TP – Sécurisation des communications

Objectifs :

- ⇨ Mettre en œuvre de méthodes de chiffrement symétrique simple
- ⇨ Utiliser une bibliothèque de chiffrement
- ⇨ Mettre en œuvre une transmission d'information par chiffrement symétrique RSA

I – Cryptage symétrique

1) Exemple de cryptage simpliste : Le chiffre de César

Ce cryptage, baptisé ainsi car il fut utilisé par Jules César pour chiffrer sa correspondance est un chiffrement par substitution simple : A chaque lettre du message original, on substitue une autre lettre de l'alphabet.
La lettre substituée est même que la lettre à chiffrer, mais décalée d'un nombre n dans l'alphabet. On considère ici que la lettre 'z' est suivie de la lettre 'a' (rebouclage de l'alphabet).

Exemple :
Message à chiffrer : `LA PHYSIQUE C EST FANTASTIQUE` avec n = 3

L	A		P	H	Y	S	I	Q	U	E		C		E	S	T		F	A	N	T	A	S	T	I	Q	U	E	
															+3															
O	D		S	K	B	V	L	T	X	H		F		H	V	W		I	D	Q	W	D	V	W	L	T	X	H	

On voit bien que le message chiffré est effectivement incohérent s'il n'est pas déchiffré. Par contre cette méthode a plusieurs inconvénients :
On ne peut pas chiffrer les caractères non alphabétiques comme les espaces ou les apostrophes. On peut soit les supprimer (cela rend le message déchiffré plus difficile à lire), soit les laisser tels quels (cela rend le message chiffré plus facile à deviner).
Il n'y a que 25 clés possibles et l'algorithme étant très simple, on peut très facilement trouver la clé par essai de toutes les clés (on parle de « force brute »).

Question 1 :

1) Ecrire une fonction de chiffrement par l'algorithme du chiffre de César dont la signature sera :
`str chiffreDeCesar(str phrase, int n)` où `phrase` est la phrase à chiffrer et `n` la clé de chiffrement (nombre de 1 à 25). La valeur de retour est la phrase chiffrée.
Tester la fonction en écrivant un programme principal qui l'appelle avec comme argument ceux de l'exemple précédent.
3) Modifier le programme pour qu'il demande la clé de chiffrement (une valeur entre 1 et 25), la phrase à chiffrer, puis qu'il affiche la phrase chiffrée par l'algorithme du chiffre de César.
4) Le ROT13 est un cas particulier de cette méthode de chiffrement pour laquelle la clé vaut 13. En quoi ce cas est-il particulier ? Utiliser cette particularité pour tester votre programme et vérifier qu'il fonctionne bien.
5) On souhaite rajouter la possibilité de décoder un message. Quelle astuce peut-on employer pour ré-utiliser la fonction précédemment écrite afin de décoder ? Mettre cette astuce en œuvre et vérifier qu'elle fonctionne.

Pour cet exercice, on pourra s'aider de la page web suivante qui traite des chaînes de caractères et des méthodes et fonction connexes. Ne la lisez pas en entier, mais parcourez-la pour savoir ce qu'il est possible de faire et revenez-y pour retrouver la syntaxe exacte d'une fonctionnalité dont vous avez besoin.
https://python.sdv.univ-paris-diderot.fr/10_plus_sur_les_chaines_de_caracteres/

On utilisera en particulier les méthodes `.isalpha()` pour tester si la lettre à coder est alphabétique (si ce n'est pas le cas on insère un espace à sa place dans la phrase chiffrée) et `.upper()` pour convertir le texte en majuscule et ainsi travailler uniquement avec les 26 lettres majuscules du code ASCII.

2) Un chiffrement plus avancé : le chiffre de Vigenère

Le chiffre de Vigenère est un système de chiffrement polyalphabétique. C'est un chiffrement par substitution, mais une même lettre du message en clair peut, suivant sa position dans celui-ci, être remplacée par des lettres différentes, contrairement à un système de chiffrement monoalphabétique comme le chiffre de César (qu'il utilise cependant comme composant). Cette méthode résiste ainsi à l'analyse de fréquences[1], ce qui est un avantage décisif sur les chiffrements monoalphabétiques. Cependant le chiffre de Vigenère a été percé par le major prussien Friedrich Kasiski qui a publié sa méthode en 1863. Il n'offre plus depuis cette époque aucune sécurité.
Ce type de chiffrement utilise une clé qui est un mot (ou une phrase dont les mots sont accolés). Pour procéder au chiffrement, on va placer en correspondance la phrase à chiffrer et la clé (en la répétant autant de fois que nécessaire pour qu'il y ait autant de caractères que dans la phrase à chiffrer).
On chiffre ensuite caractère par caractère le message en utilisant la méthode du chiffre de César avec comme clé la valeur de la lettre de la clé correspondante moins 1. Ainsi si la lettre de la clé est 'A', on chiffre avec n = 0, si c'est 'B', avec n =1, pour 'D' ce sera n = 3 ,…

Exemple :
Soit la phrase à chiffrer : `"Me Voici Tout Mouille, J'ai Suivi Un Nuage"`
La clé sera : `"PLANETES"`

Message	M	E		V	O	I	C	I		T	O	U	T		M	O	U	I	L	L	E			J		A	I		S	U	I	V	I		U	N		N	U	A	G	E
Clé	P	L	A	N	E	T	E	S	P	L	A	N	E	T	E	S	P	L	A	N	E	T	E	S	P	L	A	N	E	T	E	S	P	L	A	N	E	T	E	S	P	L
Décalage	15	11	0	13	4	19	4	18	15	11	0	13	4	19	4	18	15	11	0	13	4	19	4	18	15	11	0	13	4	19	4	18	15	11	0	13	4	19	4	18	15	11
Message chiffré	**B**	**P**		**I**	**S**	**B**	**G**	**A**		**E**	**O**	**H**	**X**		**Q**	**G**	**J**	**T**	**L**	**Y**	**I**			**B**		**L**	**I**		**W**	**N**	**M**	**N**	**X**		**U**	**A**		**G**	**Y**	**S**	**V**	**P**

Le message chiffré sera alors : « BP ISBGA EOHX QGJTLYI B LI WNMNX UA GYSVP »

Question 2 :
1) Ecrire une procédure de chiffrement par l'algorithme du chiffre de Vigenère dont la signature sera :
`str chiffreDeVigenere(str message, str cle)` où `message` est la phrase à chiffrer et `cle` la clé de chiffrement. La valeur de retour est la phrase chiffrée.
Tester la procédure en écrivant un programme principal qui l'appelle avec comme argument ceux de l'exemple précédent.
2) Comment déchiffrer simplement le message en se servant au maximum du code précédent ? Mettre en œuvre votre solution pour retrouver le message original.

Aide :
On pourra ré-utiliser la fonction précédente (`chiffreDeCesar()`) sur chaque caractère du message en y appliquant le bon code (à retrouver dans la clé).

[1] Certaines lettres sont plus fréquentes que d'autres suivant la langue (en français, c'est le « e » le plus fréquent). En analysant la fréquence des lettres dans le message chiffré, on peut deviner quel symbole code le caractère « e » par exemple et ainsi reconstituer facilement le code.

II – Mot de passe et hash

Dans cette partie, nous allons essayer de retrouver un mot de passe à partir de son hash.
En effet, les systèmes d'exploitation, les SGBD et les sites web stockent généralement leurs mots de passe sous forme de hash. Lorsqu'un utilisateur cherche à s'authentifier, on compare le hash du mot de passe qu'il fournit à celui stocké dans la table des mots de passe. S'ils sont égaux c'est que le mot de passe est valide.
Cette méthode permet d'éviter qu'un pirate accédant à la table des mots de passe connaisse les mots de passe de tous les utilisateurs car les fonctions de hash sont non réversibles et il n'est pas possible de retrouver le mot de passe en clair à partir de son hash.
On peut cependant tenter une attaque par force brute pour chaque mot de passe (ce qui est bien trop long pour une table entière mais peut être praticable pour un seul mot de passe s'il n'est pas trop long ou complexe).
C'est ce que nous allons faire dans cette partie.

Nous utiliserons le module `hashlib` de la bibliothèque standard de python donc la documentation se trouve sur https://docs.python.org/fr/3/library/hashlib.html

Ce module permet de hacher tout type de données. Il manipule donc des octets et non juste des caractères. Pour ce faire on utilise un nouveau type de données : les `bytestring` ou `bytes`.
On peut facilement définir une `bytestring` en préfixant une chaine normale du caractère b.
Exemple : `maBytestring = b'Ceci est une bytestring'`

On peut facilement passer d'une chaîne classique (type `str`) à une `bytestring` en utilisant les méthodes `.encode()` et `.decode()` :

```
bstring1 = b'Ceci est une bytestring'
strNormale = bstring1.decode()           # Converti la bytestring en str normale
bstring2 = "Chaîne à convertir".encode() # Converti la chaîne str en bytestring
```

L'avantage des `bytestring`, c'est qu'elles peuvent contenir des caractères non ascii qui seront alors affichés comme leur équivalent hexadécimal.
Exemple : `b'\xb0\xa2\x89'` contient les octets b0, a2 et 89 en hexadécimal. Une tentative d'exécuter la méthode `.decode()` sur cette chaîne déclenchera une erreur car l'octet b0 ne peut pas se trouver au début d'une chaine codée en utf-8.

Dans la première partie, l'algorithme de hash utilisé sera md5 (c'est un des plus simple, mais peu sécurisé). Des explications basiques en français sur l'utilisation de md5 et `hashlib` sont consultables sur https://fr.acervolima.com/hachage-md5-en-python/ .

Question 3 :

1) Compléter le programme « Q3_force_brute.py » pour demander un mot de passe à l'utilisateur et afficher le hash md5 correspondant.
   Vérifier notamment que le mot de passe `'abc'` donne bien le hash `900150983cd24fb0d6963f7d28e17f72`.
2) Combien de mots de passe différents peut-on avoir avec des mots de 3 lettres utilisant uniquement les lettres minuscules ? Même question pour des mots de 4 lettres. Si on utilise les minuscules et les majuscules combien y a-t-il de combinaisons de 3 lettres ? de 4 lettres ? Finalement est-il plus rentable d'augmenter l'alphabet (nombre de symboles possibles pour chaque lettre) ou le nombre de lettres dans le mot de passe ?

3) Ecrire une fonction `retrouve_mdp(hash_mdp, longueur_max, alphabet)` qui permet d'essayer de retrouver toutes les combinaisons et compléter le code de la question 3 pour que votre programme retrouve par force brute le mot de passe demandé à la question 1.
4) Utiliser votre programme pour retrouver le mot de passe dont le hash est `c67916bc13f15ed1689bc74bc8e4bcab`. Aide : ce mot de passe est constitué uniquement de chiffres et de lettres minuscules.
5) On utilise maintenant l'algorithme SHA256 qui est nettement plus sécurisé que md5 (SHA256 est considéré comme sûr actuellement alors que md5 est connu pour être très peu fiable et ne doit jamais être utilisé dans un contexte de sécurité : on le limite à des vérifications des données lors de transmissions peu sensibles). Quelle ligne du programme faut-il changer ? Le hash du mot de passe inconnu en SHA256 est `8b02dfed70d50b12c13e8cc43cffd08ef9ca596697d51e4baaa3dd6a13473f6e`. Modifier le programme pour qu'il retrouve ce mot de passe. Combien de temps faut-il pour retrouver le mot de passe avec SHA256 ?
6) Le temps de calcul peut être assez long et doit être fait pour chaque mot de passe. Les pirates utilisent donc des tables arc-en-ciel (*rainbow tables* en anglais) pour faciliter leur tâche. Faites une rapide recherche sur internet et expliquer en une ou deux phrases l'intérêt de ces tables.

III – Chiffrement asymétrique

1) Le chiffrement RSA

a. Présentation

L'implémentation la plus connue d'un chiffrement asymétrique est due à trois mathématiciens qui l'on découverte en cherchant justement à prouver que de tels systèmes ne pouvaient pas exister. Ces trois hommes ont donné leurs initiales au nom du procédé RSA (Ronald **R**ivest, Adi **S**hamir et Leonard **A**dleman).
RSA est de nos jours énormément utilisé dans toutes les transactions bancaires notamment et n'a pas encore été « cassé » à ce jour (du moins pas officiellement). Il existe quelques points faibles qui ont été mis à jour, mais ils concernent essentiellement la génération des clés ou les chiffrements dont les clés sont faibles car courtes (moins de 2048 bits).

b. Principe

Nous allons voir dans cette partie le principe mathématique derrière RSA. Celui-ci utilise les congruences vues en math expertes et la notion de nombres premiers[2].

L'objectif est de disposer de trois nombres n, d et e tels que pour tout entier a premier avec n (le message en clair), on ait :

$$a^{e \cdot d} = a \ [n]$$

par exemple si $n = 323$; $e = 11$ et $d = 131$ on aura : $a^{11\times131} = a$ [323] pour tout nombre entier a
ex : $74^{11\times131} = 365\ldots1824 = 74$ [323] de même $231^{11\times131}$ [323] = 231

La clé publique sera constituée des nombres e et n tandis que la clé secrète sera constituée par d et n.

Ainsi un message M chiffré avec la clé secrète donnera $C = M^d$ [n].

Lorsque le destinataire reçoit le message chiffré C, il le déchiffre en le mettant à la puissance e modulo n :

[2] Rappelons qu'un nombre premier est un nombre qui n'est divisible que par 1 et par lui-même. Par exemple 1, 2, 3, 5, 7, 11 sont premiers, mais 4 (divisible par 2), 6 (divisible par 2 et par 3), 8 (divisible par 2 et 4), 9 (divisible par 3) et 10 (divisible par 2 et 5) ne le sont pas.

$M_{déchiffré} = C^e\ [n] = (M^d)^e\ [n] = M^{d \cdot e}\ [n] = M$

Comment trouver les trois nombres n, d et e ?

On commence par choisir 2 nombres premiers p et q au hasard. Prenons 17 et 19 comme exemple (dans la pratique on utilise des nombres extrêmement grands sélectionnés au hasard).

On calcule alors le produit $n = p \times q$ de ces deux nombres. Pour nous $n = 17 \times 19 = 323$.

p et q étant premiers, il n'est pas possible de les retrouver à partir de la connaissance de n (le nombre n fait d'ailleurs partie de la clé publique).

Le théorème de Bezout stipule que si a et b sont deux entiers premiers entre eux (ils n'ont pas de multiple commun en dehors de 1) alors il existe des couples d'entiers relatifs u et v tels que : $u \cdot a + v \cdot b = 1$

Appliquons le théorème de Bezout à notre problème avec $a = e$ et $b = (p - 1)(q - 1)$, $u = d$ et $v = m$:

$d \cdot e + m \cdot (p - 1)(q - 1) = 1$

(e doit donc être premier avec $(p\text{-}1)(q\text{-}1)$ en pratique on prend généralement $e = 65537$)

Pour trouver d et m (mais seule la valeur de d nous intéresse), on doit trouver d tel que $e \cdot d - 1$ est un multiple de $(p - 1)(q - 1)$. En pratique on utilise l'algorithme d'Euclide étendu[3].

Dans notre exemple, avec e =11, cela donne $d = 131$ et $m = -5$: on a bien $131 \times 11 + (-5) \times (17 - 1)(19 - 1) = 1$

Pour prouver qu'on a bien la propriété désirée ($a^{e \cdot d} = a\ [n]$), il faut utiliser le petit théorème de Fermat généralisé qui nous assure que pour tout entier a premier avec un entier n, on a : $a^{\varphi(n)} \equiv 1\ [n]$.
$\varphi(n)$ est l'indicatrice d'Euler de n, c'est-à-dire le nombre d'entiers dans [1, n] qui sont premiers avec n (donc le nombre d'entiers qui ne sont pas des diviseurs de n).
Pour un nombre premier p, $\varphi(p) = p - 1$ car comme il est premiers tous les nombres entre 2 et lui-même sont premiers avec lui.
D'autre part, on a $\varphi(p \times q) = (p - 1) \times (q - 1)$ pour p et q deux nombres premiers distincts. En effet, les seuls nombres entiers compris entre 1 et $p \times q$ qui ne sont pas premiers avec $p \times q$ sont les multiples de p ou de q. Il y a exactement p multiples de q dans $\{1, \ldots, p \times q\}$ et q multiples de p. L'entier $p \times q$ est à la fois multiple de p et de q, donc on a $p + q - 1$ diviseurs de $p \times q$ distincts dans l'ensemble $\{1, \ldots, p \times q\}$, donc $\varphi(p \times q) = p \times q - p - q + 1 = (p - 1)(q - 1)$
Le petit théorème de Fermat permet donc d'établir pour l'entier $p \times q$: $a^{\varphi(p \times q)} \equiv 1\ [n]$ donc $a^{(p-1)(q-1)} \equiv 1\ [n]$ (①)

D'après le théorème de Bezout précédent $d \cdot e + m \cdot (p - 1)(q - 1) = 1$
D'où $d \cdot e = 1 - m \times (p - 1)(q - 1)$.
Soit a un nombre premier avec $p \times q$. On a $a^{d \cdot e} = a^{1 - m \times (p - 1)(q - 1)}$
$a^{d \cdot e} = a \times a^{-m \times (p - 1)(q - 1)} = a \times (a^{(p - 1)(q - 1)})^{-m}$ donc d'après la propriété ① on a : $a^{d \cdot e} = a \times (1)^{-m}\ [n] = a\ [n]$

On retrouve bien la propriété désirée : $a^{e \cdot d} = a\ [n]$

Maintenant que nous avons établit les principes mathématiques, voyons un exemple pratique :

[3] Comme on a fixé une valeur de e, il se peut que l'algorithme ne puisse pas trouver de solution. Dans ce cas il faut réessayer avec une autre valeur de e jusqu'à trouver une solution.

Alice veut pourvoir échanger avec Bob de manière sécurisée.

- Elle choisit deux nombres premiers $p = 17$ et $q = 19$.
- Elle calcule le produit $n = p \times q = 17 \times 19 = 323$
- Puis un nombre qui soit premier avec $(p - 1)(q - 1) = (17 - 1)(19 - 1) = 288$, par exemple $e = 11$
  $\Rightarrow$ Sa **clé publique est alors (*e*, *n*)** c'est-à-dire (11, 323) et elle la transmet à Bob.
- Par ailleurs en utilisant l'algorithme d'Euclide étendu elle calcule d tel que $d \cdot e + m \cdot (p - 1)(q - 1) = 1$
  Elle trouve $d = 131$.
  $\Rightarrow$ Sa **clé secrète est (*d*, *n*)**, soit (131,323).
- Elle peut détruire les entiers p et q qui lui ont servi à calculer les deux clés car elle n'en a plus besoin.

Bob veut envoyer le nombre 231 à Alice.

- Il utilise la clé publique d'Alice et calcule 231^{11} [323] ce qui lui donne le nombre 105.
- Il transmet le message chiffré (le nombre 105) à Alice par un canal non sécurisé.

Alice reçoit le nombre 105 provenant de Bob et chiffré avec sa clé publique.

- Elle utilise sa clé secrète et calcule donc 105^{131} [323] ce qui lui donne 231, soit le message original de Bob.

c. RSA « à la main »

Dans cette partie, nous allons réaliser un chiffrement et déchiffrement RSA avec de petits nombres et en gérant toutes les étapes directement.

Question 4 :

1) Charger le programme « Q4_RSA_à_la_main.py », regardez le code et exécutez-le pour vérifier qu'il fonctionne correctement.
2) Modifier le programme pour qu'il envoie un nombre très grand (par exemple 547896). Que constatez-vous ?

On touche ici une limite de RSA qui est que le message ne peut pas être plus grand que n. Si on veut transmettre des messages plus grands, il faudra les découper en paquets ≤ n ou (c'est généralement ce que l'on fait pour de grands volumes de données) transmettre par RSA la clé d'un chiffrement symétrique qui chiffrera le grand volume de donnée.

3) Modifier le programme pour que p et q soient choisis entre 1500 et 3000. Le problème précédent est-il réglé ? Quel autre problème potentiel est apparu ? D'où cela peut-il venir ? Modifier les paramètres du programme pour valider votre hypothèse.

d. Exponentiation rapide

Dans l'algorithme RSA, on est amené à calculer la puissance d'un nombre très grand a une puissance très grande. Si python sait parfaitement faire cela, le temps de calcul peut être extrêmement long si les entiers en jeu sont importants, ce qui sera toujours le cas avec RSA.
Heureusement, tous les calculs se font *modulo n*, ce qui va permettre de réduire les entiers utilisés.

Prenons le calcul 3^{15} [50] et décomposons-le :
3^{15} [50] = 3^7 [50] × $3^{\wedge 8}$ [50]
avec 3^7 [50] = 3^3 [50] × 3^4 [50] et 3^8 [50] = 3^4 [50] × 3^4 [50]
or 3^3 [50] = 3^1 [50] × 3^2 [50] = 3 × 9 [50] = 27 et 3^4 [50] = 3^3 × 3 [50] = 27 × 3 [50] = 31
d'où 3^7 [50] = 27 × 31 [50] = 837 [50] = 37 et 3^8 [50] = 31 × 31 [50] = 961 [50] = 11
enfin 3^{15} [50] = 37 × 11 [50] = 407 [50] = 7

On voit qu'en appliquant le modulo à chaque étape de la décomposition on ne fait que des calculs de multiplication avec des nombres strictement inférieurs à n, donc relativement rapide.

Etant donné que le même calcul revient plusieurs fois on peut même utiliser la programmation dynamique ou la mémoïsation[4] pour accélérer le processus.

Question 5 :
1) Compléter le programme « Q5_Exponentiation rapide.py » pour implémenter l'algorithme vu précédemment.
2) Introduire la fonction `exponentiation_rapide` dans le programme de la question 4 et vérifier que le temps de calcul est grandement amélioré.

e. Mise en œuvre réelle avec la bibliothèque `cryptography`

Nous allons maintenant essayer de mettre en œuvre un échange de message avec chiffrement RSA.

Pour ce faire on va utiliser les fonctions de la bibliothèque cryptography.
Cette bibliothèque très complète propose de nombreuses méthodes de chiffrement. Nous nous intéresserons uniquement ici au module RSA.

Question 6 :
1) Prenez connaissance des programmes « Q6_Generation_cles_RSA.py », « Q6_Chiffrement_RSA.py » et « Q6_Déchiffrement_RSA.py ». Essayer de comprendre le fonctionnement.
2) Utiliser le premier programme (« Q6_Generation_cles_RSA.py ») pour vous générer un couple clé publique/clé privé que vous garderez précieusement par la suite (ce programme n'est à exécuter qu'une seule fois, sinon les deux clés vont changer et vous perdrez les précédentes sauf si vous avez renommé les fichiers).
3) Communiquer votre clé publique à un ou plusieurs autres élèves via la messagerie.
4) Utiliser le programme « Q6_Chiffrement_RSA.py » et la clé d'un de vos camarades pour créer un message chiffré à son attention. Envoyez-lui ensuite le message par messagerie.
5) De votre côté, utiliser le programme « Q6_Decryptage_RSA.py » pour déchiffrer les messages chiffrés reçus à l'aide de votre clé privé. Etes-vous capable de déchiffrer un message chiffré qui ne vous serait pas destiné ?
6) Essayer d'envoyer un message assez volumineux (500 octets ou plus). Que remarque-t-on ?

En fait RSA ne permet pas de chiffrer des données plus grandes que la clé utilisée. Pour chiffrer un gros volume de donnée (comme un fichier pdf), il faudra donc obligatoirement utiliser un chiffrement symétrique.
De même pour authentifier un gros volume de données, on calculera le hash qu'on chiffrera ensuite avec la clé privée.

Question 7 :
1) En vous inspirant des 3 programmes sur le chiffrement, créer un programme qui permette de chiffrer et de signer un court message entré au clavier. La documentation de `cryptography` sera ici particulièrement utile !
2) Modifier le programme pour ajouter la possibilité de déchiffrer un message signé et vérifier sa signature. On pourra ajouter un menu qui permette de choisir entre chiffrer et déchiffrer.

[4] Voir le chapitre sur la programmation dynamique.