# TP de cryptographie pour l'embarqué


**2A informatique par apprentissage**

Chaque TP peut être fait en monôme ou en binôme. Déposer le fichier source python sur la plateforme, au plus tard une semaine après la fin du TP. Commenter directement dans le fichier source. L'ordre des TP pourra être modifié en fonction de l'avancement du cours.

Notation : contrôle continu où la moitié de la note correspond au travail réalisé pendant le TP. Les absences injustifiées donnent une note de 0 au TP sauf en cas d'accord exceptionnel avec l'encadrant de TP **et la scolarité**. Les retards sont aussi sanctionnés dans la note.

Chaque TP doit commencer par votre nom et `# -*- coding: utf-8 -*-` même si les caractères accentués ne posent pas de problème sur votre PC (-2 en cas d'absence dans les deux cas). Vous devez utiliser python 3.

# TP1 : Librairie Crypto et preuve de travail

L'objectif du TP est de découvrir la librairie pycryptodome à travers la notion de preuve de travail, utilisée par exemple dans la blockchain de bitcoin.
La documentation en ligne de la librairie Pycryptodome :
`https://www.pycryptodome.org/`

**Introduction :** un symbole étant codé sur 8 bits (1 octet) est donc représenté par deux caractères hexadécimaux. Il faut donc 64 caractères hexadécimaux pour coder un mot de 256 bits (ou 32 octet).

Il est demandé de mesurer le temps d'exécution de chaque exercice. Pour cela on vous pouvez utiliser la fonction `perf_counter()` de la librairie `time` (utiliser `import time`, ), à l'aide d'instructions du type :

```
t0 = time.perf_counter()
time.sleep(2)
t1 = time.perf_counter()
print("Il a fallut ", t1-t0, " secondes")
```

**Exercice 1 : fonctions de hachage.** On utilise le module SHA-256 à l'aide de la ligne `from Crypto.Hash import SHA256`. Ce module contient notamment la fonction `new(data)` et la classe `SHA256` avec les méthodes `update(data)` et `hexdigest()`. Par exemple si on lance le code suivant :

```
m = SHA256.new()
m.update(b"debut")
m.update(b"suite")
print(m.hexdigest())
n = SHA256.new(data = b"debutsuite")
print(n.hexdigest())
```

On obtient :
`5c77966489f2c7d3ee6b3eb059c9bf1765f6c38c76db9531c0d6ca947cd7f873`
`5c77966489f2c7d3ee6b3eb059c9bf1765f6c38c76db9531c0d6ca947cd7f873`
Remarquez le `b` devant une chaine de caractère fait la conversion directement en binaire. Le retour de la méthode `hexdigest` est sous forme hexadécimale.

**Travail à faire.** Ecrire une fonction qui compte le nombre minimum de fois qu'il faut concaténer votre prénom pour que l'empreinte du message par SHA256 se termine par `LOL`. Remarque : le code ASCII de `L` est 76 (donc 4C) et celui de `O` est 79 (donc 4F). Pour rappel, en python, une chaine de caractère peut être vue comme une liste de caractères (si `s='abc'` alors `s[0] = a, s[1] = b` et `s[2] = c`).
Comparer le résultat obtenu avec le nombre moyen qu'il fallait à priori attendre (une puissance de deux à déterminer grâce à l'introduction).

Vérifier le résultat de la manière suivante : soit $n$ le nombre trouvé. Alors
`m = SHA256.new(data = n*'prenom')`
`print m.digest()` doit afficher une empreinte (en binaire) avec `LOL` à la fin.

**Exercice 2 : chiffrement symétrique.** On utilise le module AES à l'aide de la ligne `from Crypto.Cipher import AES`. Ce module contient notamment la fonction `new(key, mode, iv)`, et la classe `AES` avec les méthodes `encrypt(data)` et `decrypt(data)`. Le paramètre `key` est la clé de chiffrement de 128 bits, `mode` est une constante pour le mode opératoire (par exemple `AES.MODE_ECB` et `AES.MODE_CBC`) et `iv` est la valeur initiale utilisée dans les modes non déterministes comme CBC. Par exemple si on lance le code suivant :

```
from Crypto.Cipher import AES

chiffrement = AES.new(b"Sixteen byte key", AES.MODE_ECB)
m_chiffre   = chiffrement.encrypt(b"message_128nbits")
print(chiffrement.decrypt(m_chiffre).decode("utf8"))
```

affiche à l'écran `message_128nbits`. Notez que la clé de chiffrement doit être exactement du bon nombre de bits ; ici 128.

Bob souhaite envoyer une lettre à Alice (modélisée ici par `16*'x'`) pour des raisons de confidentialité chiffrée en AES ECB 128 de telle sorte que le message chiffré commence par `ILOVEYOU`. Expliquer à Bob pourquoi ça va prendre trop de temps de trouver une clé adéquate en calculant en moyenne combien de clés vous avez à tester avant d'en trouver une qui marche (déterminer la puissance de deux adéquate). Proposer plutôt à Bob une clé de telle manière que le message chiffré se termine par `LOL`.

Pour cela vous devez dans un premier temps partir d'un nombre aléatoire de 128 bits. Comme c'est pour la génération d'une clé, on privilégie un générateur aléatoire dédié à la cryptographie. Vous pouvez utiliser la fonction `Crypto.Random.get_random_bytes(N)`, qui va générer aléatoirement N octets aléatoires.

Une fois ce nombre aléatoire obtenu, chiffrer le message et incrémenter la clé jusqu'à trouver un chiffré se terminant par `LOL`. Vous aurez sans doute besoin de transformer des chaines de caractères en entiers (pour pouvoir incrémenter) ou inversement (pour obtenir une clé). Cela peut se faire avec les deux fonctions du module `number` (appelé avec `from Crypto.Util import number`) : `number.bytes_to_long(byte)` et `number.long_to_bytes(long)` qui prend une chaine de caractères et la convertie en entier et inversement (que l'on peut combiner avec la fonction classique `hex()` pour avoir l'entier en hexadécimal).

Vérifier le résultat en affichant le chiffré en binaire et le message déchiffré avec la clé trouvée. Donner une estimation de la taille de l'espace des clés dans les deux cas de figure (`ILOVEYOU` vs `LOL`).

# TP2 : Courbes elliptiques sur un corps premier

Soit $p > 3$ un nombre premier et $\mathbb{Z}_p$ le corps fini à $p$ éléments. On rappelle que les points d'une courbe elliptique sont le point à l'infini $0_\infty$ et les couples $(X, Y)$ d'éléments de $\mathbb{Z}_p$ qui vérifient l'équation $Y^2 = X^3 + AX + B$, où $A, B \in \mathbb{Z}_p$.
Un point de la courbe est représenté comme une liste à trois éléments $[X, Y, Z]$ où $0_\infty = [0, 0, 0]$ et les autres points $P = (X, Y)$ sont représentés par $[X, Y, 1]$.
Remarque : c'est une représentation indépendante des coordonnées projectives.

On note $E_1$ la courbe elliptique définie sur $\mathbb{Z}_5$ par l'équation $Y^2 = X^3 + 3X + 2$ et $E_2$ celle définie sur $\mathbb{Z}_{11}$ par l'équation $Y^2 = X^3 + X + 2$.

Ecrire une fonction `verifie_point(A,B,p,P)` qui vérifie si le point $P$ est sur la courbe elliptique définie sur $\mathbb{Z}_p$, dont l'équation est décrite par $A$ et $B$. La fonction renvoit `True` ou `False` (attention $P$ peut être le point à l'infini). Vérifier que les points $0_\infty$ et $(2, 1)$ sont sur $E_1$, mais que $(2, 2)$ ne fait pas partie de cette courbe.

Ecrire une fonction `addition_points(A, B, p, P, Q)` qui prend en entrée les points $P$ et $Q$ et retourne le point $P + Q$ (définie par la loi de groupe du cours). Attention $P$ et $Q$ peuvent être identiques et/ou le point à l'infini.

**Algorithme d'addition de points :** Cet algorithme retourne le point $P + Q$ en prenant $P$ et $Q$ en entrée :

1. Si $P = 0_\infty$ retourner $Q$
2. Si $Q = 0_\infty$ retourner $P$
3. Si $P \neq Q$ et $X_P = X_Q$ retourner $0_\infty$
4. Si $P \neq Q$ et $X_P \neq X_Q$ alors calculer $\lambda = (Y_Q - Y_P)(X_Q - X_P)^{-1}$, $X = \lambda^2 - X_P - X_Q$, $Y = \lambda(X_P - X) - Y_P$ et retourner $(X, Y)$
5. Si $P = Q$ et $Y_P = 0$ retourner $0_\infty$
6. Si $P = Q$ et $Y_P \neq 0$ alors calculer $\lambda = (3X_p^2 + A)(2Y_p)^{-1}$, $X = \lambda^2 - 2X_p$, $Y = \lambda(X_P - X) - Y_P$ et retourner $(X, Y)$

Calculer sur $E_1$ les points $(2, 1) + (2, 4)$, $(2, 1) + (2, 1)$, $(2, 1) + 0_\infty, (2, 1) + (1, 1), (2, 1) + (1, 4)$ et $(1, 4) + (1, 4)$. Il faut trouver $0_\infty, (1, 4), (2, 1), (2, 4), (1, 1)$ et $(2, 4)$.

Ecrire une fonction `groupe_des_points(A, B, p)` qui retourne une liste contenant l'ensemble des points de la courbe elliptique correspondante (en utilisant la fonction `verifie_point` pour chaque valeur possible de $(X, Y, 1)$). Vérifier que la courbe $E_1$ a 5 points et que la courbe $E_2$ a 16 points.

Ecrire une fonction `ordre_point(A, B, p, P)` qui retourne l'ordre du point $P$ sur la courbe elliptique définie par $Y^2 = X^3 + AX + B$ sur $\mathbb{Z}_p$. Pour cela on pourra utiliser l'algorithme suivant :

1. $X \leftarrow P$, $c \leftarrow 1$
2. Tant que $X \neq 0_\infty$ :
   (a) $X \leftarrow X + P$
   (b) $c \leftarrow c + 1$
3. retourner $c$

Ecrire une fonction `generateurs(A, B, p)` qui retourne une liste contenant l'ensemble des éléments générateurs du groupe des points de la courbe. Vous pouvez commencer par utiliser la fonction `groupe_des_points` pour obtenir le groupe, et utiliser la fonction `ordre_point` pour discriminer les points qui sont des générateurs.

Vérifier que le groupe des points de la courbe elliptique $E_2$ n'est pas cyclique, contrairement à celui de la courbe $E_1$.

Ecrire une fonction `double_and_add(A, B, p, P, k)` qui prend en entrée un point $P$, trois entiers $A$, $B$ et $p$ et un entier $k$ et qui calcule le point $kP$ avec l'algorithme *Double-and-Add* en notant que `range(n,-1,-1)` retourne la liste $[n, n-1, \ldots, 1, 0]$, que l'on obtient la taille binaire $n$ de l'entier $k$ avec le logarithme en base deux, obtenu par `n = int(math.log(k, 2)) + 1`, et que le $i$ eme bit $k_i$ de $k$ s'obtient par l'expression `(k>>i) & 1`. Vérifier la fonction avec $k$ entre 0 et 5 avec $P = (2, 4)$ sur $E_1$. Il faut trouver $0_\infty, (2, 4), (1, 1), (1, 4), (2, 1), 0_\infty$.

# TP3 : Cryptographie elliptique sur la courbe P256

La courbe elliptique P256 définie dans la norme FIPS 186-4 (*Digital Signature Standard (DSS)*) est définie par les paramètres suivants :

$p = 115792089210356248762697446949407573530086143415290314195533631308867097853951$

$n = 115792089210356248762697446949407573529996955224135760342422259061068512044369$

$B =$ 5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b

$G_x =$ 6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296

$G_y =$ 4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5

Pour transformer les données hexadécimale en entiers on utilise la fonction `int()`. Par exemple `x = int('a80e',16)` retourne `43022`.

Cette courbe est définie sur $\mathbb{Z}_p$ par l'équation $Y^2 = X^3 - 3X + B$. Le point de base est le point $G = (G_x, G_y)$ qui est d'ordre $n$. Comme $n$ est premier, le cofacteur $h$ est 1 (i.e. $(G_x, G_Y)$ est générateur du groupe).

Calculer $p + 1 \pm 2\sqrt{p}$ et regarder si la courbe vérifie le théorème de Hasse (i.e. $n$ est dans cet intervalle), à l'aide d'une fonction `test_Hasse(n, p)`. Essayer de comprendre le résultat obtenu.

Écrire une fonction `ecdh(A, B, p, P)` qui simule le protocole Diffie Hellman en générant deux entiers aléatoires $a$ et $b$ entre 1 et $n$, puis en calculant les points $a(bG)$ et $b(aG)$ et en vérifiant l'égalité (retourner `False` si ce n'est pas le cas). Cette fonction doit être appelée avec la la courbe P256 et doit retourner une chaine de 256 bits contenant le haché par SHA256 de la première coordonnée si l'égalité est vérifiée.

Rappel : utiliser `from Crypto.Hash import SHA256`, puis `x = SHA256.new()`, `x.update(donnée)` et `x.hexdigest()`. Utiliser la fonction `long_to_bytes()` du module `number` pour convertir en chaine de caractère la donnée à hacher.

Écrire une fonction `ecdsa(A, B, p, P, n, m, a)` utilisant la courbe P256 qui retourne une signature ECDSA (deux entiers) avec la clé privée $a$, en hachant le message $m$ avec la fonction SHA256 (utiliser `bytes_to_long()`).
Écrire une fonction `ecdsa_verif(A, B, p, P, n, m, A1, sign)` qui vérifie la signature précédente *sign* sur le message $m$ avec la clé publique $A1$ (correspondant à la clé privée qui a été utilisée pour signer le message).

**Question bonus :** Implémenter l'attaque sur l'algorithme de signature ECDSA présenté dans le cours quand l'entier aléatoire $k$, généré lors de la signature, est constant (ie : le même pour deux signatures).

# TP4 : RSA CRT et attaque de Bellcore

**Génération de clés RSA :** Soient $p$ et $q$ deux grand nombres premiers, $N = p * q$, $e$ un entier premier avec $\phi(N) = (p-1)(q-1)$ et $d$ l'inverse de $e$ modulo $\phi(N)$. La clé privée est $(p, q, d)$ et la clé publique est $(N, e)$. Écrire une fonction `generer_cle_RSA(n)` qui prend en entrée un entier $n$ et génère une paire de clés RSA $p_K = (N,\ e)$ et $s_K = (p, q, d)$, où $p$ et $q$ sont des entiers premier de $n$ bits. Dans ce TP, on utilise la valeur $n = 1024$ bits. La fonction `getStrongPrime(n)` du module `number` (`from Crypto.Util import number`) génère un nombre premier fort et la fonction `inverse(x,n)` de ce module retourne l'inverse de $x$ modulo $n$. On aura vérifié que $e$ est bien premier avec $\phi(N) = (p-1)(q-1)$ lors de la génération de clé (avec une fonction `pgcd`), avant de calculer son inverse $d$ modulo $\phi(N)$.

**Présentation de RSA-CRT :** une signature RSA-CRT utilise le théorème des restes chinois pour accélérer les calculs d'exponentiation (très intéressant sur carte à puce). Ainsi une signature RSA-CRT utilise une paire de clés $p_K = (N,\ e)$ et $s_K = (p, q, d, d_p, d_q, i_p, i_q)$ où $d_p \equiv d \bmod p-1$, $d_q \equiv d \bmod q-1$, $i_p = p^{-1} \bmod q$ et $i_q = q^{-1} \bmod p$ complètent la clé privée. Modifier la fonction précédente pour retourner aussi ces valeurs.

L'idée est qu'au lieu de calculer directement la signature $s \equiv m^d \bmod N$, on calcule les entiers $s_p \equiv m^{d_p} \bmod p$ et $s_q \equiv m^{d_q} \bmod q$ et on retrouve l'unique l'entier $s$ (inférieur à $N$) tel que $s \equiv s_p \bmod p$ et $s \equiv s_q \bmod q$, soit avec le théorème des restes chinois, soit par la formule du cours $s \equiv ((q^{-1} \bmod p)qs_p + (p^{-1} \bmod q)ps_q) \bmod N$, qui correspond à la signature.

Ecrire une fonction `signature_RSA_CRT(m,sK)` qui prend en entrée un message $m$ et une clé privée RSA-CRT et qui retourne la signature RSA correspondante, calculée par le théorème chinois (ou `False` si le message $m \geq N$). Tester la fonction en comparant le résultat avec une signature RSA classique (en utilisant par exemple la fonction `pow`).

**Attaque de Bellcore :** Si une erreur se passe lors du calcul de $s_q \equiv m^{d_q} \bmod q$ (par exemple par une attaque par fautes sur carte à puce), on obtient en sortie une signature $s'$ qui vérifie $s' \equiv m^{d_p} \bmod p$ et $s' \neq m^{d_q} \bmod q$. Ainsi on a $s - s' \equiv 0 \bmod p$ et $s - s' \neq 0 \bmod q$ ce qui implique que $\text{pgcd}(s - s', N) = p$. On retrouve donc la clé privée à partir d'une signature exacte et une signature fautée sur un même message.

Ecrire une fonction `signature_RSA_CRT_faute(m,sK)` qui prend en entrée un message $m$ et une clé privée RSA CRT et qui retourne la signature RSA CRT correspondante (ou `False` si $m \geq N$) où une faute aléatoire est réalisée sur le calcul de $s_q$ (par exemple en rajoutant 1).

Ecrire une fonction `RSA_CRT_Bellcore(m, sK, pK)` qui réalise l'attaque de Bellcore sur un message $m$ en signant $m$ sans fautes puis avec une faute avec la paire de clés $K$. La fonction doit retourner $p$ et $q$ (bien sûr ceux ci doivent être retrouvés par le calcul du pgcd et non récupérés à partir de $s_K$ !).