Chiffrements symétrique et asymétrique

La cryptographie n'est pas seulement l'apanage des militaires. Dans le domaine informatique, la sécurisation données est crucial lors de l'échange d'informations par exemple lors d'un paiement sur internet ou encore pour renseigner des informations personnelles (arrêt maladie, impôts, opinion personnel ...).

1. Principes de chiffrement

Le principe de base de la sécurisation d'une communication consiste à modifier la données qui doit être transmise (le chiffrement), de façon à ce que toute personne qui l'intercepterait ne pourrait pas en comprendre le sens. Seul le destinataire va pouvoir retrouver la donnée initiale (le déchiffrement) et la lire.

Les méthodes de chiffrement sont basées sur l'utilisation de clés (chaîne de caractères) qui vont, par l'application d'un algorithme, chiffrer ou déchiffrer le message. Il existe deux principaux types de chiffrement de données qui permettent de rendre un message lisible uniquement par son destinataire:

- ✗ le chiffrement symétrique à clé partagée,
- ✗ le chiffrement asymétrique avec une paire clé publique clé privée.

2.Chiffrement symétrique

Principe du chiffrement XOR

Cette technique repose sur l'utilisation d'une clé unique qui doit être connue par l'expéditeur et le destinataire. Il existe de nombreux chiffrements (code de César, Chiffrement de Vigenère etc.) de ce type qui ont évolué en complexité au cours du temps.

Une méthode de chiffrement symétrique consiste à chiffrer le message par XOR (OU exclusif). Celle ci repose sur l'utilisation de l'opération logique OU exclusif noté $\oplus$.

Cet opérateur a la particularité d'être sa propre réciproque, ce qui n'est pas le cas du ET ni du OU. C'est à dire que :

$$(a \oplus b) \oplus b = a$$

$S = a \oplus b$

a	b	S
0	0	0
0	1	1
1	0	1
1	0	0

Figure 1: Table de vérité du OU exclusif

Q1. **Démontrer** la propriété précédente à l'aide de tables de vérité.

$S = a \oplus b$ $\quad S_2 = \overline{(a \oplus b)} \oplus b$

a	b	S
0	0	0
0	1	1
1	0	1
1	0	0

S	b	S_2
0	0	0
1	1	0
1	0	1
0	1	1

Identique

Le chiffrement par XOR consiste donc à effectuer une opération XOR bit à bit entre un message et une clé. Par exemple le chiffrement du mot **NSI** avec la clé **CLE** consiste à représenter en binaire les deux chaînes de caractères selon la table ASCII puis d'effectuer l'opération XOR bit à bit. Le message crypté correspond alors à la chaîne de caractère correspondant au résultat binaire obtenu.

Q2. **Déterminer** le résultat du chiffrement du mot **NSI** avec la clé **CLE** dans le cas d'un chiffrement symétrique par XOR. **Vérifier** que le déchiffrement du message chiffré permet d'obtenir le message initial **NSI**.

```
⊕ 0100 1110  0101 0011  0100 1001  ← 'NSI'
  0100 0011  0100 1100  0100 0101  ← 'CLE'
  ---------------------------------
  0000 1101  0001 1111  0000 1100  → $ 0D 1F 0C
    0    D     1    F     0    C
```

Déchiffrement :

```
⊕ 0000 1101  0001 1111  0000 1100  ← $ 0D 1F 0C
  0100 0011  0100 1100  0100 0101  ← 'CLE'
  ---------------------------------
  0100 1110  0101 0011  0100 1001  → 'NSI'
```

Programmation du chiffrement symétrique XOR

Dans le cas général, la clé peut être de taille différente du message. Dans le cas où la clé est plus courte que le message, il faut alors reproduire la clé vers la droite autant de fois que nécessaire. Si la taille du message n'est pas un multiple de la taille de la clé, on peut reproduire seulement quelques bits de la clé pour la fin du message.

A partir des fonctions natives Python ci-contre, programmer et tester une fonction `chiffrement_XOR(message, cle)` dont le prototype est les suivant :

```
def chiffrement_XOR(message : str, cle : str)
    ''' chiffre le message avec cle par un
    chiffrement symétrique XOR
    cle doit être de taille inférieure ou égale
    à message
    exemple :
    >>> chiffrement_XOR('NSI', 'CLE')
    '\r\x1f\x0c'
    '''
```

Fonctions natives utiles

`ord(str)` : retourne le code ascii d'un caractère → `ord('N')` retourne `78`

`chr(int)` retourne le caractère ascii d'un entier → `chr(78)` retourne `'N'`

`^` : opérateur OU exclusif bit à bit → `10 ^ 9` retourne `3` (1010 ^ 1001 = 0011)

Q3. **Justifier** le résultat obtenu dans l'exemple présenté dans la spécification de la fonction

Les octets obtenus ne correspondent pas à des lettres dans la table ASCII, l'interpréteur renvoie la valeur hexa pour les deux derniers octets (\x → octet codé en hexa) et le premier octet qui correspond au caractère 'Carriage Return'

Q4. **Donner** le chiffrement de `'SPECIALITE NSI'` avec la clé `'TERM'`. **Proposer** une méthode pour déchiffrer le message chiffré

```
>>> chiffrement_XOR('SPECIALITE NSI', 'TERM')
'\x07\x15\x17\x0e\x1d\x04\x1e\x04\x00\x00r\x03\x07\x0c'
```

Le déchifrage s'obtient en appliquant le même algorithme sur le message chiffré avec la même clé :

```
>>> chiffrement_XOR('\x07\x15\x17\x0e\x1d\x04\x1e\x04\x00\x00r\x03\x07\x0c', 'TERM')
'SPECIALITE NSI'
```

Conclusion

L'opérateur OU exclusif est l'une des opérations de base implémentée dans l'unité arithmétique et logique d'un microprocesseur. Ces caractéristiques font que cet opérateur est une brique de base couramment utilisée dans les algorithmes de chiffrement modernes même si appliquée naïvement comme dans l'exercice précédent, elle n'apporte pas une grande sécurité.

Parmi les algorithmes de chiffrement symétriques les plus utilisés, on peut citer AES (advances Encryption Standard) et ChaCha20. Bien que très complexes ces algorithmes reposent sur des principes similaires au chiffrement XOR.

3.Chiffrement asymétrique

Malgré les nombreux avantages du chiffrement symétrique, ce dernier possède un défaut important. Si deux personnes veulent établir un canal de communication sûr en utilisant une méthode de chiffrement symétrique, les deux personnes doivent se mettre d'accord sur la clé de chiffrement. Ceci impose d'envoyer la clé par un canal non sécurisé.

Le chiffrement asymétrique fut inventé par Whitfield Diffie et Martin Hellman en 1976, qui reçurent le prix Turing de 2015 pour cette découverte. Dans ce chiffrement les deux interlocuteurs n'ont pas besoin de partager une "clé secrète". Il y a 2 clés :

- **la clé publique** : Celle-ci, tout le monde peut la posséder, il n'y a aucun risque, elle peut être transmise à n'importe qui. La clé publique sert à chiffrer le message.
- **la clé privée** : Seul le récepteur la possède. Elle sert à déchiffrer le message chiffré avec la clé publique.

Un algorithme de chiffrement asymétrique très utilisé se nomme RSA. Dans cette partie, nous allons utiliser une version plus simple nommée `kidRSA` : le RSA pour les "enfants".

Principe de l'algorithme kidRSA

A partir de quatre entiers a1, b1, a2, b2, on calcule les entiers suivants :

$M = a1 \times b1 - 1$

$e = a2 \times M + a1$

$d = b2 \times M + b1$

$n = (e \times d - 1)/M$

➔ La clef publique est le tuple `(e,n)` et la clef secrète est le tuple `(d,n)` .

➔ Pour chiffrer un message représenté par un entier `m` plus petit que `n`, on effectue l'opération e x m (modulo n).

➔ Pour déchiffrer un message représenté par un entier `m` plus petit que `n`, on effectue l'opération d x m (modulo n).

Un exemple (avec des petits nombres)

Prenons a1=5, b1=3, a2=7 et b2=5.

Q5. **Calculer** `M`, e, d et n. **En déduire** la clef publique et la clef secrète.

M=a1×b1−1 = 3×5-1 = 14

e=a2×M+a1 = 7×14+5 = 103

d=b2×M+b1 = 5×14+3 = 73

n=(e×d−1)/M = (103×73-1)/14 = 537

Clé publique : (103, 537)

Clé privée = (73, 537)

Q6. **Donner** le message chiffré par la clef publique représenté par le code ASCII de la lettre `'a'` (en minuscule). **Montrer** que si on chiffre le message avec la clef secrète on retrouve bien le code ASCII de la lettre `'a'`.

‘a’ = 97 → messageChiffré = (97×103) modulo 537

messageChiffré = 325

messageClair = (325×73) modulo 537

messageClair = 97 → ‘a’

Implémentation en Python du chiffrement avec l'algorithme KidRSA

Q7. **Implémenter** les fonctions Python suivantes :

- `genere_clefs_publique_et_privee(a1,b1,a2,b2)` : génère et retourne la clef publique `(e,n)` et la clef secrète `(d,n)` à partir des 4 entiers passés en paramètre `a1`, `b1`, `a2`, `b2`
- `chiffre_message(m,clef)` : chiffre un message `m` qui est une chaîne de caractères avec la clef `clef`, en remplaçant chaque caractère par son code ASCII en décimal.

  Le message chiffré retourné est une liste de nombres. La taille de la liste étant égale à la longueur de la chaîne de caractères `m`.
- `dechiffre_message(m,clef)` : déchiffre un message `m` qui est une liste de nombres et renvoie le message déchiffré sous la forme d'une chaîne de caractères.

Tests de cette implémentation en Python de KidRSA

On choisit comme valeurs a1=13, b1=32, a2=69 et b2=35

Q8. **Donner** à partir des fonctions Python codées pour l'algorithme KidRSA :

➔ la valeur de la clef publique : (28648, 1004889)

➔ la valeur de la clef secrète : (14557, 1004889)

➔ la liste des 3 nombres correspondant au chiffrement de la chaîne de caractères `"NSI"` avec la clef publique. [224766, 368006, 81526]

Q9. **Montrer** que si on déchiffre avec la clef secrète le message `NSI` préalablement chiffré avec la clef publique, on retrouve bien la chaîne `"NSI"`.

```
>>> dechiffre_message([224766, 368006, 81526],(14557, 1004889))
'NSI'
```

Casser (ou décrypter) le chiffrement KidRSA

Un message a été chiffré avec KidRSA à l'aide de la clef publique suivante :

(e,n) = (53447,5185112).

Voici le message chiffré obtenu :

[3580949, 2084433, 3687843, 4436101, 4489548, 1710304, 4329207, 4542995, 3901631, 1710304, 4061972, 3687843, 1710304, 3527502, 4222313, 4436101, 4436101, 1710304, 3687843, 4168866, 1710304, 4168866, 4436101, 3901631, 1710304, 3367161]

On souhaite "casser" le message chiffré et retrouver le message en clair. Pour cela, on a besoin de connaître la clef secrète (n,d). Comme on connaît déjà n (n=5185112), il faut trouver une méthode pour calculer d.

En étudiant la relation entre les nombres qui constituent les clefs publique et privées e,d et n : $n=e\times d-1M$, qui peut s'écrire aussi : $e\times d-1=n\times M$. On en déduit que $e\times d-1$ est divisible par n.

Pour trouver l'entier d qui fait partie de la clef secrète, comme on connaît déjà n et e, il suffit d'étudier parmi toutes les valeurs de d comprises entre 1 et n−1, laquelle vérifie la condition e×d−1 est divisible par n

On appelle ce type d'attaque par **force brute** car on doit étudier (dans le pire des cas) tous les entiers d inférieurs à n. Ce qui peut être long si n est grand.

Q10. **Ecrire** le corps de la fonction `bruteForceKidRSA(e,n)` qui permet de calculer et de retourner le premier entier d inférieur à n qui vérifie la relation e×d−1 est divisible par n

Q11. **En déduire** la valeur de d et obtenir ainsi le déchiffrement du message chiffré donné plus haut.

```
>>> bruteForceKidRSA(53447,5185112)
(323639, 5185112)
```

d est donc égal à 323639

```
>>> dechiffre_message([3580949, 2084433, 3687843, 4436101, 4489548, 1710304,
4329207, 4542995, 3901631, 1710304, 4061972, 3687843, 1710304, 3527502,
4222313, 4436101, 4436101, 1710304, 3687843, 4168866, 1710304, 4168866,
4436101, 3901631, 1710304, 3367161] ,(323639, 5185112))
"C'EST QUI LE BOSS EN NSI ?"
```

Le message codé est "C'EST QUI LE BOSS EN NSI ?"

Attaque de KidRSA à l'aide de l'algorithme d'Euclide étendu

La réponse au message chiffré précédent est aussi chiffrée avec KidRSA mais avec une clef publique de longueur beaucoup plus grande.

Voici la clef publique utilisée : (e,n) = (230884490440319, 194326240259798261076).

Et voici le message chiffré obtenu avec la clef publique :

[16623683311702968, 19625181687427115, 16392798821262649, 16392798821262649, 20548719649188391, 7388303694090208, 17547221273464244, 15931029840382011, 19163412706546477, 7388303694090208, 15238376369061054, 18239874744785201, 18008990254344882, 19163412706546477, 7388303694090208, 19394297196986796, 19625181687427115, 20548719649188391, 15007491878620735, 19625181687427115, 20317835158748072, 7388303694090208, 7619188184530527]

Q12. **Essayer** de retrouver la clef secrète à l'aide de la fonction `bruteForceKidRSA(e,n)`. **Décrire** le problème rencontré.

Une méthode mathématique qui permet de résoudre ce problème se nomme l'*algorithme d'Euclide étendu.*

Cet algorithme d'Euclide étendu est basé sur une suite de divisions euclidiennes que nous ne détaillerons pas ici. On peut retenir que la clef secrète d est l'inverse entier de e modulo n et que l'algorithme d'Euclide étendu résout de façon très efficace cette relation. La complexité algorithmique par force brute est dans le pire des cas en $O(n)$ alors que celle avec l'algorithme d'Euclide étendu est en $O(log(n))$.

Voici deux fonctions données en pseudo-code qui permettent de trouver l'inverse d'un entier modulo un autre entier.

```
fonction modinv(e,n)
        g,x,y = egcd(e,n)
        si g != 1 alors
            retourner Faux
        sinon
            retourner x % n
```

```
fonction egcd(a,b)
        Si a = 0 alors
            retourner (b,0,1)
        sinon
            g,y,x = egcd(b % a,a)
            retourner (g, x - (b//a)*y,y)
```

Q13. **Coder** ces deux fonctions en Python. **En déduire** la clef privée secrète associée à la clef publique : (230884490440319, 194326240259798261076). **En déduire** le décodage du deuxième message chiffré avec cette clef publique.

```
>>> e,n = 230884490440319, 194326240259798261076
>>> d = modinv(e,n)
>>> d
158674832848565541575
>>> deuxiemeMessage = dechiffre_message([16623683311702968,
19625181687427115, 16392798821262649, 16392798821262649,
20548719649188391, 7388303694090208, 17547221273464244,
15931029840382011, 19163412706546477, 7388303694090208,
15238376369061054, 18239874744785201, 18008990254344882,
19163412706546477, 7388303694090208, 19394297196986796,
19625181687427115, 20548719649188391, 15007491878620735,
19625181687427115, 20317835158748072, 7388303694090208,
7619188184530527] , (d, n))
>>> deuxiemeMessage
'HUGGY LES BONS TUYAUX !'
```

Etude de la vulnérabilité de l'algorithme RSA

Nous venons de vérifier que l'algorithme KidRSa pouvait être facilement "cassé" même avec des clefs assez grande à l'aide de l'algorithme d'Euclide étendu.

Heureusement, le véritable algorithme **RSA** utilisé par HTTPS sur internet est bien plus robuste.

Q14. **Donner** la taille des clefs couramment utilisées par RSA pour sécuriser des données sur Internet. **Donner** aussi quelle nouvelle technologie pourrait permettre de casser RSA en quelques secondes.

Les clés RSA sont codées sur 1 024 ou 2 048 bits (2^{2048} = 323170060713110073007148766886699519604441026697154840321303454275246551388678908931972014115229134636887179609218980194941195591504909210950881523864482831206308773673009960917501977503896521067960576383840675682767922186426197561618380943384761704705816458520363050428875758915410658086075523991239303855219143333896683424206849747865645694948561760353263220580778056593310261927084603141502585928641771167259436037184618573575983511523016459044036976132332872312271256847108202097251571017269313234696785425806566979350459972683529986382155251663894373355436021354332296046453184786049521481935558536110595962306 56)

Les nombres mis en œuvre sont donc très grands, le code est donc impossible à casser en un temps acceptable. Cependant on pense que grâce à l'informatique quantique, l'utilisation de algorithme quantique de Shor permettrait de casser ce chiffrement RSA.