modulo) sont importants à plusieurs égards. Ils jouent un rôle central dans les techniques de pointe de factorisation de grands nombres (section 3.3) et sont essentiels pour l'utilisation des courbes elliptiques (chapitre 4, section 4.4). Dans cette section, nous présenterons d'abord un outil mathématique mis au point par Legendre, puis généralisé par Jabobi, qui permet d'identifier les nombres qui sont effec-

tivement des résidus quadratiques. Nous verrons ensuite comment calculer la racine carrée d'un nombre lorsque ce dernier est un résidu quadratique.

3.1.1 Symboles de Legendre et de Jacobi

À compléter...

3.1.2 Calcul de la racine carrée d'un résidu

À compléter...

3.2 Courbes elliptiques

L'origine de l'étude des courbes elliptiques remonte au 18^{e} siècle. À ce moment, l'étude de la longueur d'arcs d'ellipses a mené à des intégrales de la forme:

$$\int \sqrt{f(x)}dx$$

où $f(x)$ est un polynome de degré 3 ou 4. On a appelé ces fonctions des *intégrales elliptiques*. Par la suite leurs inverses, appelées *fonctions elliptiques*, ont été étudiées, entre autres par Niels Abel dans les années 1820. La forme principale des courbes elliptiques utilisées maintenant en cryptographie est due à Karl Weierstrass, au milieu du 19^{e} siècle:

$$y^2 = x^3 + ax + b$$

Dans le domaine des nombres réels, on peut représenter géométriquement les courbes elliptiques par un graphe $y = \pm\sqrt{f(x)}$ où $f(x) = x^3 + ax + b$. Selon la position des maxima et minima locaux de la fonction $f(x)$ relativement à l'axe des x, la courbe elliptique prendra diverses formes. La figure 3.1 présente quatre exemples, où la courbe pâle représente $f(x)$ alors que la courbe foncée indique la courbe elliptique qui lui est associée.

L'une des caractéristiques des courbes elliptiques est qu'à partir de deux points connus sur la courbe, il est possible d'en trouver un troisième (sous certaines conditions). Nous verrons un peu plus loin que c'est une propriété particulièrement intéressante pour leur utilisation en cryptographie.

Au lieu d'utiliser les courbes elliptiques dans le domaine des nombres réels, il est possible de les définir sur un corps de nombres ("number field" en anglais).

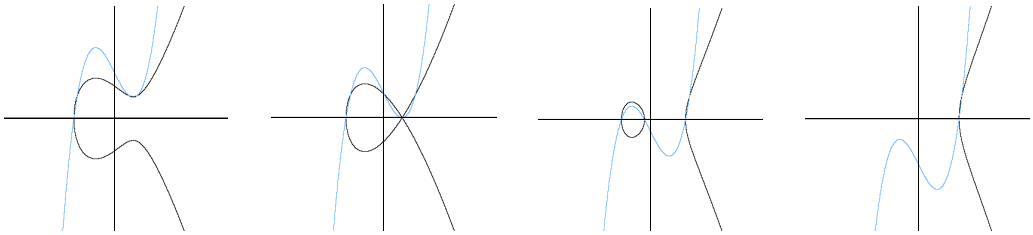


Figure 3.1: Courbes elliptiques - quatre exemples

Par exemple, on peut utiliser l'opérateur modulo et limiter les valeurs acceptées aux entiers positifs plus petits qu'un certain nombre.

[Was08, Kob07] [Mil86, Kob87]*À compléter...*

3.3 Factorisation de très grands nombres

Le problème de factoriser un nombre en ses facteurs premiers intéresse les mathématiciens depuis très longtemps. Déjà, au 17^{e} siècle, Fermat avait identifié une méthode qui permettait de factoriser certains nombres beaucoup plus rapidement que par la division tentative par un ensemble de nombres premiers. Dans cette section, nous étudierons quelques méthodes de factorisation qui peuvent être utilisées pour trouver les facteurs premiers de nombres de taille appréciable. En particulier, nous étudierons deux méthodes dues à Pollard, qui sont utiles lorsque les facteurs premiers d'un grand nombre ont des caractéristiques bien particulières. Nous verrons ensuite la méthode du crible quadratique, ainsi que les méthodes antérieures qui l'ont influencé. Depuis 1991, la méthode de factorisation la plus performante est le crible généralisé de corps de nombres (Generalized Number Field Sieve). Cette méthode, inspirée du crible quadratique, est toutefois trop complexe pour être étudiée ici. Nous la mentionnerons cependant en conclusion de cette section sur la factorisation, parce que c'est elle qui détermine en ce moment la taille des nombres premiers à utiliser pour que la méthode RSA soit sécuritaire.

3.3.1 Méthode ρ de Pollard

En 1975, Pollard a proposé une méthode efficace pour trouver les facteurs premiers d'un nombre, méthode qui a été améliorée par Brent en 1980 [Pol75, Bre80, San05]. Dans ce qui suit, nous étudierons la méthode ρ de Pollard, la plus simple des méthodes de factorisation couvertes dans cette section.

Soit n, un nombre composé (i.e. qui n'est pas premier) et p, l'un des facteurs premiers de n. Nous supposerons que $p \leq \sqrt{n}$ (il est certain qu'il existe toujours au moins un facteur de n qui est plus petit ou égal à sa racine carrée). Prenons maintenant deux nombres entiers arbitraires a et b, compris entre 0 et n. La seule possibilité pour a et b d'être congruents modulo n est que ce soient des nombres identiques. Toutefois, il est possible que ces deux nombres soient congruents modulo p, c'est-à-dire: $a \equiv b \pmod{p}$, même si a et b sont différents. En effet, puisque a et b sont situés entre 0 et n, et que $p \leq \sqrt{n}$, il existe pour un nombre a donné une grande quantité de nombres b qui y seront congruents modulo p: environ $\frac{n}{p} \geq \frac{n}{\sqrt{n}} = \sqrt{n}$ multiples de p se retrouvent entre 0 et n, et donc il y a autant de nombres entre 0 et n dont les résidus modulo p sont identiques. Il faut noter que plus le facteur p est petit, plus il y aura de nombres entre 0 et n qui seront congruents modulo p et plus la probabilité sera grande que deux nombres entiers aléatoires $0 \leq a, b < n$ soient congruents modulo p.

Si on réussit à trouver une paire de nombres a et b tels que $a \equiv b \pmod{p}$, alors $(a - b)$ est un multiple de p, tel que vu précédemment à la section 2.2. Et puisque n est aussi un multiple de p (car $n = p \times q$), alors le plus grand commun diviseur (PGCD) de n et $(a - b)$ sera p. On peut donc déjà imaginer un algorithme utilisant des nombres aléatoires qui nous permettrait de trouver p: on génère deux nombres aléatoires a et b (compris entre 0 et n), et on calcule le PGCD de n et $(a - b)$. Si le PGCD n'est ni 1 ni n, on vient de trouver un facteur p de n. Sinon, on génère un nouveau nombre aléatoire, c, et on calcule le PGCD de n et $(a - c)$, et de n et $(b - c)$. Si l'un de ces PGCD n'est ni 1 ni n, on a trouvé un facteur p de n. Sinon, on génère un nouveau nombre aléatoire, et on continue... Évidemment, à la $k^{\text{ième}}$ itération de notre algorithme, il faudra effectuer k calculs de PGCD. La complexité de notre algorithme est donc quadratique (puisque $1 + 2 + \cdots + k = \frac{k(k-1)}{2}$), et il ne sera utilisable que pour un nombre restreint d'essais.

Heureusement, on peut modifier la façon dont on obtient les nombres aléatoires pour réduire de beaucoup le nombre de calculs de PGCD requis. Pour ce faire, utilisons un polynôme $f(x)$ pour obtenir une séquence de nombres x_k , où $x_{k+1} = f(x_k) \bmod n$. L'utilisation de l'opérateur modulo lors du calcul de la série de nombres x_k nous assure qu'ils seront toujours compris entre 0 et n (ils conserveront une taille raisonnable). Si la séquence de nombres $x_1, x_2, \cdots, x_k, \cdots$ est assez longue, il arrivera un moment où on obtiendra $x_k \equiv x_j \pmod{p}$, où $j < k$. Comment peut-on être sûr que cette situation se produira? Puisque p (le facteur de n qu'on recherche) est un nombre fini, alors il est certain que le $(p+1)^{\text{ième}}$ nombre de la série (x_{p+1}) sera congruent (modulo p) à l'un des nombres qui le précèdent dans la série (l'un de $x_1, x_2, \cdots, x_p$): puisqu'il n'existe que p résidus distincts modulo p, il est clair qu'en générant $(p + 1)$ nombres on en obtiendra deux qui seront congruents. Et avec un peu de chance, on obtiendra deux nombres congruents (modulo p) beaucoup plus rapidement. La méthode ρ de Pollard tient son nom de cette dernière carac-

téristique: la séquence de nombres x_k produira une série de nombres différents, qui à partir d'une certaine valeur bouclera sur elle-même, à l'image de la lettre grecque ρ.

Dès qu'on obtiendra un nombre x_k tel que $x_k \equiv x_j \pmod{p}$, on sait que les nombres qui suivront x_k seront identiques à ceux qui suivent x_j, c'est-à-dire, $x_{k+w} \equiv x_{j+w} \pmod{p}$ où w est un nombre entier positif. On parlera alors d'une boucle, dont la longueur est $\Delta = k - j$. Évidemment, on ne connaît pas p (c'est ce qu'on recherche), et donc nous ne pourrons trouver k, j et Δ directement. Cependant, supposons qu'on continue la séquence de nombres jusqu'à ce que $k + w$ soit un multiple de Δ, puis qu'on prolonge la séquence jusqu'à $x_{2\times(k+w)}$. Puisque $k + w$ est un multiple de Δ, alors $2 \times (k + w)$ sera aussi un multiple de Δ. Ce qui implique que $x_{(k+w)}$ et $x_{2\times(k+w)}$ seront identiques modulo p (entre les deux, on aura fait le tour de la boucle un nombre entier de fois, et on sera revenu au même point). Puisque ces deux nombres sont identiques modulo p, alors leur différence est un multiple de p. Ceci implique que le PGCD de n et de $\left|x_{2\times(k+w)} - x_{(k+w)}\right|$ sera p. À partir de ces observations, on peut maintenant construire un algorithme pour trouver p, un facteur de n. On énumérera une suite de nombres x_k, et chemin faisant nous chercherons le PGCD de n et $|x_k - x_{k/2}|$, pour toutes les valeurs paires de k. Le PGCD sera toujours 1, jusqu'à ce qu'il soit égal au facteur p.

Avant d'aller plus loin, nous étudierons les différentes façons d'énumérer les nombres x_k et de calculer $|x_k - x_{k/2}|$. Une méthode simple est de conserver toutes les valeurs de x_k dans un tableau, et d'effectuer le calcul $|x_k - x_{k/2}|$ chaque fois que l'indice k est pair.

L'algorithme 3.1 présente ce qu'on obtient alors.

```
factorisation( n ) {
  i = 0;
  x[0] = 1;
  while (TRUE) {
    i++;
    x[i] = polynome_f(x[i-1]) % n;
    if((i % 2) == 0) {
      if (pgcd( n , abs(x[i] - x[i/2])) != 1) {
        return( pgcd( n , abs(x[i] - x[i/2])) );
      }
  }
}
```

Algorithme 3.1: Factorisation - première tentative

Cet algorithme fonctionne, mais il souffre d'un défaut majeur: la quantité de mémoire requise égale la taille maximale de l'indice k. Lorsque le nombre à

factoriser n est grand, la quantité de mémoire utilisée par cet algorithme est prohibitive et donc limite la taille des nombres qu'on peut factoriser.

Pour contourner ce problème, Pollard a utilisé l'algorithme de Floyd[BGS72, Fic81], qui permet de détecter des cycles en utilisant une quantité fixe de mémoire. Cependant, le prix à payer est un nombre de calculs trois fois plus important. Dans l'algorithme de Floyd, on utilise deux variables, x et y, qu'on calcule simultanément de la façon suivante : à chaque itération, on calcule $x_{k+1} = f(x_k)$ et $y_{k+1} = f(f(y_k))$. En démarrant l'algorithme avec $x_0 = y_0$, chaque itération produit x_k et $y_k = x_{2k}$.

Cette première version de la méthode de Pollard apparaît à l'algorithme 3.2.

```
Pollard_rho_1( n ) {
  x = 1;  y = 1;
  while (TRUE) {
    x = polynome_f(x) % n;
    y = polynome_f(polynome_f(y) % n) % n;
    if (pgcd( n, abs(x - y)) != 1)  {
      return( pgcd( n, abs(x - y)) );
    }
  }
}
```

Algorithme 3.2: Factorisation ρ de Pollard - première tentative

En analysant l'algorithme initial, Pollard a réalisé que le temps de calcul du PGCD était important, et qu'il était possible de le réduire en multipliant plusieurs nombres avant d'en effectuer le calcul du PGCD. En effet, effectuer le calcul du PGCD est environ $\ln(n)$ fois plus long que de faire une multiplication modulo n [CP05]. Puisque le PGCD de n et des différences $|x - y|$ sera toujours égal à 1 sauf lorsqu'on trouvera le facteur p, multiplier plusieurs différences avant d'effectuer le PGCD ne changera pas le résultat. En supposant qu'on effectue M multiplications avant de faire un calcul de PGCD, on aura alors l'algorithme 3.3 proposé par Pollard.

```
Pollard_rho( n ) {
  x = 1;  y = 1;  z = 1;  m = M ;
  while (TRUE) {
    x = polynome_f(x) % n;
    y = polynome_f(polynome_f(y) % n) % n;
    z = (z * abs(x - y))  % n ;   // Multiplication modulo n
    m = m - 1 ;
    if ( m == 0) {
      if (pgcd(n , z) != 1) {
        return( pgcd(n , z) );
      }
      z = 1;  m = M ;
    }
  }
}
```

Algorithme 3.3: Factorisation ρ de Pollard

En 1980, Brent a proposé une modification de l'algorithme de Pollard qui utilise une autre méthode que celle de Floyd pour détecter les cycles. Cette modification diminue de 24% le temps d'exécution de l'algorithme original. L'étude des détails spécifiques de cette méthode ne sera pas faite ici, mais pour plus d'information on peut consulter l'article de Brent directement sur le réseau [Bre80].

Nous n'avons pas encore parlé du polynôme à utiliser pour générer la suite de nombres pseudo aléatoires. Évidemment, cette fonction doit se calculer assez rapidement, mais nous avons peu d'autres contraintes. Typiquement, on utilisera une fonction de la forme: $f(x) = ax^2 + b$, où b est différent de $(-a \pm 1) \bmod n$, et a est souvent choisi égal à 1. La condition sur b vient du fait que si $b = (-a \pm 1) \bmod n$ et que pour un certain x_k, $f(x_k) = \pm 1$, alors $\forall j \mid j > k, f(x_j) \equiv \pm 1$, puisque $f(\pm 1) \equiv (a \times (\pm 1)^2 - a \pm 1) \bmod n = \pm 1 \bmod n$, et notre fonction génératrice ne donnera plus jamais d'autres nombres que $(\pm 1) \bmod n$.

Il est important de noter que l'algorithme de Pollard requiert de plus en plus d'itérations à mesure que p augmente. On peut prouver que le temps d'exécution de cet algorithme est proportionnel à $\sqrt{p}$, et donc dans le pire des cas il est proportionnel à $\sqrt[4]{n}$. En pratique, sur les systèmes actuels, l'algorithme de Pollard peut factoriser des nombres où $p < 10^{20}$.

Exemple: Factorisation de $n = 73\,537$. Sur la figure 3.2, on observe la séquence de valeurs de $x_k = f(x_{k-1}) \bmod n$ et $y_k = f(f(y_{k-1})) \bmod n$ avec $f(x) = x^2 + 1$. À la 5^{e} itération, on s'aperçoit que le PGCD calculé n'est pas égal à 1. Nous avons alors trouvé l'un des facteurs de n, 151 et par division nous obtenons que $73\,537 = 151 \times 487$.

k	x_k	$y_k = x_{2k}$	$PGCD(n, \lvert x_k - y_k \rvert)$
1	1	2	1
2	2	26	1
3	5	17 108	1
4	26	63 917	1
5	677	39 786	151

Figure 3.2: Itérations de l'algorithme ρ de Pollard

Il est important de remarquer que les paramètres a et b de la fonction $f(x) = ax^2 + b$ ont un impact notable sur le temps requis par l'algorithme pour en arriver à une solution. Ainsi, en utilisant différentes fonctions $f(x) = ax^2 + b$, on s'aperçoit à la figure 3.3 que l'algorithme de Pollard utilise plus ou moins d'itérations pour trouver le résultat.

Fonction $f(x) = ax^2 + b$ utilisée pour factoriser 73 537	Nombre d'itérations
$f(x) = x^2 + 1$	5
$f(x) = x^2 + 2$	13
$f(x) = x^2 + 3$	22
$f(x) = x^2 + 4$	12
$f(x) = x^2 + 32\,767$	1
$f(x) = 2x^2 + 1$	21

Figure 3.3: Efficacité des fonctions utilisées pour l'algorithme ρ de Pollard

La fonction $f(x) = ax^2+b$ qui réduit le nombre d'itérations au minimum dépend des facteurs premiers du nombre n, qui sont évidemment inconnus au départ. Il n'est donc pas possible d'utiliser une fonction $f(x) = ax^2 + b$ optimale pour un certain nombre n à factoriser avant d'en connaître les facteurs...

3.3.2 Méthode $p - 1$ de Pollard

Avant de proposer sa méthode ρ, Pollard avait déjà identifié une autre méthode de factorisation similaire, qui s'applique lorsque les facteurs premiers d'un nombre à factoriser ont des caractéristiques bien particulières [Pol74, Bre89, CP05]. Cette méthode, appelée "$p - 1$" pour des raisons qui seront claires un peu plus loin, est basée sur la forme simplifiée du petit théorème de Fermat. Selon ce dernier, $a^{(p-1)} \equiv 1 \pmod{p}$. Supposons que p est un des facteurs premiers d'un nombre $n = p \times q$ qu'on désire factoriser. Supposons de plus que les facteurs premiers du nombre $p - 1$ sont petits (plus petits qu'une certaine valeur maximum M). Calculons la factorielle de M, $F = M! = 1 \times 2 \times 3 \times \cdots \times M$.

Supposons que $p-1 = p_1^{w_1} \times p_2^{w_2} \times \cdots p_t^{w_t}$, où les p_i sont des nombres premiers, $p_1 < p_2 < \cdots < p_t < M$ et les w_i sont les puissances correspondantes des p_i. Supposons enfin que tous les produits $p_i \times w_i$ sont plus petits que M. Alors F est nécessairement un multiple de $p-1$, et donc $a^F \equiv a^{k(p-1)} \equiv 1 \pmod{p}$. Ceci implique que $a^F - 1$ est un multiple de p. Dans ce cas, le PGCD de $a^F - 1$ et de n sera égal à p.

Le problème avec l'approche précédente est que le nombre $F = M!$ deviendra très grand très rapidement si nous calculons la factorielle standard. Cependant, comme pour la méthode ρ de Pollard, le résidu de F modulo n fait aussi bien l'affaire que F, puisque nous recherchons le PGCD de n et $a^F - 1$. En effet, si le PGCD de n et $a^F - 1$ est p, alors $n = p \times q$ et $a^F - 1 = k \times p$, ce qui veut dire que $(a^F - 1) \bmod n = (k \bmod q) \times p$, qui est toujours un multiple de p.

Par exemple, soit $a^F - 1 = Y = 120$ et $n = 5 \times 11 = 55$, impliquant que le PGCD de Y et n est 5. Mais $Y \bmod n = 10 = 2 \times 5 = (24 \bmod 11) \times 5$, et le PGCD de 10 et 55 est toujours 5, l'un des facteurs premiers de n.

```
Pollard_pm1( n ) {
  m = 2; max = MAXIMUM;
  for (i = 1; i < max; i++) {
    m = puissance_mod( m, i, n);
    if (pgcd( n, m-1) != 1) {
      return( pgcd( n, m-1) );
    }
  }
}
```

Algorithme 3.4: Factorisation $p-1$ de Pollard

Exemple: Factorisation de $n = 3\,131$. Sur la figure 3.4, on observe la séquence de valeurs de m: $m_1 = ((m)^1) \bmod n$, $m_2 = (((m)^1)^2) \bmod n$, ..., $m_5 = ((((((m)^1)^2)^3)^4)^5) \bmod n$. À la 5^{e} itération, on s'aperçoit que le PGCD calculé n'est pas égal à 1. Nous avons alors trouvé l'un des facteurs de n, 31 et par division nous obtenons que $3131 = 31 \times 101$.

Itération	m	$PGCD(n, m-1)$
1	2	1
2	4	1
3	64	1
4	1 318	1
5	2 822	31

Figure 3.4: Itérations de l'algorithme $p-1$ de Pollard

3.3.3 Méthode de factorisation par courbe elliptique de Lenstra

À compléter...

3.3.4 Méthode du crible quadratique

La méthode du crible quadratique (*quadratic sieve* en anglais) a été proposée par Pomerance en 1981 [Pom82, Bre89, Yan04]. Cette méthode a été utilisée en 1994 pour factoriser le nombre n à la base de l'encryptage RSA d'un message publié en 1977 dans le Scientific American[1] , et réputé à l'époque comme étant inviolable pour une période de plusieurs millions d'années... Cette méthode est une amélioration d'un algorithme proposé par Dixon, lui-même basé sur des travaux de Jacobi et Legendre, eux-mêmes utilisés pour faciliter des calculs liés à une méthode de factorisation proposée à l'origine par Fermat. Dans ce qui suit, nous retracerons donc l'évolution de cette méthode à partir de son origine dans les cahiers de Fermat, jusqu'à son expression sous forme d'algorithme exécutable en parallèle sur des ordinateurs modernes.

3.3.4.1 Méthode de Fermat

La méthode de Fermat, dont nous avons brièvement parlé au début de la section 3.3, repose sur la relation bien connue [Bre89]:

$$a^2 - b^2 \quad = \quad (a+b)(a-b)$$

Supposons qu'on veuille factoriser un nombre composé $n = p \times q$, où p et q sont deux nombres premiers[2]. Supposons de plus qu'on puisse trouver deux nombres a et b tels que $n = a^2 - b^2$. Alors:

$$n \quad = \quad a^2 - b^2 \quad = \quad (a+b)(a-b)$$

De cette relation, on tire que deux des facteurs de n sont $(a+b)$ et $(a-b)$. Il faut noter que $a = \frac{(n+1)}{2}$, $b = \frac{(n-1)}{2}$ est toujours une solution de cette relation, mais que dans ce cas $(a+b) = n$ et $(a-b) = 1$. Pour obtenir des facteurs intéressants de n, il faut donc choisir d'autres valeurs pour a et b. Une première façon systématique de rechercher a et b est de débuter avec $a = \lceil \sqrt{n} \rceil$, $b = 0$, puis d'incrémenter la valeur de b jusqu'à ce que $a^2 - b^2$ soit plus petit que n. À ce moment, on incrémente la valeur de a et on recommence le tout jusqu'à ce que $a^2 - b^2$ soit égal à n. Dans l'algorithme qui suit, on utilise $r = a^2 - b^2 - n$

[1] Le message était : "The Magic Words are Squeamish Ossifrage "

[2] Si n est le produit de plus de deux nombres premiers, il suffit de déterminer si l'un ou les deux facteurs obtenus par la méthode sont composés. Si c'est le cas, on effectue de nouveau la factorisation sur les facteurs composés, jusqu'à ce qu'on n'ait plus que des nombres premiers comme facteurs.

comme variable intermédiaire pour ne pas avoir à répéter les calculs lors des différentes comparaisons:

```
Fermat_preliminaire(n) {
  a = ⌈√n⌉ ;
  b = 0;
  r = a² − b² − n;
  while (r ≠ 0) {
    if (r > 0 ) b = b + 1;
    else a = a + 1;
    r = a² − b² − n;
  }
  facteur_1 = a + b;
  facteur_2 = a − b;
}
```

Algorithme 3.5: Méthode de factorisation de Fermat: version préliminaire

Il est possible d'accélérer l'algorithme 3.5 en remarquant qu'on peut mettre à jour directement la valeur de r sans avoir à effectuer le calcul $r = a^2 + b^2 - n$ à chaque itération. À cet effet, calculons ce que devient r lorsqu'on ajoute 1 à a, et lorsqu'on ajoute 1 à b. Soit r' la nouvelle valeur de r lorsque a augmente de 1, et r'' la nouvelle valeur de r lorsque b augmente de 1. Alors:

$$\begin{aligned} r' &= (a+1)^2 - b^2 - n \\ &= (a^2 - b^2 - n) + 2a + 1 \\ &= r + 2a + 1 \end{aligned}$$

$$\begin{aligned} r'' &= a^2 - (b+1)^2 - n \\ &= (a^2 - b^2 - n) - 2b - 1 \\ &= r - 2b - 1 \end{aligned}$$

Posons $u = 2a + 1$ et $v = 2b + 1$. Alors, augmenter a de 1 est équivalent à calculer

$$\begin{aligned} & r' = r + u \\ \text{et} \quad & u' = u + 2 \end{aligned}$$

De même, augmenter b de 1 est équivalent à calculer

$$\begin{aligned} & r'' = r - v \\ \text{et} \quad & v' = v + 2 \end{aligned}$$

L'intérêt de procéder de cette façon est qu'aucune multiplication n'est nécessaire dans les itérations, accélérant ainsi les calculs. À la fin, lorsque r est nul, on peut retrouver les facteurs de n directement à partir de u et v: puisque les facteurs de n sont $(a + b)$ et $(a - b)$ et que $u = (2a + 1)$, $v = (2b + 1)$, alors les facteurs de n sont $\frac{u+v-2}{2}$ et $\frac{u-v}{2}$. L'algorithme de Fermat devient ainsi:

```
Factorisation_Fermat(n) {
  a = ⌈√n⌉ ;
  r = a^2 − n;
  u = 2 × a + 1;
  v = 1;
  while (r ≠ 0) {
    if (r > 0 ) {
      r = r − v;
      v = v + 2;
    } else {
      r = r + u;
      u = u + 2;
  }
  facteur_1 = (u + v − 2)/2;
  facteur_2 = (u − v)/2;
}
```

Algorithme 3.6: Méthode de factorisation de Fermat: version finale

La méthode de Fermat est intéressante, entre autres parce qu'elle n'exige que des additions et des soustractions dans les itérations, qui s'exécutent donc très rapidement. Cependant, un nombre très élevé d'itérations doit typiquement être calculé avant de parvenir à la solution, rendant la méthode peu efficace pour factoriser des nombres très grands. Cette méthode est donc peu utilisée dans la pratique. Nous verrons cependant dans ce qui suit comment elle a influencé les algorithmes plus performants qui l'ont suivi.

3.3.4.2 Méthode de Kraitchik et de Lehmer

Maurice Kraitchik, un mathématicien belge d'origine russe, a proposé en 1922 une modification intéressante de la méthode de Fermat [Kra22, Mat24, Pom96]:

3.3.4.3 Algorithme de Dixon

[Dix81]

3.3.4.4 Algorithme de Pomerance

3.3.4.4.1 Nombres B-smooth et choix de B

3.3.4.4.2 Crible quadratique

3.3.4.4.3 Élimination de Gauss

3.3.5 Autres méthodes de factorisation

[LL93]

La méthode ρ de Pollard vue à la section 3.3.1 fonctionne bien pour factoriser des nombres dont le plus petit facteur $p < 10^{20}$. Quant à la méthode $p-1$ de Pollard vue à la section 3.3.2, elle ne fonctionne que si $p-1$ a de petits facteurs premiers, où p est l'un des facteurs premiers du nombre n qu'on veut factoriser. Enfin, la méthode du crible quadratique permet de factoriser des nombres dont la taille dépasse 10^{100}. Parmi les algorithmes les plus performants en ce moment, le "Generalized Number Field Sieve" (en français, le crible général de corps de nombres) peut factoriser des nombres aussi grands que 10^{200} [Inc06]. L'avantage de cette méthode est qu'on peut l'exécuter en parallèle sur un très grand nombre de machines. La théorie derrière le GNFS dépasse le cadre d'une introduction à la cryptographie et nous ne nous y attarderons pas. Toutefois, il est important de connaître la taille des nombres les plus grands qu'on peut factoriser en ce moment, car comme nous le verrons à la section 4.3.4.1, cela détermine la taille minimale sécuritaire des clés d'encryptage.

3.4 Extraction du logarithme discret

À compléter...

Chapitre

4

Encryptage par clés publiques

4.1 Méthode de Diffie-Hellman (DH)

Nous revenons maintenant à la méthode Diffie-Hellman, pour étudier plus en détail son fonctionnement et les précautions à prendre pour qu'elle soit sécuritaire.

4.1.1 DH : Comment ça marche

La méthode DH propose une façon de créer une clé symétrique unique à l'aide de deux nombres échangés par deux interlocuteurs. Chaque interlocuteur produit son nombre à être échangé à partir d'un nombre privé qu'il est seul à connaître. Par la suite, seuls les deux interlocuteurs peuvent obtenir la clé symétrique à l'aide de leur nombre privé respectif et du nombre reçu de l'autre. Mathématiquement, les deux nombres sont de la forme g^x et g^y, avec la clé unique $K \equiv g^{xy}$. En supposant qu'il est facile d'obtenir g^x à partir de g et x, mais très difficile d'obtenir x à partir de g et g^x, l'échange de g^x et g^y permet aux deux interlocuteurs de calculer $K \equiv g^{xy}$, alors que personne ne peut obtenir $K \equiv g^{xy}$ seulement à partir de g^x et g^y. En effet, chacun des deux interlocuteurs conserve en lieu sûr l'exposant (x ou y) utilisé pour générer le nombre échangé (g^x ou g^y). Ainsi, il devient possible pour celui qui reçoit g^y (et qui a envoyé g^x) de calculer $K \equiv (g^y)^x = g^{xy}$. De façon analogue, l'interlocuteur qui reçoit g^x (et qui a envoyé g^y) peut calculer $K \equiv (g^x)^y = g^{xy}$. Cependant, pour un tiers qui aurait intercepté g^x et g^y mais qui ne connaîtrait pas x ou y, il serait possible d'obtenir $g^x \times g^y = g^{(x+y)}$, mais pas $g^{xy} = (g^x)^y = (g^y)^x$.

Avec la méthode Diffie-Hellman, il est ainsi possible d'utiliser une clé d'encryptage/décryptage sans avoir à en faire l'échange de façon sécurisée. Il est important de remarquer que la méthode présentée plus haut n'est pas sécuri-

taire en pratique. En effet, on a supposé qu'un tiers ne pouvait obtenir x à partir de g et g^x, ce qui n'est pas le cas pour des nombres obtenus à l'aide d'opérateurs arithmétiques réguliers, puisque dans ce cas on peut obtenir x à l'aide de logarithmes ($x = \log(g^x)/\log(g)$). Cependant, pour s'assurer qu'on ne puisse obtenir x à partir de g et g^x, on utilise les résidus des opérations de multiplication et de mise à une puissance (résidus calculés pour une certaine base). En effet, il n'existe pas de méthode connue pour effectuer efficacement le logarithme du résidu lorsque la base est un très grand nombre.

La méthode de Diffie-Hellman complète est donc la suivante : À l'aide d'une base p (où p est un nombre premier de grande taille), définir un ensemble $\mathrm{Z}_p^* = \{1, 2, \cdots, p-1\}$, et choisir un nombre $g \in \mathrm{Z}_p^*$ qui sera utilisé pour calculer $g^x \bmod p$, $g^y \bmod p$ et $K = g^{xy} \bmod p$, où on choisira $x, y \in \mathrm{Z}_p^*$.

4.1.2 DH : Les pièges et les solutions

Pour utiliser la méthode de Diffie-Hellman de façon sécuritaire, il y a certains éléments auxquels on doit faire particulièrement attention. Il va de soi que la base p doit avoir une taille suffisante, et qu'on ne devrait jamais choisir $g = \pm 1 \bmod p$ (dans ce dernier cas, on obtient $K = g^{xy} \bmod p = \pm 1$, ce qui n'en fait pas une clé symétrique très difficile à découvrir...). Mais en plus de ces règles élémentaires, il existe d'autres considérations moins évidentes qui sont toutes aussi importantes. Considérons le nombre $g \in \mathrm{Z}_p^*$ qu'on pourrait être tenté de choisir sans utiliser de critères particuliers. La séquence de ses puissances sera: $g, g^2, g^3, \cdots, g^{p-1}$, à partir desquelles on obtiendra $g \bmod p, g^2 \bmod p, g^3 \bmod p, \cdots, g^{p-1} \bmod p$. Puisque p est par définition un nombre premier, alors le théorème de Fermat nous indique que $g^{p-1} \bmod p = 1$, ce qui fait que $g^p \bmod p = g$, c'est-à-dire qu'on est revenu au début de la séquence de puissances de g. Cette dernière relation explique pourquoi on choisit $x, y \in \mathrm{Z}_p^*$: seuls les exposants plus petits que $p-1$ donneront nécessairement des puissances différentes. Cependant une autre question, dérivée de la précédente, est aussi d'une extrême importance: quel est le plus petit exposant à partir duquel la séquence de puissances de g revient sur elle-même? Si c'est $p-1$, alors de façon analogue à ce qu'on a vu à la section 2.2, on peut dire que g^x est une génératrice de Z_p^*. Si ce n'est pas le cas, on doit se demander quelle est la puissance à partir de laquelle la séquence boucle sur elle-même. Si cette puissance est relativement petite, alors il y aura peu de puissances de g distinctes, et on peut imaginer qu'un tiers pourra trouver la clé symétrique $K = g^{xy} \bmod p$ par recherche exhaustive, puisque g et p sont connus.

Par exemple, avec $p = 29$, le calcul de $(g^x \bmod p)$ (où $g, x \in \mathrm{Z}_{29}^*$) donne les valeurs apparaissant à la figure 4.1.

On remarque dans l'exemple de la figure 4.1 qu'il n'y a que 12 valeurs de g dont les puissances sont des génératrices de Z_{29}^*, c'est-à-dire qu'il n'y a que ces 12 valeurs pour lesquelles on obtient:

g x	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	2	4	8	16	3	6	12	24	19	9	18	7	14	28	27	25	21	13	26	23	17	5	10	20	11	22	15	1
3	3	9	27	23	11	4	12	7	21	5	15	16	19	28	26	20	2	6	18	25	17	22	8	24	14	13	10	1
4	4	16	6	24	9	7	28	25	13	23	5	20	22	1	4	16	6	24	9	7	28	25	13	23	5	20	22	1
5	5	25	9	16	22	23	28	24	4	20	13	7	6	1	5	25	9	16	22	23	28	24	4	20	13	7	6	1
6	6	7	13	20	4	24	28	23	22	16	9	25	5	1	6	7	13	20	4	24	28	23	22	16	9	25	5	1
7	7	20	24	23	16	25	1	7	20	24	23	16	25	1	7	20	24	23	16	25	1	7	20	24	23	16	25	1
8	8	6	19	7	27	13	17	20	15	4	3	24	18	28	21	23	10	22	2	16	12	9	14	25	26	5	11	1
9	9	23	4	7	5	16	28	20	6	25	22	24	13	1	9	23	4	7	5	16	28	20	6	25	22	24	13	1
10	10	13	14	24	8	22	17	25	18	6	2	20	26	28	19	16	15	5	21	7	12	4	11	23	27	9	3	1
11	11	5	26	25	14	9	12	16	2	22	10	23	21	28	18	24	3	4	15	20	17	13	27	7	19	6	8	1
12	12	28	17	1	12	28	17	1	12	28	17	1	12	28	17	1	12	28	17	1	12	28	17	1	12	28	17	1
13	13	24	22	25	6	20	28	16	5	7	4	23	9	1	13	24	22	25	6	20	28	16	5	7	4	23	9	1
14	14	22	18	20	19	5	12	23	3	13	8	25	2	28	15	7	11	9	10	24	17	6	26	16	21	4	27	1
15	15	22	11	20	10	5	17	23	26	13	21	25	27	28	14	7	18	9	19	24	12	6	3	16	8	4	2	1
16	16	24	7	25	23	20	1	16	24	7	25	23	20	1	16	24	7	25	23	20	1	16	24	7	25	23	20	1
17	17	28	12	1	17	28	12	1	17	28	12	1	17	28	12	1	17	28	12	1	17	28	12	1	17	28	12	1
18	18	5	3	25	15	9	17	16	27	22	19	23	8	28	11	24	26	4	14	20	12	13	2	7	10	6	21	1
19	19	13	15	24	21	22	12	25	11	6	27	20	3	28	10	16	14	5	8	7	17	4	18	23	2	9	26	1
20	20	23	25	7	24	16	1	20	23	25	7	24	16	1	20	23	25	7	24	16	1	20	23	25	7	24	16	1
21	21	6	10	7	2	13	12	20	14	4	26	24	11	28	8	23	19	22	27	16	17	9	15	25	3	5	18	1
22	22	20	5	23	13	25	28	7	9	24	6	16	4	1	22	20	5	23	13	25	28	7	9	24	6	16	4	1
23	23	7	16	20	25	24	1	23	7	16	20	25	24	1	23	7	16	20	25	24	1	23	7	16	20	25	24	1
24	24	25	20	16	7	23	1	24	25	20	16	7	23	1	24	25	20	16	7	23	1	24	25	20	16	7	23	1
25	25	16	23	24	20	7	1	25	16	23	24	20	7	1	25	16	23	24	20	7	1	25	16	23	24	20	7	1
26	26	9	2	23	18	4	17	7	8	5	14	16	10	28	3	20	27	6	11	25	12	22	21	24	15	13	19	1
27	27	4	21	16	26	6	17	24	10	9	11	7	15	28	2	25	8	13	3	23	12	5	19	20	18	22	14	1
28	28	1	28	1	28	1	28	1	28	1	28	1	28	1	28	1	28	1	28	1	28	1	28	1	28	1	28	1

Figure 4.1: Résultats des puissances dans Z_{29}^*

$$\{g \bmod p, g^2 \bmod p, g^3 \bmod p, \cdots, g^{p-1} \bmod p\} = \{1, 2, \cdots, p-1\} = \mathrm{Z}_p^*$$

Les autres valeurs produisent des cycles (zones grisées dans le tableau) qui se répètent. Par exemple, en choisissant $g = 17$, il n'y aurait que 4 nombres possibles pour $g^{xy} \bmod p$, soient 1, 12, 17 et 28. Pour avoir un système qui soit le plus sécuritaire possible, idéalement on devrait donc choisir un nombre g qui génère Z_p^*. Ce choix est malheureusement difficile à effectuer, puisque pour une base arbitraire, on doit procéder par essais et erreurs pour déterminer si les puissances d'un nombre g sont génératrices de Z_p^*. Or justement on veut choisir p et g pour s'assurer que personne ne puisse énumérer toutes les puissances de $g^x \bmod p$... La solution se trouve dans une étude plus attentive des cycles de puissances. En observant la figure 4.1, on s'aperçoit que les cycles ont des longueurs bien définies: 1, 2, 4, 7 et 14. Ces longueurs correspondent aux facteurs de $(p-1)$. Si on y réfléchit un peu, cela va de soi : nous savons que $g^{p-1} \bmod p = 1$, donc tous les cycles doivent entrer un nombre entier de fois dans $(p-1)$ et se terminer par la valeur 1 mod p.

À partir de ces observations, on peut utiliser une méthode pour découvrir des nombres p et g qui donnent un système DH sécuritaire. Puisque la longueur des cycles est déterminée par les facteurs de $(p-1)$, on peut choisir p de telle sorte

que $(p-1)$ aura des facteurs intéressants. En particulier, on peut commencer par trouver un nombre premier q ayant une taille acceptable (par exemple, un nombre de 256 bits). Puis, on multiplie le nombre premier q par un nombre pair arbitraire N de telle sorte que $p = qN + 1$ soit aussi un nombre premier, d'une taille prédéterminée. Par exemple, pour obtenir un nombre p de 2 048 bits à partir d'un nombre q de 256 bits, on doit utiliser un nombre N de $2\,048 - 256 = 1\,792$ bits. Évidemment, on doit habituellement essayer plusieurs valeurs de N avant de trouver une valeur qui produise un nombre premier $p = qN + 1$.

Lorsque les nombres premiers p et q ont été déterminés, il reste à trouver un nombre g acceptable. Puisque $(p-1)$ est un multiple de q, alors il existe assurément des nombres g dont les puissances ont un cycle de longueur q. Or le dernier nombre d'un cycle doit nécessairement être 1. Pour vérifier qu'un nombre g produit un cycle de longueur q, il s'agit donc de vérifier que $g^q \equiv 1 \pmod{p}$. Puisque q est un nombre premier, alors aucun autre facteur de $(p-1)$ ne peut être un facteur de q, et donc il ne peut y avoir de cycles plus courts qui produisent (après un certain nombre de répétitions) un cycle de longueur q.

Pour utiliser la méthode DH de façon sécuritaire, deux interlocuteurs Alice et Bob peuvent donc procéder de la manière suivante. Alice établit un triplet (q, p, g) qu'elle transmet à Bob. Pour s'assurer que ces nombres vont permettre un échange vraiment sécuritaire, Bob effectue les vérifications suivantes :

1. q est un nombre premier (vérifiable à l'aide de l'algorithme de Miller-Rabin)
2. p est un nombre premier (idem)
3. $p-1$ est un multiple de q (vrai si $p \equiv 1 \pmod{q}$)
4. g est acceptable (vrai si $g^q \equiv 1 \pmod{p}$)

Alice et Bob peuvent ensuite chacun déterminer un exposant (x pour Alice, y pour Bob), s'envoyer mutuellement les puissances correspondantes ($g^x \bmod p$, $g^y \bmod p$) sur le médium de communication non sécuritaire et obtenir une clé symétrique privée ($K = g^{xy} \bmod p$).

4.2 Méthode de Elgamal

Depuis 1976, de nombreuses méthodes d'encryptage à clés publiques/privées ont vu le jour. La méthode de Elgamal a été proposée en 1985. C'est une méthode qui s'apparente fortement à la méthode de Diffie-Hellman[1]. La différence

[1]Il est intéressant de noter que Taher Elgamal a fait son Ph.D. à Stanford sous la direction de Martin Hellman, graduant en 1984. Son nom apparaît souvent comme "El Gamal" dans la littérature, mais lui-même l'écrit Elgamal. C'est donc la forme que nous avons adopté [Mol04].

notable entre les deux méthodes est que celle de Elgamal permet directement l'encryptage et le décryptage d'un message. Dans ce qui suit, nous verrons comment fonctionne la méthode de Elgamal et quelles précautions prendre pour assurer un bon niveau de sécurité.

4.2.1 Elgamal : Comment ça marche

Tout comme la méthode Diffie-Hellman, la méthode Elgamal est basée sur l'utilisation de puissances de nombres modulo un nombre premier. Elle fonctionne comme suit [Elg85, Spi05, TW06]. Étant donné un nombre premier p, un nombre α qui est une racine primitive modulo p (c'est-à-dire, les puissances successives de α génèrent Z_p^*), et un nombre secret a utilisé pour calculer $\beta \equiv \alpha^a \pmod{p}$, on utilise le triplet (p, α, β) comme clé publique. Pour crypter un message, on choisit un entier aléatoire $0 \leq k < p$ et on l'utilise pour calculer $r \equiv \alpha^k \pmod{p}$. On crypte ensuite le message m à l'aide de k, β et p: $t \equiv \beta^k m \pmod{p}$. On fait ensuite parvenir le couple (r, t) à notre interlocuteur, qui peut décrypter le message en effectuant le calcul suivant: $m \equiv tr^{-a} \pmod{p}$. Le décryptage fonctionne puisque

$$tr^{-a} \equiv \beta^k m(\alpha^k)^{-a} \equiv (\alpha^a)^k m\alpha^{-ak} \equiv m \pmod{p}$$

4.3 Méthode de Rivest Shamir Adleman (RSA)

La présentation de la méthode RSA de la section 2.1 était incomplète, plusieurs éléments importants n'étant pas expliqués, en particulier comment trouver e, d et n. Dans ce qui suit, nous comblerons ces lacunes et discuterons des précautions à prendre pour que la méthode soit vraiment sécuritaire.

4.3.1 RSA : Comment ça marche

Le principe de la méthode RSA est le suivant : sous certaines conditions, le résidu d'un nombre élevé à une puissance est exactement égal au nombre de départ: dans ce cas, on a $m^t \equiv m \pmod{n}$. Nous avons déjà vu aux sections 2.3.3 et 2.3.4 que c'est exactement la forme du petit théorème de Fermat ainsi que du théorème de l'indicatrice d'Euler. En décomposant la puissance t en un produit de deux nombres e et d (avec $t = e \times d$), on peut alors faire une première élévation à une puissance et produire le résidu du résultat: $c = m^e \bmod n$. Le nombre c qu'on obtient est complètement différent du nombre initial m (il est crypté). Pour réobtenir le nombre initial, il s'agit ensuite d'utiliser le deuxième facteur d de t comme exposant d'une deuxième élévation à une puissance. On

effectue alors $c^d \bmod n$, et puisque $c = m^e \bmod n$, alors cette deuxième élévation à une puissance donne:

$$\begin{aligned} c^d \bmod n &\equiv (m^e \bmod n)^d (\mathrm{mod} n) \\ &\equiv m^{e \times d} (\mathrm{mod} n) \\ &\equiv m^t (\mathrm{mod} n) \\ &\equiv m (\mathrm{mod} n) \\ &= m \end{aligned}$$

On a ainsi réobtenu le nombre de départ, et le message m est maintenant décrypté.

En partant du principe de fonctionnement exposé plus haut, comment pouvons-nous arriver à un système opérationnel? Les paramètres importants sont n, t, e et d. Nous devons établir une méthode pour déterminer ces quatre paramètres et pour faire en sorte que le message crypté soit en sécurité, c'est-à-dire qu'il soit excessivement difficile pour un tiers non autorisé de le décrypter. Évidemment, le but de la méthode RSA est de permettre l'échange de message cryptés à l'aide de clés publiques et décryptés à l'aide de clés privées. Il faut donc aussi déterminer ce qui sera échangé publiquement, et ce qui sera conservé à l'abri des regards indiscrets.

Tentons une première solution au problème. Puisque la relation qui nous intéresse ressemble au petit théorème de Fermat, utilisons des nombres auxquels le théorème s'applique: nous prendrons un nombre premier pour n, et nous prendrons $t = k \times (n-1) + 1$. Selon le petit théorème de Fermat, $a^{(n-1)} \equiv 1 \pmod{n}$ lorsque n est premier et a est relativement premier avec n. En prenant $0 < a < n$, la deuxième condition est assurée (puisque n est premier par définition, tous les nombres entiers positifs plus petits que n y sont relativement premiers). Du petit théorème de Fermat, on tire : $a^{k(n-1)+1} \equiv a \pmod{n}$, où k est un entier positif quelconque, et on a obtenu exactement la relation qu'on recherchait pour t. Il nous reste à trouver deux nombres e et d tels que leur produit soit égal à t. Remarquons que t représente un nombre entier qui suit immédiatement un multiple de $(n-1)$. En d'autres termes, nous recherchons une paire de nombres e et d dont le produit sera égal à $1 \pmod{n-1}$. C'est donc que nous recherchons deux nombres qui sont des inverses multiplicatifs modulo $(n-1)$. Nous avons vu auparavant que ceci n'est possible que si les deux nombres sont relativement premiers avec $(n-1)$. On peut donc choisir un nombre e relativement premier avec $(n-1)$, puis on pourra calculer d à l'aide de l'algorithme étendu d'Euclide.

Par exemple, prenons $n = 11$. Alors $(n-1) = 10$, et on doit choisir e relativement premier à 10. Prenons $e = 7$. À l'aide de l'algorithme étendu d'Euclide, on obtient $d = 3$. On a donc obtenu deux clés, l'une publique, pour l'encryptage: $(n, e) = (11, 7)$ et l'autre privée, pour le décryptage: $(n, d) = (11, 3)$. Maintenant, tentons de crypter un nombre quelconque $0 < m < 11$. Soit $m = 4$. Alors la version cryptée c de m est:

$$\begin{aligned} c &= m^7 \bmod 11 \\ &= 4^7 \bmod 11 \\ &= 16\,384 \bmod 11 \\ &= 5 \end{aligned}$$

Pour décrypter c, on refait la même opération, cette fois avec d:

$$\begin{aligned} m &= c^3 \bmod 11 \\ &= 5^3 \bmod 11 \\ &= 125 \bmod 11 \\ &= 4 \end{aligned}$$

Nous avons donc obtenu un système d'encryptage et de décryptage qui fonctionne dans les deux sens, produisant un message chiffré qu'on peut ensuite déchiffrer. Toutefois, une question importante demeure: est-ce que notre système d'encryptage est sécuritaire? On peut reformuler cette question en se demandant si un tiers non autorisé peut décrypter un message à partir du message crypté c et de la clé publique (n, e). Malheureusement, la réponse est oui, il est très facile de retrouver la clé privée à partir de la clé publique. Puisque $d = e^{-1} \mod (n-1)$, l'algorithme étendu d'Euclide nous permet de trouver d directement, et il s'agit ensuite d'effectuer $c^d \bmod n$ pour retrouver le message d'origine... Notre premier essai, quoique fonctionnel, n'est donc pas acceptable puisque la clé de décryptage ne peut être gardée secrète.

La façon de rendre notre système sécuritaire est relativement simple, et implique de passer du petit théorème de Fermat au théorème de l'indicatrice d'Euler. Le théorème d'Euler ($a^{k\times\phi(n)+1} = a \bmod n$) s'applique entres autres à des nombres composés. Ce qui est intéressant ici, c'est la façon de calculer l'indicatrice $\phi(n)$. Pour un nombre n qui est le produit de deux nombres premiers p et q, $\phi(n) = (p-1) \times (q-1)$. Pour calculer $\phi(n)$, il faut donc trouver p et q auparavant. Si on ne connaît que n, il faut le factoriser avant de pouvoir obtenir $\phi(n)$. Si p et q sont petits, cela est aisé. Cependant il n'existe pas d'algorithme connu qui permette de factoriser rapidement n lorsque c'est un nombre suffisamment grand. Pour qui choisit p et q, $\phi(n)$ est connu et il est facile de trouver un nombre e relativement premier avec $\phi(n)$, puis de calculer son inverse multiplicatif: $d \equiv e^{-1} \pmod{\phi(n)}$. On peut par la suite rendre publique la clé (n, e), et personne ne pourra obtenir la clé privée (n, d) seulement à partir de la clé publique, puisque pour ce faire il faudrait factoriser n.

4.3.2 RSA : Questions variées

Cette section présente certains aspects plus techniques de la méthode RSA. On y étudie de plus près le type de messages qui peuvent être cryptés à l'aide de cette méthode, et on observe les solutions possibles pour les exposants $\phi(n)$, e

et d. Ces questions sont importantes, parce que dans certains cas elles peuvent avoir un impact majeur sur la sécurité globale du système.

4.3.2.1 Multiples de p et q

En étudiant la méthode RSA, on peut se demander s'il existe des limites qui encadrent son fonctionnement. En particulier, est-ce que cette méthode permet de crypter tous les messages $0 \leq m < n$? Selon le théorème de l'indicatrice d'Euler, la relation $m^{\phi(n)} \equiv 1 \pmod{n}$ n'est vraie que si m est relativement premier avec n. Puisque $n = p \times q$, il est certain qu'il existe un certain nombre de multiples de p et de q contenus entre 0 et n qui ne satisfont pas cette condition: $\{p, 2p, 3p, \ldots, (q-1)p\}$ et $\{q, 2q, 3q, \ldots, (p-1)q\}$ ne sont pas relativement premiers avec n. Peut-on tout de même utiliser la formule

$$m^{e \times d} = m^{k \times \phi(n)+1} \equiv m \pmod{n}$$

pour crypter puis décrypter ces nombres, ou bien faut-il les éviter? Nous répondrons à cette question en utilisant le théorème du reste chinois. Prenons l'un des multiple de p, sp, contenu entre 0 et n et utilisons la représentation vectorielle de résidus: $sp \overset{p \times q}{\longleftrightarrow} (0, sp \bmod q)$. Effectuons maintenant la mise à la puissance $k \times \phi(n) + 1$ de sp et de son vecteur de résidus. Le premier terme du vecteur de résidus demeurera 0, alors que le second terme sera:

$$\begin{aligned}
(sp \bmod q)^{k \times \phi(n)+1} &= (sp \bmod q)^{k \times (p-1) \times (q-1)+1} \\
&= (((sp)^{(q-1)} \bmod q)^{k \times (p-1)} \times (sp \bmod q)) \bmod q \\
&= (1^{k \times (p-1)} \times (sp \bmod q)) \bmod q \quad \text{(théorème de Fermat)} \\
&\equiv sp (\bmod q)
\end{aligned}$$

On obtient donc:

$$(sp)^{k \times \phi(n)+1} \overset{p \times q}{\longleftrightarrow} (0, sp(\bmod q))$$

On remarque que la représentation vectorielle de résidus est identique à celle de sp, ce qui implique que

$$(sp)^{k \times \phi(n)+1} \equiv sp(\bmod n)$$

On voit que les multiples de p peuvent être cryptés et décryptés sans problème par la méthode RSA. Un raisonnement analogue prouve aussi la validité de la méthode avec les multiples de q.

Une remarque importante découle du calcul précédent: puisque la représentation vectorielle des résidus des nombres p et q contient un 0, ceci indique que lors de l'encryptage (la mise à la puissance e), le nombre résultant aura un facteur commun (p ou q) avec le nombre de départ. Le nombre crypté ne sera donc pas totalement indépendant du nombre initial dans ce cas.

Par exemple, soit $p = 7$ et $q = 11$, nous donnant $n = 77$ et $\phi(n) = 60$. Prenons $e = 13$ (et $d \equiv e^{-1} \bmod \phi(n) = 37$). En cryptant les 10 messages m_i qui sont des multiples de 7:

$$(7, 14, 21, 28, 35, 42, 29, 56, 63, 70)$$

on obtient les 10 messages cryptés

$$c_i = m_i^{13} \bmod 77 : (35, 49, 21, 7, 63, 14, 70, 56, 28, 42),$$

et tous les messages cryptés sont des multiples de 7. De même, l'encryptage des 6 messages qui sont des multiples de 11 produit des multiples de 11.

4.3.2.2 Encryptage et décryptage de 0, 1 et $(n-1)$

La méthode RSA est basée sur l'élévation à une puissance (couplée à une opération modulo) de nombres à crypter, où les nombres représentent des messages. Ultimement, ce qu'on désire c'est évidemment l'obtention de nombres "cryptés" à partir desquels on peut difficilement retrouver le nombre d'origine. L'inverse de l'élévation à une puissance de résidus est l'extraction de logarithmes de résidus. De façon analogue à la factorisation de très grands nombres, on ne connaît pas de méthode efficace pour faire l'extraction de logarithmes de résidus. À partir d'un nombre qui a été " crypté " à l'aide d'une élévation de puissance modulo une certaine base, il n'est donc pas possible de retrouver le nombre initial en calculant le logarithme. Or il existe trois nombres particuliers (0, 1 et $n-1$) dont l'élévation à une puissance modulo la base n produit des nombres "cryptés" qui ne cachent pas très bien les nombres originaux. En effet, $0^t \bmod n = 0$ et $1^t \bmod n = 1$, quels que soient t ou n, alors que $(n-1)^t \equiv \pm 1 \pmod{n}$. Pour ce qui est de $(n-1)^t \bmod n$, on obtient $1 \bmod n$ si t est pair, et $-1 \bmod n$ si t est impair. Puisque dans la méthode RSA l'exposant est toujours impair (il doit être relativement premier avec $\phi(n)$ et celui-ci est toujours pair), alors l'encryptage de $n-1$ à l'aide de RSA va toujours produire $n-1$ comme message crypté.

Les considérations précédentes nous démontrent qu'on ne peut crypter efficacement les nombres 0, 1 et $n-1$ à l'aide de la méthode RSA.

4.3.2.3 Utilisation de $\phi(n)$, $\lambda(n)$ et $PPCM(p-1, q-1)$

Dans la plupart des descriptions de la méthode RSA, on utilise $\phi(n)$ pour calculer d à partir de e, le tout étant basé sur le théorème de l'indicatrice d'Euler. Quoique cette façon de faire soit correcte, on peut poser le problème un

peu différemment et arriver à une solution distincte [FS03]. La nouvelle question à se poser est la suivante: quel est le plus petit exposant t pour lequel la relation $m^t \equiv 1 \pmod{n}$ est vraie ? Est-il possible de trouver pour t une valeur plus petite que $\phi(n)$ pour laquelle la relation est encore valide? R.D. Carmichael s'est posé cette question en 1910 et y a répondu en proposant la fonction $\lambda(n)$[Car10]. Lorsque n est le produit de deux nombres premiers p et q, $\lambda(n) = \frac{(p-1)\times(q-1)}{PGCD(p-1,q-1)}$. On peut vérifier la validité de cette expression en utilisant le théorème du reste chinois: puisque $m^t \equiv 1 \pmod{n}$ et que $n = p \times q$, alors $m^t \overset{p\times q}{\longleftrightarrow} (1(\mathrm{mod}p), 1(\mathrm{mod}q))$, c'est-à-dire $m^t \equiv 1(\mathrm{mod}p)$ et $m^t \equiv 1(\mathrm{mod}q)$. En utilisant le petit théorème de Fermat, on peut ensuite affirmer que les deux dernières relations seront satisfaites si t est à la fois un multiple de $(p-1)$ et de $(q-1)$. Le nombre t le plus petit qui satisfasse ces conditions est le plus petit commun multiple (PPCM) de $(p-1)$ et $(q-1)$, qui est le produit de ces deux nombres divisé par leur plus grand commun diviseur, précisément l'expression de $\lambda(n)$ lorsque $n = p \times q$. Lorsqu'on fait le calcul des nombres d et e de la méthode RSA, on peut donc utiliser $d \equiv e^{-1}(\mathrm{mod}\lambda(n))$ au lieu de $d \equiv e^{-1}(\mathrm{mod}\phi(n))$. Puisque p et q sont des nombres premiers impairs, alors $PGCD(p-1, q-1)$ est toujours plus grand ou égal à 2. Donc $\lambda(n)$ est strictement plus petit que $\phi(n)$, et le nombre d calculé à l'aide de $\lambda(n)$ sera plus petit que celui calculé à l'aide de $\phi(n)$. Ceci implique que le décryptage d'un message m par mise à la puissance d d'un message crypté c quelconque ($m = c^d \bmod n$) sera effectuée plus rapidement.

4.3.2.4 Comment forme-t-on le message m?

Tout message est une suite de caractères, qui peuvent être interprétés comme une suite de nombres. Puisque tout message à crypter doit être plus petit que n, on peut découper le message original en blocs de nombres qui sont plus petits que n. Par exemple, pour une clé où n utilise 1 024 bits, on peut découper notre message en blocs de $1\,024/8 = 128$ octets, et crypter chaque bloc indépendamment. En pratique, il arrive très rarement qu'on procède de cette façon, parce que la méthode RSA requière énormément de calculs quand on la compare aux méthodes à clé symétrique unique. La technique usuelle est donc d'utiliser la méthode RSA pour que deux interlocuteurs puissent échanger une clé symétrique privée. Par la suite, on utilise la clé symétrique pour échanger le message. On gagne ainsi en efficacité et en temps de calcul.

4.3.2.5 Choix de e

Lors de la création de la paire de clés, la première étape est de trouver (ou de générer) deux nombres premiers de taille acceptable. Par exemple, on utilisera deux nombres de 512 bits pour obtenir un n de 1 024 bits. Par la suite, il faut choisir un nombre e qui soit relativement premier avec $\lambda(n)$, et de ces deux

nombres obtenir $d \equiv e^{-1}(\mathrm{mod}\lambda(n))$. Dans ce scénario, e dépend en quelques sortes de $\lambda(n)$ et est conditionné par ce dernier, puisque e et $\lambda(n)$ doivent être relativement premiers. Il ne peut donc prendre une valeur déterminée à l'avance. Cependant, nous savons que dans un système à clés publiques de type RSA, la taille de l'exposant a un impact direct sur le temps de calcul, comme nous l'avons vu à la section 2.2.2. Puisque l'exposant e est public, sa taille n'a aucun impact sur la sécurité du système global. Une pratique courante est donc de faire en sorte que e soit petit, et même de prédéfinir sa valeur. Par exemple, on peut décider que pour notre système, $e = 3$. Il s'agit par la suite de s'assurer que $\lambda(n)$ ne se divise pas par 3. Puisque $\lambda(n) = \frac{(p-1)(q-1)}{PGCD(p-1,q-1)}$, on peut faire en sorte que p et q soient différents de 1 (mod 3).

On peut modifier l'algorithme 2.7 pour filtrer les nombres premiers dont le modulo 3 est 1 : il s'agit de retrancher les résidus 1, 7, 13 et 19 de la liste de résidus acceptables modulo 30. On obtient alors l'algorithme 4.1.

```
trouver_premier_e3(taille) {
  TAILLE_RESIDUS = 4 ;
  BASE = 30 ;
  residu [TAILLE_RESIDUS] = {11,17,23,29};
  for (i = 0;
       i < TAILLE_RESIDUS;
       i++) {                          // delta[] = {6,6,6,12}
    delta [i] = (residu [(i+1) % TAILLE_RESIDUS]
                 - residu [i]) % BASE;
  }
  temp = nombre_aleatoire(taille);     // Nombre d'une certaine taille
  temp_residu = temp % BASE ;
  for (i = 0 ;
       (i < TAILLE_RESIDUS) && (residu[i] < temp_residu);
       i++) ;
  if (residu [i % TAILLE_RESIDUS] != temp_residu) {
    temp = temp
    + residu[i % TAILLE_RESIDUS]   // On demarre avec un nombre dont
    - temp_residu;                 // le residu est acceptable
  }
  if (i >= TAILLE_RESIDUS) {
    temp = temp + BASE;
  }
  while ( Miller_Rabin(temp) ! = TRUE) {
    temp = temp + delta [i];  // Utilisation des residus acceptables
    i = (i+1) % TAILLE_RESIDUS;
  }
  return(temp) ;
}
```

Algorithme 4.1: Génération de nombres premiers utilisables avec e ≡ 3

4.3.3 RSA : Signature électronique

L'un des aspects intéressants de la méthode RSA est qu'on peut l'utiliser aussi pour signer électroniquement un document. Le principe de cette opération est le suivant : puisque e et d ont des fonctions complémentaires, on peut utiliser arbitrairement l'un des deux lors de "l'encryptage", puis on "décryptera" avec l'autre. Nous savons que tous nos correspondants possèdent la clé e, et que nous sommes les seuls à posséder la clé d. Alors, si un usager veux signer un document quelconque, c'est-à-dire transformer le document de telle sorte que tous soient certains que lui et lui seul a pu faire cette transformation, il n'a qu'à crypter le document avec sa clé privée. Ainsi, tous pourront le décrypter avec sa clé publique, et tous sauront qu'il est le seul qui puisse faire cet encryptage.

Cette capacité de la méthode RSA à permettre la signature de documents est très intéressante, mais elle doit être utilisée avec soin. Un tiers mal intentionné pourrait faire parvenir un document à signer, document qui est en fait un document crypté par quelqu'un d'autre à l'aide de la clé publique du signataire. Le tiers, recevant le document "signé", recevrait en réalité le document décrypté!

Pour éviter ce genre de problème, il est habituel de produire deux paires de clés pour un nombre n donné. L'une des paires de clés est utilisée uniquement pour l'encryptage et le décryptage de documents, alors que la seconde paire de clés est utilisée uniquement pour la signature et l'authentification de signature [FS03]. De cette manière, il est impossible qu'un usager décrypte par mégarde l'un de ses documents et l'envoie à un tiers mal intentionné. Par exemple, on peut générer deux paires de clés, l'une avec $e = 3$, et l'autre avec $e = 5$, en utilisant une méthode analogue à celle exposée dans la section 4.3.2.5. Il s'agit alors de n'utiliser que les nombres relativement premiers avec la base 30 qui sont différents de 1 (mod 3) et 1 (mod 5). En conséquence, on doit enlever le résidu 11 de la liste utilisée dans l'algorithme 4.1. En n'utilisant plus que les 3 résidus 17, 23 et 29, l'algorithme 4.1 va trouver des nombres premiers qui peuvent être utilisés simultanément pour les exposants $e = 3$ et $e = 5$.

4.3.4 RSA : Les attaques possibles

Depuis plus de 25 ans qu'elle est utilisée et étudiée, la méthode RSA a fait ses preuves pour ce qui est de sa sécurité. D'un point de vue théorique, c'est une méthode dont la sécurité est très solide. Malheureusement, dans la pratique, il existe des situations particulières où il peut être possible de décrypter un message crypté avec RSA [Bon99]. Dans ce qui suit, nous verrons différentes méthodes qui ont été proposées pour percer le système RSA. L'intérêt de faire cette revue est qu'elle nous permet de connaître les problèmes potentiels et d'y remédier avant qu'ils ne se présentent.

La façon la plus simple de décrypter un message RSA est d'obtenir la clé de décryptage. Pour ce faire, nous avons vu plus haut qu'il faut obtenir $\lambda(n)$, afin

de calculer (n, d) à partir de (n, e). Ce qui nous amène à tenter de factoriser n, puisque $\lambda(n)$ s'obtient à partir des facteurs premiers de n. Nous avons déjà étudié au chapitre 2 quelques méthodes de factorisation qui fonctionnent pour des nombres qui sont d'assez bonne taille, tout en étant relativement aisées à comprendre. Nous avons vu par la suite ce que permettent les techniques de factorisation les plus récentes. Nous verrons dans ce qui suit quel impact ont ces méthodes sur la taille requise de la base n, pour assurer la sécurité du message crypté. Puis, nous aborderons le problème de l'obtention de la clé de décryptage d'un tout autre point de vue, celui du temps de calcul requis pour le décryptage. Nous terminerons cette revue des attaques possibles en étudiant ce qui se passe lorsque le message et l'exposant d'encryptage sont petits.

4.3.4.1 Factorisation du nombre n

Dans les systèmes d'encryptage de type RSA actuels, on utilise habituellement des nombres à 1 024 ou même 2 048 bits (soit des nombres de l'ordre de 10^{300} ou 10^{600}). En règle générale, les algorithmes de factorisation vus à la section 3.3 ne sont donc d'aucune utilité pratique, sauf si les nombres à factoriser ont des caractéristiques bien précises. Il n'existe actuellement aucun algorithme qui soit capable de factoriser de tels nombres. En 2001, la compagnie RSA Security a lancé un défi qui consiste à factoriser une séquence de nombres de plus en plus grands, pour lesquels une récompense de plus en plus importante est offerte[2] . En décembre 2003, le premier de ces nombres, RSA-576 (qui comporte 174 chiffres décimaux) a été factorisé par une équipe allemande qui a utilisé le "Lattice Sieving". Depuis, RSA-200 (un nombre de 663 bits) a été factorisé en mai 2005 et RSA-640 a été factorisé en novembre 2005. Plus récemment, en décembre 2009, RSA-768 a été factorisé par une équipe regroupant des chercheurs de nombreuses institutions[KAF+09]. Il est important de connaître la taille des nombres les plus grands qu'on peut factoriser en ce moment: ceci nous indique quelle taille de clé est encore sécuritaire. Ainsi, depuis 1999, il n'est désormais plus sécuritaire d'utiliser des clés publiques à 512 bits. Par contre, les clés à 1 024 bits sont encore sécuritaires pour le moment. Il faut cependant comprendre que même une clé à 512 bits exige beaucoup de temps pour être factorisée (plusieurs semaines sur une grande quantité de machines), et que ce n'est donc pas à la portée de tous.

4.3.4.2 Méthode du délai et autres attaques " latérales "

En 1996, Paul Kocher a présenté une nouvelle façon d'attaquer les systèmes d'encryptage, en utilisant le délai de calcul encouru lors du décryptage d'un message connu [Koc96]. Le point central de cette méthode est qu'en connaissant l'algorithme de décryptage, il est possible de l'analyser pour savoir quelles

[2]Les récompenses offertes vont de 10 000 $ pour RSA-576 à 200 000 $ pour RSA-2048. Voir [Inc06] pour plus d'information sur le concours de décryptage.

opérations sont plus longues. Il est aussi possible de déterminer quels types de paramètres (quelles valeurs de clés) vont utiliser les longues instructions. Par la suite, en déterminant les temps de calcul requis pour décrypter un ensemble de messages connus, on peut arriver à réduire de beaucoup le nombre de clés qui produiront une signature temporelle donnée.

Depuis, plusieurs méthodes ont été proposées pour faire ce que certains appellent des attaques "latérales", c'est-à-dire des attaques qui utilisent de l'information supplémentaire aux seules valeurs des clés publiques [KSWH00]. Par exemple, Kocher a proposé une méthode apparemment très efficace pour déterminer les clés utilisées dans les cartes à puces, en mesurant les variations de courant consommé [KJJ99]. Si elles sont faites avec assez de précision, ces mesures peuvent permettre de déterminer quelles instructions sont exécutées, et donc de trouver (au moins partiellement) sur quelles données travaille l'algorithme de décryptage.

4.3.4.3 Les petits e et les petits m

Nous avons vu à la section 4.3.2.5 qu'il est avantageux de produire des clés publiques où l'exposant e est petit, pour ainsi réduire le temps d'encryptage. Cependant, il faut être vigilant lorsqu'on procède ainsi, parce qu'il peut arriver qu'on produise des messages qui ne sont pas vraiment cryptés. En effet, si le message à crypter m est lui aussi petit, alors il est possible que le nombre m^e soit plus petit que la base n. Dans ce cas, m^e apparaît directement comme message crypté, et on peut déterminer efficacement la racine "$e^{\text{ième}}$" de ce nombre pour retrouver le message original.

4.3.5 RSA: Méthodes de protection

Pour chacune des attaques évoquées à la section 4.3.4, il existe des parades efficaces que nous présenterons rapidement ici. Le premier point d'importance à remarquer, c'est que la taille du nombre n doit absolument être suffisante. Depuis 1999, il est impératif d'utiliser des clés d'au moins 1 024 bits, c'est-à-dire des nombres p et q d'au moins 512 bits. On recommande d'utiliser des clés d'au moins 2 048 bits si la protection des documents cryptés doit durer au moins jusqu'en 2025 [FS03].

Pour ce qui est des attaques basées sur le délai, elles ne sont pas spécifiques à la méthode RSA. De même, la parade présentée ici s'applique à d'autres méthodes d'encryptage. Le coeur du problème vient du fait qu'un tiers peut facilement connaître et même créer le message qui se fait décrypter. Le tiers peut se servir de cette connaissance du message et des mesures de temps pour en déduire la clé privée. L'une des mesures souvent envisagée est d'ajouter un délai aléatoire au temps de décryptage. Malheureusement, sur un grand nombre de décryptages, la moyenne de ce délai aléatoire sera constante, et il sera possible de l'éliminer.

La mesure qui a le plus de succès est de choisir un certain nombre aléatoire et de trouver son inverse (à la puissance d) modulo n. Alors avant d'effectuer le décryptage d'un message, on le multiplie par le nombre aléatoire. Puis on fait le décryptage (c'est-à-dire, on effectue le calcul modulo n du nouveau message à la puissance d). Puis on termine par une multiplication par l'inverse du nombre aléatoire à la puissance d, et on récupère ainsi le message original. L'ajout du facteur aléatoire durant le décryptage fait que la personne mal intentionnée n'a plus d'information sur le message dont on fait le calcul de la puissance d, et il n'est plus possible pour cette personne de déduire quelle est cette puissance. Mathématiquement, le calcul qui est fait est le suivant:

$$\begin{aligned}(r^{-1})^d \times ((c \times r)^d \bmod n) \bmod n &= r^{-d} \times c^d \times r^d \bmod n \\ &= c^d \bmod n \\ &= (m^e)^d \bmod n \\ &= m^{ed} \bmod n \\ &= m\end{aligned}$$

Dans les publications en anglais, on nomme cette méthode "padding" ou "blinding".

Pour s'assurer que les petits m (par exemple dans le cas fréquent où on utilise la méthode RSA pour échanger une clé symétrique de 256 bits), la méthode habituelle est de s'assurer que le message crypté soit grand. Pour un petit message de taille connue, on peut par exemple multiplier par un multiple de 10 (ou de 2 en binaire) assez grand pour rendre le message initial insensible aux attaques liées aux petits m. Par exemple, pour échanger une clé symétrique de 256 bits avec une clé n de 1024 bits, on décale la clé symétrique de 768 bits (ou de façon équivalent, on multiplie m par 2^{768}, et on envoie $c' = (m')^e \bmod n = (m \times 2^{768})^e \bmod n$.

4.4 Méthodes à courbes elliptiques

[HMV04] *À compléter...*

4.4.1 Adaptation de la méthode Diffie-Hellman

À compléter...

4.4.2 Adaptation de la méthode Elgamal

À compléter... À compléter...

4.4.3 Adaptation de la méthode Rivest Shamir Adleman (RSA)

À compléter...

Annexe

A

Groupes et corps de Galois

[Blo87, Irv04, Con99, AF05]

Un groupe G est un ensemble d'éléments liés par un opérateur binaire $\bullet$ et qui obéissent aux axiomes suivants:

I : G est fermé, c'est-à-dire que si a et b font partie de G, alors $a \bullet b$ fait partie de G.

II : G est associatif : $a \bullet (b \bullet c) = (a \bullet b) \bullet c$ pour tous les éléments $a, b, c \in G$.

III : G possède un élément neutre e tel que $a \bullet e = e \bullet a = a$ pour tous les éléments $a \in G$

IV : G contient des inverses a' pour tous ses éléments $a \in G$, tels que $a \bullet a' = a' \bullet a = e$.

Exemple : Soit V_i un vecteur de 4 nombres entiers compris entre 1 et 4, où un entier ne peut apparaître qu'une seule fois dans le vecteur V_i. Soit P_4 l'ensemble de tous les vecteurs V_i possibles et différents. P_4 contient $n!$ éléments (toutes les permutations possibles de 4 nombres). Définissons maintenant un opérateur

À compléter...

Annexe

B

Preuves de formules diverses

B.1 Calcul des combinaisons

Le nombre de combinaisons de n objets pris m à la fois est:

$$\binom{n}{m} = \frac{n!}{m!(n-m)!}$$

Preuve : Considérons un ensemble de n objets. Choisissons un premier objet de l'ensemble. Les n objets sont disponibles, il y a donc n choix possibles. Choisissons maintenant un 2$^{\text{e}}$ objet parmi les $(n-1)$ objets restants. Pour le choix de ces deux objets, il y a donc $n \times (n-1)$ choix possibles. Continuant le raisonnement jusqu'à ce que m objets aient été choisis, il y aura $n \times (n-1) \times \cdots \times (n-m+1)$ choix possibles, c'est-à-dire: $\frac{n!}{(n-m)!}$. Dans ce calcul, nous avons énuméré tous les ordres possibles des m objets. Il y a $m!$ façons d'ordonner m objets, puisqu'il y a m choix pour le premier objet de la liste, puis $(m-1)$ pour le 2$^{\text{e}}$, et ainsi de suite jusqu'au dernier objet de la liste, qui est le seul restant. Dans une combinaison de n objets pris m à la fois, l'ordre des objets n'est pas important. On divise donc le nombre de choix évalué plus haut par le nombre de façons d'ordonner m objets : $\frac{n!}{(n-m)!} \times \frac{1}{m!}$

B.2 Formule du binôme de Newton

La formule du binôme de Newton est la suivante :

$$(a+b)^n = \sum_{i=0}^{n} \binom{n}{i} a^{n-i} b^i$$

Nous prouverons cette formule par induction :

1. Nous débutons avec l'équation $(a+b)^1 = a+b$. Dans ce cas, la formule du binôme de Newton est valide, puisque:

$$\binom{1}{0} = \frac{1!}{0!\times 1!} = 1 \quad \text{et} \quad \binom{1}{1} = \frac{1!}{1!\times 0!} = 1$$

   et que selon la formule du binôme, $(a+b)^1 = \binom{1}{0}a + \binom{1}{1}b = a+b$.

2. Supposons maintenant que la formule du binôme de Newton est valide pour $(a+b)^n$ et vérifions qu'elle l'est alors aussi pour $(a+b)^{n+1}$:

$$\begin{aligned}
(a+b)^{n+1} &= (a+b)\times(a+b)^n && \text{(I)}\\
&= (a+b)\times\sum_{i=0}^{n}\binom{n}{i}a^{n-i}b^i && \text{(II)}\\
&= \sum_{i=0}^{n}\binom{n}{i}a^{n+1-i}b^i + \sum_{i=0}^{n}\binom{n}{i}a^{n-i}b^{i+1} && \text{(III)}\\
&= \sum_{i=0}^{n}\tfrac{n!}{i!(n-i)!}a^{n+1-i}b^i + \sum_{i=0}^{n}\tfrac{n!}{i!(n-i)!}a^{n-i}b^{i+1} && \text{(IV)}\\
&= \sum_{i=0}^{n}\tfrac{(n+1)!(n+1-i)}{(n+1)i!(n+1-i)!}a^{n+1-i}b^i + \sum_{j=1}^{n+1}\tfrac{n!}{(j-1)!(n-j+1)!}a^{n-j+1}b^j && \text{(V)}\\
&= \sum_{i=0}^{n+1}\tfrac{(n+1)!(n+1-i)}{(n+1)i!(n+1-i)!}a^{n+1-i}b^i + \sum_{j=1}^{n+1}\tfrac{j(n+1)!}{j!(n+1)(n-j+1)!}a^{n-j+1}b^j && \text{(VI)}\\
&= \sum_{i=0}^{n+1}\tfrac{(n+1)!(n+1-i)}{(n+1)i!(n+1-i)!}a^{n+1-i}b^i + \sum_{j=0}^{n+1}\tfrac{j(n+1)!}{j!(n+1)(n-j+1)!}a^{n-j+1}b^j && \text{(VII)}\\
&= \sum_{i=0}^{n+1}\left(\tfrac{(n+1)!(n+1-i)}{(n+1)i!(n+1-i)!} + \tfrac{i(n+1)!}{i!(n+1)(n-i+1)!}\right)a^{n+1-i}b^i && \text{(VIII)}\\
&= \sum_{i=0}^{n+1}\tfrac{(n+1)!}{i!(n+1-i)!}a^{n+1-i}b^i && \text{(IX)}\\
&= \sum_{i=0}^{n+1}\binom{n+1}{i}a^{n+1-i}b^i && \text{(X)}
\end{aligned}$$

   Notes :

   Equation (V) : Dans la somme de gauche, $n!$ au numérateur est remplacé par $\frac{(n+1)!}{(n+1)}$, ce qui est équivalent. De même pour $(n-i)!$ au dénominateur. Dans la somme de droite, $(i+1)$ est remplacé par j, avec les ajustements d'usage aux bornes de la somme.

   Équation (VI) : Dans la somme de gauche, la borne supérieure est augmentée à $(n+1)$ sans autre ajustement, puisque le facteur $(n+1-i)$ est nul dans ce cas. Dans la somme de droite, $n!$ au numérateur est remplacé par $\frac{(n+1)!}{(n+1)}$. De même pour $(j-i)!$ au dénominateur.

   Équation (VII) : Dans la somme de droite, la borne inférieure est réduite à 0 sans autre ajustement, puisque le facteur j est nul dans ce cas.

Équation (VIII) : Fusion et factorisation des deux sommes: les bornes sont identiques.

Équation (IX) : Dans l'équation (VIII), les deux fractions ont le même dénominateur. Lors de leur addition, le terme $(n+1)!$ se factorise au numérateur, laissant $(n+1-i+i) = (n+1)$, qui se simplifie avec le même terme au dénominateur.

3. Puisque la formule est vraie pour $n = 1$, et qu'elle est vraie pour $(n+1)$ lorsqu'elle est vraie pour n, alors elle est vraie pour tous les n entiers et positifs.

B.3 Puissance du nombre premier p dans le nombre $N!$

Soit p un nombre premier plus petit que N un nombre entier positif. Alors, il existe un nombre w tel que $N! = p^w \times R$, où R est un nombre entier qui n'est pas un multiple de p (R est un produit où tous les nombres premiers plus petits que N (autres que p) apparaissent au moins une fois). Dans ce qui suit, nous démontrerons que $w \leq \frac{N-1}{p-1}$.

$$
\begin{array}{rcll}
w & = & \left\lfloor \frac{N}{p} \right\rfloor + \left\lfloor \frac{N}{p^2} \right\rfloor + \cdots + \left\lfloor \frac{N}{p^k} \right\rfloor \text{ où } k = \left\lfloor \frac{\ln N}{\ln p} \right\rfloor & \text{(I)} \\
& \leq & \frac{N}{p} + \frac{N}{p^2} + \cdots + \frac{N}{p^k} & \text{(II)} \\
& \leq & \sum_{i=1}^{\infty} \frac{N}{p^i} - \frac{1}{p^k} \sum_{i=1}^{\infty} \frac{N}{p^i} & \text{(III)} \\
& \leq & \left(1 - \frac{1}{p^k}\right) \sum_{i=1}^{\infty} \frac{N}{p^i} & \text{(IV)} \\
& \leq & \left(1 - \frac{1}{N}\right) \sum_{i=1}^{\infty} \frac{N}{p^i} & \text{(V)} \\
& \leq & (N-1) \times \left(\frac{1 - \frac{1}{p^\infty}}{1 - \frac{1}{p}} - 1 \right) & \text{(VI)} \\
& \leq & (N-1) \times \left(\frac{1}{1 - \frac{1}{p}} - 1 \right) & \text{(VII)} \\
& \leq & \frac{N-1}{p-1} & \text{(VIII)}
\end{array}
$$

Notes :

Equation (I) : Cette somme représente le nombre de fois que p apparaît dans l'ensemble de ses multiples qui sont plus petits ou égaux à N (les fractions successives correspondent aux multiples de p, p^2, etc., sans atteindre la puissance de p qui dépasse N).

Équation (II) : La fonction plancher d'un nombre X est nécessairement plus petite ou égale à ce nombre X.

Équation (V) : Posons $T = \frac{\ln N}{\ln p}$. Alors $k = \left\lfloor \frac{\ln N}{\ln p} \right\rfloor = \lfloor T \rfloor \leq T$. Ainsi, $p^k \leq p^T$. Or $p^T = p^{\frac{\ln N}{\ln p}} = N$. D'où $\left(1 - \frac{1}{p^k}\right) \leq \left(1 - \frac{1}{N}\right)$.

Équation (VI) : $1 + a + a^2 + \cdots a^m = \frac{1-a^{m+1}}{1-a}$. Ici, $a = \frac{1}{p}$, $m = \infty$ et la série commence avec a (et non avec 1, qu'il faut donc soustraire).

Équation (VII) : $\frac{1}{p^\infty} = 0$

Par exemple, soit $p = 3$ et $N = 29$. Alors $w \leq 14$ selon la relation établie plus haut. Vérifions le résultat en énumérant les multiples de 3 qui sont plus petits ou égaux à 29:

$$\begin{aligned}
3 &= \boxed{\mathbf{3}} \times 1 \\
6 &= \boxed{\mathbf{3}} \times 2 \\
9 &= \boxed{\mathbf{3}} \times \boxed{\mathbf{3}} \\
12 &= \boxed{\mathbf{3}} \times 4 \\
15 &= \boxed{\mathbf{3}} \times 5 \\
18 &= \boxed{\mathbf{3}} \times \boxed{\mathbf{3}} \times 2 \\
21 &= \boxed{\mathbf{3}} \times 7 \\
24 &= \boxed{\mathbf{3}} \times 8 \\
27 &= \boxed{\mathbf{3}} \times \boxed{\mathbf{3}} \times \boxed{\mathbf{3}}
\end{aligned}$$

Le nombre 3 apparaît 13 fois dans la liste ci-haut. En conséquence, le facteur premier 3 apparaît à la puissance $w = 13$ dans le produit 29!, ce qui confirme la borne établie auparavant.

Annexe

C

Systèmes et bibliothèques de calcul de grands nombres

C.1 GMP

GMP (GNU Multiple Precision Arithmetic Library) existe depuis 1991. Cette bibliothèque en C permet le calcul de nombres de tailles arbitraires. La bibliothèque est destinée à plusieurs champs d'expertise, dont la cryptographie. Un soin particulier est donné à la performance des méthodes et algorithmes implémentés. Ainsi, de nombreuses fonctions ont été codées directement en langage assembleur, et sont supportées sur de nombreux processeurs.

Début	1991
Langage	C et C++
Version courante	6.2.1 (novembre 2020)
Référence	[Gra10]
Hyperliens	Site web
Avantages	Rapidité Thread-safe
Notes	Précision arbitraire Plusieurs processeurs supportés au niveau assembleur

C.2 PARI/GP

De 1985 à 1994, Pari/GP a été développé à l'université Bordeaux I par Henri Cohen et son équipe. Depuis 1995, Karim Belabas, lui aussi de l'université de Bordeaux I, a pris la relève et dirige un ensemble de bénévoles. L'intérêt de ce système est qu'il offre à la fois une bibliothèque utilisable seule et un interpréteur (basé sur la bibliothèque) qui permet le développement de scripts et leur validation. De plus, le système offre un traducteur de scripts, qui produit du code C directement utilisable dans des applications tierces.

Début	1985
Langage	C et GP
Version courante	2.13.1 (janvier 2021)
Référence	[CB05]
Hyperliens	Site web
Avantages	Librairie et interpréteur GP interprété peut être traduit en C et compilé
Notes	Précision arbitraire

C.3 Python

[RD09] *À compléter...*

C.4 bc

À compléter...

C.5 C#: BigInteger Class

[Tan06] *À compléter...*

C.6 Big Integers in JavaScript

[Bai06] *À compléter...*

C.7 GiantInt - cross-platform C code

[Cra98] *À compléter...*

C.8 Big/Giant number package - Giant Numbers in Forth

[Hen06] *À compléter...*

C.9 java.math.BigInteger

[Sun03] *À compléter...*

Annexe

D

Systèmes et bibliothèques de cryptographie

D.1 Crypto++

Crypto++ a été rendu public en juin 1995. Écrit à l'origine par Wei Dai, cette bibliothèque open-source en C++est supportée depuis 2015 par le projet Crypto++, un regroupement de développeurs indépendants oeuvrant au développement, la mise à jour et la maintenance de l'ensemble. L'ensemble de fonctionnalités offertes est très complet, couvrant les éléments standards de la cryptographie moderne (AES, modes d'opérations divers pour le chiffrement par bloc, méthodes de hachage, génération de nombres pseudo-aléatoires, chiffrement par clés publiques/privées, etc.).

Début	1995
Langage	C++
Version courante	8.4.0 (janvier 2021)
Référence	[Dai08]
Hyperliens	Site web
Avantages	Utilisé par plusieurs projets en Code Ouvert (Open Source), tels PKIF et Secret Sharp
Notes	Précision arbitraire

D.2 Apache Milagro

Début	
Langage	C, C++, Go, Rust, Python, Java, Javascript, Swift
Version courante	2.0.1 (janvier 2020)
Référence	[Sco04]
Hyperliens	Github , Site web
Avantages	Bibliothèque unique accessible dans de nombreux langages
Notes	Précision arbitraire

D.3 Bouncy Castle

Début	
Langage	Java, C#
Version courante	Java: 1.68 (décembre 2020), C# : 1.8.10 (avril 2021)
Référence	[Sco04]
Hyperliens	Site web
Avantages	Bibliothèque Java/C#
Notes	Précision arbitraire

Bibliographie

[AF05] Marlow Anderson and Todd Feil. *A First Course in Abstract Algebra - Rings, Groups and Fields*. Chapman and Hall / CRC, second edition, 2005.

[AGP94] W. Alford, A. Granville, and C. Pomerance. There are infinitely many Carmichael numbers. *Annals of Mathematics*, 140:703–722, 1994. Voir: `http://www.dms.umontreal.ca/~andrew/PDF/carmichael.pdf`.

[AKS04] M. Agrawal, N. Kayal, and N. Saxena. Primes is in P. *Annals of Mathematics*, 160(2):781–793, 2004. Prépublié en 2002. Voir: `http://www.cse.iitk.ac.in/users/manindra/algebra/primality_v6.pdf`.

[Art67] M. Artjuhov. Certains critères de primalité des nombres utilisant le petit théorème de Fermat (en Russe). *Acta Arithmetica*, 12:355–364, 1966/67.

[Bai06] Leemon Baird. Big integers in javascript. Accessible en ligne, 2006. Voir: `http://www.leemon.com/crypto/BigInt.html`.

[BGS72] Michael Beeler, R. William Gosper, and Rich Schroeppel. HAKMEM. Memo 239, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1972. Voir: `http://www.inwap.com/pdp10/hbaker/hakmem/hakmem.html`.

[Blo87] Norman J. Block. *Abstract Algebra with Applications*. Prentice-Hall, 1987.

[BLS75] J. Brillhart, D. Lehmer, and J. Selfridge. New primality criteria and factorizations of $2m + 1$. *Mathematics of Computation*, 29:620–647, 1975.

[Bog05] A. Bogolmony. Cut the knot, extension of Euclid's algorithm. Accessible en ligne, 2005. Voir: `http://www.cut-the-knot.org/blue/extension.shtml`.

[Bon99] D. Boneh. Twenty years of attacks on the RSA cryptosystem. *Notices of the American Mathematical Society*, 46(2):203–213, 1999. Voir: `http://crypto.stanford.edu/~dabo/papers/RSA-survey.pdf`.

[Bre80] R. P. Brent. An improved Monte Carlo factorization algorithm. *BIT*, 20:176–184, 1980. Voir: `http://wwwmaths.anu.edu.au/~brent/pd/rpb051a.pdf`.

[Bre89] D. Bressoud. *Factorization and Primality Testing*. Springer-Verlag, 1989.

[Car10] R.D. Carmichael. Note on a new number theory function. *Bulletin of the American Mathematical Society*, 16:232–238, 1910. Voir: `http://projecteuclid.org/DPubS/Repository/1.0/Disseminate?view=body&id=pdf_1&handle=euclid.bams/1183420617`.

[Car12] R.D. Carmichael. On composite numbers which satisfy the Fermat congruence. *American Mathematical Society*, 19:22–27, 1912.

[CB05] H. Cohen and K. Belabas. The PARI/GP computer algebra system. Accessible en ligne, 2005. Voir: `http://pari.math.u-bordeaux.fr/`.

[Con99] E. H. Connell. *Elements of Abstract and Linear Algebra*. University of Miami, 1999. Voir: `http://www.math.miami.edu/~ec/book/book.pdf`.

[CP05] R. Crandall and C. Pomerance. *Prime Numbers, a Computational Perspective*. Springer-Verlag, 2nd edition, 2005.

[Cra98] R. Crandall. GiantInt - cross-platform C code. Accessible en ligne, 1998. Voir: `http://www.perfsci.com/freegoods/giantint.zip`.

[Dai08] W. Dai. Crypto++ Library. Accessible en ligne, 2008. Voir: `http://www.cryptopp.com/`.

[DH79] W. Diffie and M. E. Hellman. Privacy and authentication: An introduction to cryptography. *Proceedings of the IEEE*, 67(3):397–427, March 1979. Voir: `http://www-ee.stanford.edu/~hellman/publications/32.pdf`.

[Dic05] L. E. Dickson. *History of the Theory of Numbers - Volume 1: Divisibility and Primality*. Dover, 1919,2005.

[Die04] M. Dietzfelbinger. *Primality Testing in Polynomial Time, From Randomized Algorithms to "PRIMES is in P"*. Springer-Verlag, 2004.

[Dix81] J. D. Dixon. Asymptotically fast factorization of integers. *Mathematics of Computation*, 36(153):255–260, December 1981.

[DKR97] J. Daemen, L. Knudsen, and V. Rijmen. The block cipher square. In E. Biham, editor, *Fast Software Encryption*, volume 1267 of *Lecture Notes in Computer Science*, pages 149–165. Springer-Verlag, 1997. Voir: `http://www.esat.kuleuven.ac.be/~cosicart/pdf/VR-9700.PDF`.

[DOF+90] M. Damiani, P. Olivo, M. Favalli, S. Ercolani, and B. Ricco. Aliasing in signature analysis testing with multiple input shift registers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(12):1344–1353, December 1990.

[DR02] J. Daemen and V. Rijmen. *The Design of Rijndael, AES - The Advanced Encryption Standard*. Springer-Verlag, 2002.

[Dwo01] HM. Dworkin. Recommendation for block cipher modes of operation - method and techniques. NIST, December 2001. Voir: `http://www.csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf`.

[Elg85] T. Elgamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, IT-31(4):469–472, 1985.

[EP86] P. Erdös and C. Pomerance. On the number of false witnesses for a composite number. *Mathematics of Computation*, 173(46):259–279, January 1986.

[Fei73] H. Feistel. Cryptography and computer privacy. *Scientific American*, 228(5):15–23, May 1973. Voir: `http://www.prism.net/user/dcowley/docs.html`.

[Fic81] Faith E. Fich. Lower bounds for the cycle detection problem. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, Annual ACM Symposium on Theory of Computing, pages 95–105. ACM, 1981. Voir: `http://portal.acm.org/citation.cfm?doid=800076.802462`.

[FM01] S. R. Fluhrer and D. A. McGrew. Statistical analysis of the alleged RC4 keystream generator. In *7th International Workshop on Fast Software Encryption*, volume 1978 of *Lecture Notes in Computer Science*, pages 66–71. Springer-Verlag, 2001.

[FS03] N. Ferguson and B. Schneier. *Practical Cryptography*. Wiley Publishing, 2003.

[Gra04] A. Granville. It is easy to determine if a given integer is prime. *Bulletin of the American Mathematical Society*, 42(1):3–38, September 2004. Voir: `http://www.ams.org/bull/2005-42-01/S0273-0979-04-01037-7/S0273-0979-04-01037-7.pdf`.

[Gra10] Torbjorn Granlund. GNU MP - the GNU multiple precision arithmetic library. Accessible en ligne, 2010. Voir: `http://gmplib.org/gmp-man-5.0.1.pdf`.

[Har40] G. H. Hardy. *A Mathematician's Apology*. Cambridge University Press, 1940. Voir: `http://www.math.ualberta.ca/~mss/misc/A%20Mathematician's%20Apology.pdf`.

[Hen06] M. Hendrix. Big/Giant number package, Giant numbers in Forth. Accessible en ligne, 2006. Voir: `http://home.iae.nl/users/mhx/bignum.html`.

[HMV04] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag, 2004.

[IAI06] IAIK. AES. Accessible en ligne, 2006. Voir: `http://www.iaik.tu-graz.ac.at/research/krypto/AES/`.

[Inc06] Inconnu. RSA Laboratories. Accessible en ligne, 2006. Voir: `http://www.rsasecurity.com/rsalabs/node.asp?id=2093#RSA576`.

[Irv04] R. S. Irving. *Integers, Polynomials, and Rings*. Springer-Verlag, 2004.

[KAF+09] T. Kleinjung1, K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, J. W. Bos, P. Gaudry, A. Kruppa, P. L. Montgomery, D. A. Osvik, H. te Riele, A. Timofeev, and P. Zimmermann. Factorization of a 768-bit RSA modulus. *Cryptology ePrint Archive*, December 2009. Voir: `http://eprint.iacr.org/2010/006.pdf`.

[KJJ99] P.C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. Wiener, editor, *Advances in Cryptology - CRYPTO 1999, vol. 1666 of Lecture Notes in Computer Science*, pages 388–397. Springer-Verlag, 1999. Voir: `http://www.cryptography.com/resources/whitepapers/DPA.pdf`.

[KMP+98] L. R. Knudsen, W. Meier, B. Preneel, V. Rijmen, and S. Verdoolaege. Analysis methods for (alleged) RC4*. In *Proceedings of Asiacrypt 98*, pages 327–341. Springer-Verlag, 1998.

[Knu98] D.E. Knuth. *The Art of Computer Programming : Semi-numerical Algorithms*. Vol. 2. Addison-Wesley, 3rd edition, 1998.

[Kob87] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987. Voir: `http://www.ams.org/notices/200708/tx070800972p.pdf`.

[Kob07] N. Koblitz. The uneasy relationship between mathematics and cryptography. *Notices of the American Mathematical Society*, 54(8):972–979, 2007. Voir: `http://www.ams.org/notices/200708/tx070800972p.pdf`.

[Koc94] C.K. Koc. High-speed RSA implementation. RSA Laboratories, November 1994. Voir: `ftp://ftp.rsasecurity.com/pub/pdfs/tr201.pdf`.

[Koc96] P.C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS and other systems. In N. Koblitz, editor, *Advances in Cryptology - CRYPTO 1996, vol. 1109 of Lecture Notes in Computer Science*, pages 104–113. Springer-Verlag, 1996. Voir: `http://www.cryptography.com/resources/whitepapers/TimingAttacks.pdf`.

[Kra22] M. Kraitchik. *Théorie des Nombres*. Gauthier-Villars, 1922.

[KSWH00] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Side channel cryptanalysis of product ciphers. *Journal of Computer Security*, 8(2-3):141–158, 2000. Voir: `http://www.counterpane.com/side_channel.html`.

[LL93] A. K. Lenstra and H. W. Lenstra. *The Development of the Number Field Sieve*. Springer-Verlag, 1993.

[LRW00] H. Lipmaa, P. Rogaway, and D. Wagner. CTR mode encryption. NIST First Modes of Operation Workshop, October 2000. Voir: `http://www.csrc.nist.gov/encryption/modes`.

[Mar88] J.C. Martzloff. *Histoire des mathématiques chinoises*. Masson, 1988.

[Mat24] L. C. Mathewson. Review: M. Kraïtchik, Théorie des Nombres. *Bulletin of the American Mathematical Society*, 30(8):467, 1924. Voir: `http://projecteuclid.org/euclid.bams/1183486154`.

[MB76] R.M. Metcalfe and D.R. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM*, 19(5):395–405, July 1976.

[Mil75] G. Miller. Riemann's hypothesis and tests for primality. *Proceedings of the Seventh Annual ACM Symposium on the Theory of Computing*, May 1975.

[Mil86] V. S. Miller. Use of elliptic curves in cryptography. In H.C. Williams, editor, *Advances in Cryptology - CRYPTO 1986, vol. 218 of Lecture Notes in Computer Science*, pages 417–426. Springer-Verlag, 1986.

[Mol04] R. A. Mollin. *RSA and Public-Key Cryptography*. Chapman and Hall / CRC, 2004.

[MvOV97] A.J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applies Cryptography*. CRC Press, 1997.

[NIoST80] National Institute of Standards and Technology. DES modes of operation. FIPS PUB 81, 1980. Voir: `http://www.itl.nist.gov/fipspubs/fip81.htm`.

[Pin93] R. Pinch. Carmichael numbers up to 10^{15}. *Mathematics of Computation*, 61(203):381–391, July 1993. Voir: `http://www.chalcedon.demon.co.uk/rgep/carpsp.html`.

[Pol74] J. M. Pollard. Theorems on factorization and primality testing. *Proceedings of the Cambridge Philosophical Society*, 76:521–528, 1974.

[Pol75] J. M. Pollard. A Monte Carlo method for factorization. *BIT*, 15:331–334, 1975.

[Pom82] C. Pomerance. Analysis and comparison of some integer factoring algorithms. In H.W. Lenstra and R. Tijdeman, editors, *Computational Methods in Number Theory, Part I*, 1982. Mathematical Tracts #154, Mathematisch Centrum, Amsterdam.

[Pom96] C. Pomerance. A Tale of Two Sieves. *Notices of the American Mathematical Society*, pages 1473–1485, December 1996. Voir: `http://www.ams.org/notices/199612/pomerance.pdf`.

[Rab80] M. Rabin. Probabilistic algorithms for primality testing. *Journal of Number Theory*, December 1980.

[RD09] G. Van Rossum and F. L. Drake. *Introduction To Python 3*. CreateSpace, 2009.

[RSA78] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, February 1978.

[San05] P. Sanchez. Pollard's rho. Planetmath.org, 2005. Voir: `http://planetmath.org/encyclopedia/PollardsRhoMethod.html`.

[Sch96] B. Schneier. *Applied Cryptography 2nd Edition*. Wiley, 1996.

[Sco04] Mike Scott. Multiprecision integer and rational arithmetic c/c++ library. Accessible en ligne, 2004. Voir: `http://www.shamus.ie/`.

[Sha48] C. Shannon. A Mathematical Theory of Information. *Bell Systems Technical Journal*, 27:379–423,623–656, July,October 1948. Voir: `http://cm.bell-labs.com/cm/ms/what/shannonday/paper.html`.

[Sha49] C. Shannon. Communication theory of secrecy systems. *Bell Systems Technical Journal*, 28(4):656–715, July 1949. Voir: `http://netlab.cs.ucla.edu/wiki/files/shannon1949.pdf`.

[Sin00] S. Singh. *The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography.* Anchor, 2000.

[Spi05] R. J. Spillman. *Classical and Contemporary Cryptology.* Pearson Prentice Hall, 2005.

[Sta03] W. Stallings. *Cryptography and Network Security, Principles and Practice, 3rd edition.* Prentice Hall, 2003.

[Sti06] D.R. Stinson. *Cryptography, Theory and Practice, 3rd edition.* Chapman & Hall / CRC, 2006.

[Sun03] Sun Microsystems. Class BigInteger. Java™ 2 Platform Standard Edition v1.4.2, 2003. Voir: `http://java.sun.com/j2se/1.4.2/docs/api/java/math/BigInteger.html`.

[Tan06] Chew Keong Tan. C# biginteger class. Accessible en ligne, 2006. Voir: `http://www.codeproject.com/csharp/biginteger.asp`.

[TW95] R. Taylor and A. Wiles. Ring-theoretic properties of certain Hecke algebras. *Annals of Mathematics*, 141:553–572, 1995.

[TW06] W. Trappe and L. C. Washington. *Introduction to Cryptography with Coding Theory, 2nd Edition.* Pearson Prentice Hall, 2006.

[USP06] USPTO. United States Patent and Trademark Office. Accessible en ligne, 2006. Voir: `http://www.uspto.gov/patft/index.html`.

[Ver26] G. S. Vernam. Cipher printing telegraph systems for secret wire and radio telegraphic communications. *Journal of the American Institute of Electrical Engineers*, XLV:109–115, 1926.

[VGM04] A. A. Vladimirov, K. V. Gavrilenko, and A. A. Mikhailovsky. *Wi-Foo, The Secrets of Wireless Hacking.* Addison-Wesley, 2004.

[Was08] L. C. Washington. *Elliptic Curves - Number Theory and Cryptography, 2nd Edition.* CRC Press, 2008.

[Wei83] A. Weil. *Number Theory, An Approach Through History.* Birkhäuser, 1983.

[Wil95] A. Wiles. Modular elliptic-curves and Fermat's last theorem. *Annals of Mathematics*, 141:443–551, 1995.

[Yan04] S. Yan. *Primality Testing and Integer Factorization in Public-Key Cryptography.* Springer-Verlag, 2004.