Cryptographie

CSUD, Option complémentaire informatique

Cédric Donner

Mars 2017

CHAPITRE 1

Table des matières

1.1 Introduction

Source

Ce document est largement repris du chapitre “Cryptosystèmes” de l’ouvrage TigerJython (http://www.tigerjython.ch/franz/index.php?inhalt_links=navigation.inc.php&inhalt_mitte=effizienz/kryptosysteme.inc.php)

Le principe du secret des données joue un rôle de plus en plus important dans notre société moderne car elle garantit le respect de la vie privée ainsi que la confidentialité des informations gouvernementales, industrielles ou militaires. Pour cela, il est nécessaire de crypter les données de telle sorte que si elles venaient à tomber entre les mains des fausses personnes, il soit impossible ou du moins très difficile de retrouver l’information originale sans que la méthode de cryptage soit compromise au préalable.

Durant **l’encodage**, les données originales sont transformées en données codées et lors du **décodage**, les données originales sont restaurées à partir des données cryptées. Si les données originales sont constituées de lettres de l’alphabet, on parle de **texte en clair** et de **cryptotexte**.

La description de la méthode utilisée pour effectuer le décryptage est appelée **clé** qui peut consister simplement en un nombre, une chaine de caractères numériques alphabétiques. Si la même clé est utilisée pour le codage et le décodage, on parle de **système à clé symétrique**. Si, en revanche, les clés de codage et de décodage sont différentes, on parle de méthode de **cryptographie asymétrique** (à clé publique).

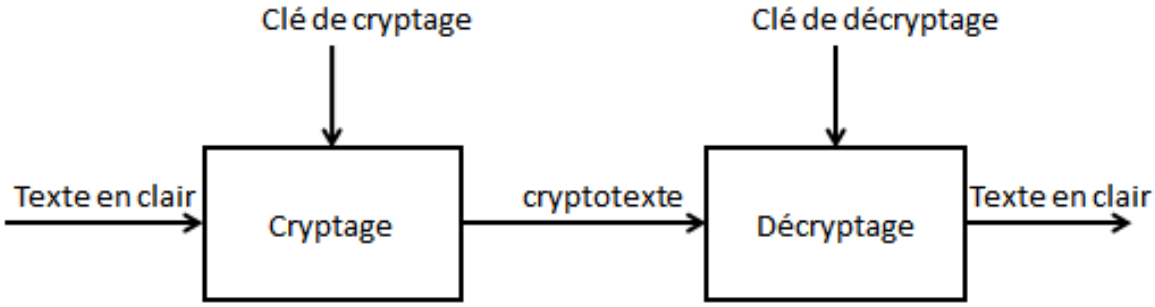

Fig. 1.1 – Fonctionnement général des algorithmes cryptographiques

Concepts informatiques
— Encodage

— décodage
— cryptographie symétrique/asymétrique
— Chiffre de César
— Chiffre de Vigenère
— cryptage RSA
— clé rivée/publique

1.2 Chiffre de César

D'après la tradition, Jules César (100 av. J.-C. - 44 av. J.-C.) utilisait déjà la méthode suivante dans ses communications militaires : chaque caractère du texte en clair était décalé dans l'alphabet d'un certain nombre de lettres en reprenant au début de l'alphabet une fois arrivé au bout.

Fig. 1.2 – Clé de chiffrement de César

Cette méthode utilisait deux anneaux imbriqués portant les lettres de l'alphabet. L'anneau intérieur était décalé d'un certain nombre de positions, ce qui constitue la clé de cryptage. Par exemple, si la clé vaut 3, le A sera codé par un D, le B par un E, le C par un F, le D par un G etc ...

Le programme ci-dessous lit le message à crypter depuis un fichier texte pour permettre de le modifier et de le partager facilement. Il faut écrire le texte en clair à l'aide d'un éditeur de code standard dans le fichier `original.txt` qu'il faut sauvegarder dans le même dossier que le programme Python. Il faut restreindre le message original aux lettres capitales, aux espaces et aux retours à la ligne. On aura par exemple

```
ON SE RETROUVE AUJOURD HUI A HUIT HEURES
BISOUS
TANIA
```

La fonction `encode(msg)` encode la chaine de caractères `msg` du message issu du fichier texte. Elle procède en remplaçant chaque caractère original par le caractère crypté correspondant, excepté le caractère de retour à la ligne `\n`.

```
import string
key = 4
alphabet = string.ascii_uppercase + " "

def encode(text):
    enc = ""
    for ch in text:
        if ch != "\n":
            i = alphabet.index(ch)
```

```
            ch = alphabet[(i + key) % 27]
        enc += ch
    return enc

fInp = open("original.txt")
text = fInp.read()
fInp.close()

print "Original:\n", text
krypto = encode(text)
print "Krypto:\n", krypto

fOut = open("secret.txt", "w")
for ch in krypto:
    fOut.write(ch)
fOut.close()
```

Voici le texte crypté :

```
SRDWIDVIXVSYZIDEYNSYVHDLYMDEDLYMXDLIYVIW
FMWSYW
XERME
```

Le décodage est implémenté de façon analogue à part le fait que les caractères sont décalés dans l'autre sens.

```
import string
key = 4
alphabet = string.ascii_uppercase + " "

def decode(text):
    dec = ""
    for ch in text:
        if ch != "\n":
            i = alphabet.index(ch)
            ch = alphabet[(i - key) % 27]
        dec += ch
    return dec

fInp = open("secret.txt")
krypto = fInp.read()
fInp.close()

print "Krypto:\n", krypto
msg = decode(krypto)
print "Message:\n", msg

fOut = open("message.txt", "w")
for ch in msg:
    fOut.write(ch)
fOut.close()
```

Memento

Notez qu'il faut conserver tous les caractères d'espacement du cryptotexte, même s'ils se trouvent au début ou à la fin de la ligne. Il est clair que cette méthode de cryptage peut être compromise très facilement. La manière la plus simple consiste simplement à tester toutes les 26 clés possibles jusqu'à l'obtention d'un texte en clair en français.

1.2.1 Exercices

Exercie 1 : force brute

Écrivez un programme qui casse qui déchiffre le cryptotexte suivant codé à l'aide du chiffre de César en testant successivement toutes les 27 clés possibles.

```
OAL SHDXTKMJXSASTVVXHLSL XSAFNALTLAGF
UMLSASVTFSGFDQSUXSL XJXSTLSXAZ L
ZJXXLAFZK
XNXDAFX
```

Exercie 2 : analyse de fréquences

Sachant que le message original a été écrit en anglais, déterminer la clé de chiffrement utilisée pour le cryptotexte ci-dessous. Décoder le message à l'aide de la clé trouvée.

Astuce : Utilisez une table de fréquence d'apparition des caractères dans la langue anglaise. Vous pouvez également construire une telle table en analysant un ouvrage en anglais contemporain.

https://en.wikipedia.org/wiki/Letter_frequency

```
..  comment::
```

1. Expliquer pourquoi le fait de ne pas crypter les espaces et les sauts de ligne affaiblit le chiffrement du message.
2. Modifier le chiffre ce César présenté pour qu'il encode également les caractères de séparation

1.3 Chiffre de Vigenère

Il est possible de renforcer le chiffre de César en appliquant un décalage alphabétique différent pour chacun des caractères du texte en clair. Cette substitution poly-alphabétique peut utiliser comme clé n'importe quelle permutation de 27 nombres. Il existe donc un nombre phénoménal de clés possibles, à savoir

$$27! = 10'888'869'450'418'352'160'768'000'000 \approx 10^{27}$$

Fig. 1.3 – Blaise Vigenère (1523 - 1596)

Il est cependant un peu plus aisé d'utiliser un mot secret auquel on fait correspondre une liste de nombres correspondant à la position de chacun de ses caractères dans l'alphabet. Ainsi, le mot secret ALICE correspond à la liste `[0, 11,`

`8, 2, 4]`. La clé de cryptage, de même longueur que le message à crypter, est construite à partir d'une juxtaposition répétée des décalages `[0, 11, 8, 2, 4]` correspondant au mot secret ALICE. Lors de l'encodage, les caractères du texte en clair sont décalés selon le nombre correspondant de la clé de cryptage.

L'illustration suivante permet de mieux comprendre le fonctionnement de la méthode de Vigenère :

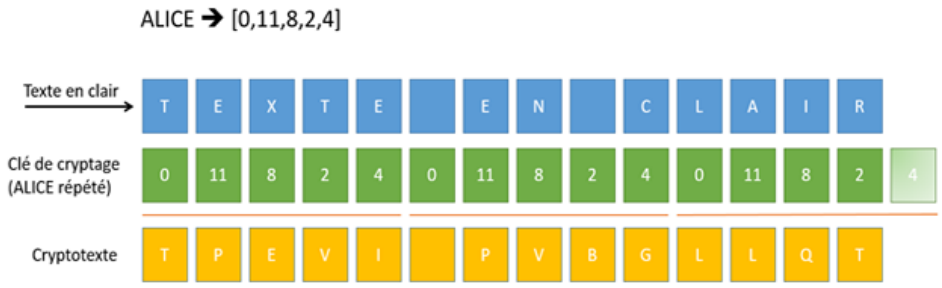

Fig. 1.4 – Fonctionnement du chiffre de Vigenère

Et voici le code permettant d'encoder un texte lu depuis le fichier `original.txt` :

```
import string
key = "ALICE"
alphabet = string.ascii_uppercase + " "

def encode(text):
    keyList = []
    for ch in key:
        i = alphabet.index(ch)
        keyList.append(i)
    print "keyList:", keyList
    enc = ""
    for n in range(len(text)):
        ch = text[n]
        if ch != "\n":
            i = alphabet.index(ch)
            k = n % len(key)
            ch = alphabet[(i + keyList[k]) % 27]
        enc += ch
    return enc

fInp = open("original.txt")
text = fInp.read()
fInp.close()

print "Original:\n", text
krypto = encode(text)
print "Krypto:\n", krypto

fOut = open("secret.txt", "w")
for ch in krypto:
    fOut.write(ch)
fOut.close()
```

Le décodeur est à nouveau pratiquement identique à l'encodeur excepté le sens de décalage.

```
import string
key = "ALICE"
```

```
alphabet = string.ascii_uppercase + " "

def decode(text):
    keyList = []
    for ch in key:
        i = alphabet.index(ch)
        keyList.append(i)
    print "keyList:", keyList
    enc = ""
    for n in range(len(text)):
        ch = text[n]
        if ch != "\n":
            i = alphabet.index(ch)
            k = n % len(key)
            ch = alphabet[(i - keyList[k]) % 27]
        enc += ch
    return enc

fInp = open("secret.txt")
krypto = fInp.read()
fInp.close()

print "Krypto:\n", krypto
msg = decode(krypto)
print "Message:\n", msg

fOut = open("message.txt", "w")
for ch in msg:
    fOut.write(ch)
fOut.close()
```

Memento

Le chiffre de Vigenère fut inventé au 16e siècle par Blaise de Vigenère et fut considéré comme très sûr pendant de nombreux siècles. Si quelqu'un entre en possession du cryptotexte et sait que la longueur du mot secret est 5, il lui faut néanmoins essayer systématiquement $26^5 = 11'881'376$ clés différentes à moins qu'il connaisse une information supplémentaire au sujet du mot secret, comme le fait qu'il s'agit d'un prénom féminin.

1.3.1 Exercices

1. Expliquer en quoi le chiffre de César est un cas particulier de chiffre de Vigenère.
2. Faire une recherche sur le Web concernant le système de cryptage Skytale et implémenter un encodeur / décodeur basé sur ce principe.
3. Expliquer ce qu'est l'analyse de fréquence et comment cette méthode peut être utilisée pour casser le code de Vigenère si l'on connaît la langue dans laquelle est écrit le message original.
4. Expliquer pourquoi le chiffre de Vigenère n'est pas considéré comme sûr du tout actuellement.

1.4 Méthode RSA

1.4.1 Fonctionnement

Dans cette méthode dont le nom provient de celui de ses inventeurs, Rivest, Shamir et Adleman, on utilise une paire de clés, à savoir une clé privée et une clé publique. Il s'agit donc d'une méthode de cryptographie asymétrique.

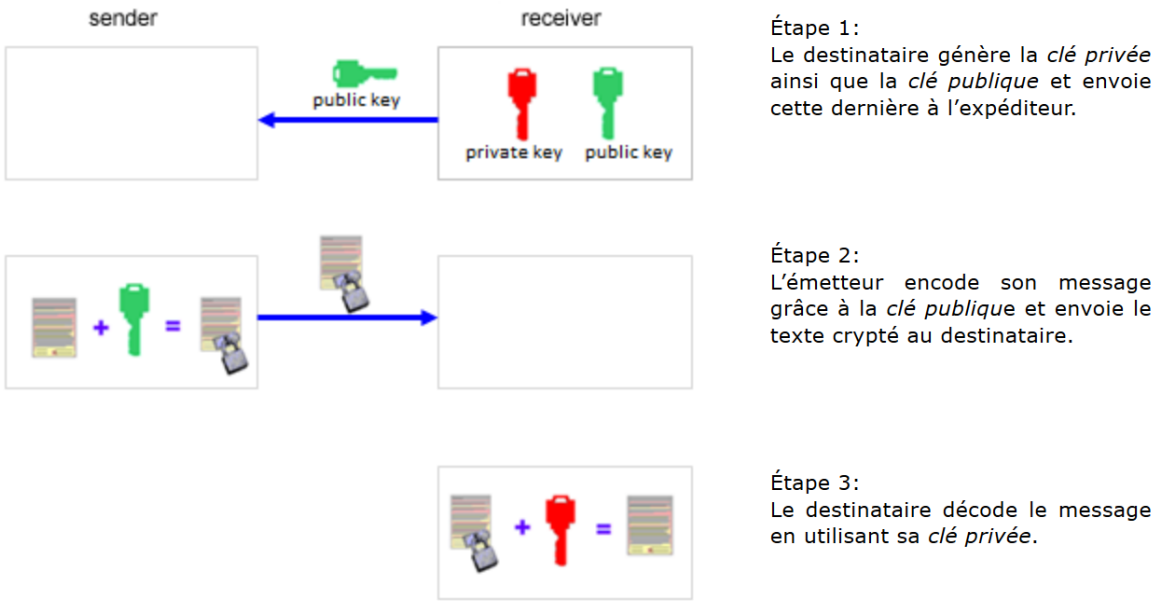

Fig. 1.5 – Fonctionnement du chiffre RSA

Les clés publiques et privées sont générées en utilisant l'algorithme suivant basé sur des résultats de la théorie des nombres. On en trouve une preuve élémentaire dans le livre Barth, *Algorithmics for beginners*, Springer-Verlag.

On commence par choisir deux nombres premiers p et q qui doivent comporter un très grand nombre de chiffres pour garantir la sécurité du cryptage. On multiplie ensuite ces deux nombres pour obtenir $m = p \cdot q$. On sait de la théorie des nombres que l'indicatrice d'Euler $\phi(m) = (p-1) \cdot (q-1)$ correspond au nombre d'entiers n inférieurs à m qui sont copremiers avec m.

Astuce : Des nombres a et b sont dits **copremiers** ou **premiers entre eux** si $\gcd(m, n) = 1$.

Le module `fractions` de Python permet de calculer le PGDC = GCD (Greatest Common Divisor) de deux nombres.

```
>>> from fractions import gcd
# 20 et 8 ne sont pas copremiers ...
>>> gcd(20,8)
4
# 10800 et 11 sont copremiers
>>> gcd(10800, 11)
1
```

On choisit ensuite un nombre e inférieur à $\phi(m)$ tel que e et $\phi(m)$ sont premiers entre eux. La paire de nombres (m, e) constitue déjà la clé publique :

Clé publique : `[m, e]`

1.4.2 Exemple

Voici un exemple de calcul de la clé publique pour les petits nombres premiers $p = 73$ et $q = 15$:

— $m = 73 \cdot 151 = 11023$
— $\phi = 72 \cdot 150 = 10800$
— $e = 11$ (choisi copremier avec ϕ)

Clé publique :

```
[m, e] = [11023, 11]
```

La clé privée est quant à elle formée de la paire de nombres :

Clé privée

```
[m, d]
```

Où d est en entier tel que $d \cdot e \pmod{\phi} = 1$.

(Puisque e et $\phi(m)$ sont premiers entre eux, l'identité de Bézout affirme que cette équation possède nécessairement au moins une solution).

On peut déterminer le nombre d à partir des valeurs de e et ϕ à l'aide d'un simple programme qui va tester 100'000 valeurs pour d au sein d'une boucle :

```
1 e = 11
2 phi = 10800
3
4 for d in range(100000):
5     if  (d * e) % phi == 1:
6         print "d", d
```

On obtient ainsi plusieurs solutions : (5891, 16691, 27491, 49091, etc.). Comme il suffit d'une seule valeur pour déterminer la clé privée, la première rencontrée fait l'affaire.

Clé privée

```
[m, d] = [11023, 5891]
```

Sécurité de RSA

Dans le cas présent, il est très facile de déterminer la clé privée uniquement à cause du fait que l'on connaît les nombres premiers p et q et, de ce fait, la valeur de ϕ. Cependant, sans la connaissance de ces nombres, il faut une puissance de calcul énorme pour calculer la clé privée.

1.4.3 Encodage de textes

L'algorithme RSA est utilisé pour encoder des nombres. Pour encoder du texte, on utilise donc le code ASCII de chacun des caractères du message en clair qui sera crypté à l'aide de la clé publique `[m, s]` selon la formule

$$s = r^e \pmod{m}.$$

Le programme suivant écrit chacun des caractères encodés sur une nouvelle ligne dans le fichier `secret.txt`.

```
publicKey = [11023, 11]

def encode(text):
    m = publicKey[0]
    e = publicKey[1]
    enc = ""
    for ch in text:
        r = ord(ch)
        s = int(r**e % m)
        enc += str(s) + "\n"
    return enc

fInp = open("original.txt")
text = fInp.read()
fInp.close()

print "Original:\n", text
krypto = encode(text)
print "Krypto:\n", krypto

fOut = open("secret.txt", "w")
for ch in krypto:
    fOut.write(ch)
fOut.close()
```

Le décodeur commence par charger ligne à ligne les nombres présents dans le fichier `secret.txt` pour les stocker dans une liste. Pour chacun des nombres présents dans cette liste, le nombre original est calculé à l'aide de la clé privée s selon la formule

$$r = s^d \pmod{m}.$$

Ce nombre correspond au code ASCII du caractère original.

```
privateKey = [11023, 5891]

def decode(li):
    m = privateKey[0]
    d = privateKey[1]
    enc = ""
    for c in li:
        s = int(c)
        r = s**d % m
        enc += chr(r)
    return enc

fInp = open("secret.txt")
krypto = []
while True:
line = fInp.readline().rstrip("\n")
if line == "":
    break
krypto.append(line)
fInp.close()

print "Krypto:\n", krypto
msg = decode(krypto)
print "Message:\n", msg

```

```
fOut = open("message.txt", "w")
for ch in msg:
    fOut.write(ch)
fOut.close()
```

Memento

Le gros avantage du code RSA réside dans le fait que l'émetteur et le récepteur n'ont pas besoin de procéder à un échange d'information secrète avant de procéder au cryptage. Au lieu de cela, le destinataire génère la clé privée et la clé publique et ne transmet que la clé publique à l'émetteur tout en gardant bien au chaud sa clé privée. L'émetteur va alors crypter son message à l'aide de la clé publique du destinataire en sachant que seul le destinataire sera capable de décrypter le message puisqu'il est le seul à disposer de la clé privée nécessaire à cette opération [plus...].

En pratique, on choisit de très gros nombres premiers p et q comportant des centaines de chiffres. Générer la clé publique ne nécessite que le produit $m = p \cdot q$, ce qui est une banalité pour l'ordinateur. Si un pirate veut déterminer la clé privée à partir de la clé publique, il lui faut déterminer les nombres premiers originaux p et q à partir de m, ce qui revient à factoriser le nombre m en ses deux facteurs premiers. Or, la factorisation de très grands nombres premiers n'est à ce jour possible qu'en utilisant une puissance de calcul absolument colossale. On voit de ce fait que le cryptosystème RSA repose sur les limites de la calculabilité. Il ne faut cependant jamais oublier qu'il n'existe en principe aucune méthode de cryptage parfaitement incassable. Heureusement, il suffit que l'opération de décryptage soit considérablement plus longue que la période de validité ou de pertinence de l'information pour que la méthode soit considérée comme étant suffisamment sûre.

1.4.4 Compléments

Il faut noter quelques compléments par rapport au chiffre présenté ci-dessus.

- La méthode RSA, telle que présentée ci-dessus, n'est pas considérée comme sûre car deux messages vont toujours donner lieu au même cryptotexte. En pratique, on utilise encore des techniques de modification du texte à crypter avant de le crypter (padding OAEP).
- On n'utilise rarement RSA pour crypter de longs textes. L'avantage de RSA est de ne pas nécessiter d'échange de la clé de cryptage. Son désavantage est d'être relativement coûteux en temps de calcul pour le cryptage et le décryptage. En pratique, on utilise des méthodes symétriques pour crypter de grandes quantités de données (méthode AES). Seule la clé de cryptage / décryptage symétrique est à ce moment transférée de manière sécurisée par RSA.

Avertissement : Il ne faut jamais tenter d'implémenter soi-même un algorithme cryptographique "maison" à moins d'être un très grand spécialiste du sujet. Même s'il est relativement facile de comprendre le principe de base du fonctionnement de RSA ou AES, le diable se cache dans les détails d'implémentation des algorithmes.
En pratique, des communications cryptées avec RSA ou d'autres protocoles cryptographiques modernes et considérés comme sécurisées ont déjà été compromises en raison de faiblesses d'implémentation et de programmation. Voici l'une des plus célèbres vulnérabilités détectée dans le protocole OpenSSL et publiée en mars 2012. Elle a affecté gravement la sécurité de plusieurs sites Internet majeurs pendant plusieurs mois : https://fr.wikipedia.org/wiki/Heartbleed.
En résumé, il est conseillé de toujours utiliser une bibliothèque cryptographique éprouvée et testée de manière extensive par la communauté. Python met à disposition de telles bibliothèques au travers du module `pycrypto` (https://pypi.python.org/pypi/pycrypto).

1.4.5 Exercices

1. Expliquer comment l'on peut utiliser RSA pour signer électroniquement des documents.
2. Utiliser la méthode RSA pour générer une clé publique et une clé privée à l'aide de deux nombres premiers p et q tous deux inférieurs à 100 et procéder au cryptage / décryptage d'un message de votre choix

3. Alice intercepte une communication contenant la clé publique de Bob : `[m, e] = [97213511, 6551]`. Malheureusement, Bob n'a pas bien suivi le cours d'OC et a implémenté lui-même son algorithme RSA en utilisant des nombres premiers trop petits pour générer sa paire de clés. Aidez Alice à retrouver la clé privée de Bob et à décoder le message suivant qui a été encodé avec cette clé publique :

   Astuce : Le texte crypté figure uniquement sur le site car il prendrait trop de place dans ce document vu que chaque nombre est codé sur une ligne séparée.

   ---

4. Expliquez comment l'on fait en pratique pour choisir des nombres premiers p et q très grands et les conditions que ces nombres doivent respecter.

   Astuce : Mot clé : tests de primalité probabilistes, test de Miller-Rabin

   ---

1.5 Références

Voici quelques ressources vous permettant de creuser davantage le sujet de la cryptograhie :

1.5.1 Articles en français

— Présentation de RSA et lien avec les nombres premiers : https://interstices.info/jcms/c_30225/nombres-premiers-et-cryptologie-l-algorithme-rsa
— Attaques sur les codes secrets : https://interstices.info/jcms/i_53837/a-l-attaque-des-codes-secrets

1.5.2 Articles en anglais

— Présentation plus détaillée du fonctionnement de RSA : http://doctrina.org/How-RSA-Works-With-Examples.html
— Explique pourquoi RSA fonctionne, très mathématique : http://doctrina.org/Why-RSA-Works-Three-Fundamental-Questions-Answered.html
— Explique le padding OAEP nécessaire pour sécuriser RSA : https://en.wikipedia.org/wiki/Optimal_asymmetric_encryption_paddi

1.5.3 Cours en ligne

— Excellent cours Udacity sur la cryptograhie : https://www.udacity.com/course/applied-cryptography–cs387