

TP N° 2
Chiffrement / Flux

Rappels :

- Un polynôme est dit **irréductible** sur un corps donné (par exemple $\mathbb{R}$, $\mathbb{C}$, $\text{GF}(2)$) s'il possède les deux caractéristiques suivantes :
 - Il n'est pas constant : Son degré doit être au moins égal à 1.
 - Il est "incassable" : On ne peut pas l'écrire comme le produit de deux polynômes de degrés plus petits.

Autrement dit : C'est un polynôme qui ne peut pas être factorisé en produit de deux polynômes non triviaux.

(C'est l'analogue d'un nombre premier pour les polynômes.)

- Un polynôme $P(x)$ de degré n est **primitif**, s'il est irréductible et s'il vérifie que :
 - $x^{m-1} \equiv 1 \pmod{P(x)}$, $m = 2^n$, et
 - $x^k \not\equiv 1 \pmod{P(x)}$ pour tout diviseur strict k de $2^n - 1$

Exercice 1 : Étude de la sécurité des LFSR (Linéarité et Périodes)

Objectif : Comprendre la relation entre les propriétés algébriques d'un polynôme (irréductibilité, primitivité) et la robustesse du flux généré, puis expérimenter la fragilité d'un LFSR isolé.

- **Partie :** Implémentation et Propriétés Algébriques
- 1. Arithmétique dans $\mathbb{Z}/2\mathbb{Z}$: Écrire une fonction $\text{ProdPoly}(P1, P2)$ qui effectue le produit de deux polynômes dans $\mathbb{Z}/2\mathbb{Z}$.
Un polynôme peut être représenté par une liste de bits (ex: $X^3 + X + 1$: [1,0,1,1]).
- 2. Le moteur LFSR : Écrire une fonction $\text{LFSR}(\text{poly}, \text{seed}, \text{NbBits})$ qui simule le registre.
 - **poly** : liste des indices des coefficients (ex: [5,2,0] pour $X^5 + X^2 + 1$).
 - **seed** : état initial (liste de bits).
 - **NbBits** : nombre de bits à générer.

3. Analyse de la période : Écrire une fonction $\text{PeriodLFSR}(\text{poly}, \text{seed})$ qui retourne le nombre d'itérations nécessaires avant que le registre ne retrouve son état initial. Appliquer cette fonction aux 3 cas suivants de degré $n=4$ (Période max théorique $2^4-1 = 15$) :

- Cas 1 (Primitif) : $P_1(X) = X^4 + X + 1$
- Cas 2 (Irréductible non primitif) : $P_2(X) = X^4 + X^3 + X^2 + X + 1$
- Cas 3 (Réductible) : $P_3(X) = (X^2 + X + 1) \cdot (X^2 + X + 1)$ (Utilisez la question 1).
- **Conclusion** : interpréter les périodes de P_1 et P_2 , puis comparer période de P_3 et la période de $(X^2 + X + 1)$.

- **Partie** : Application au Chiffrement

4. Chiffrement d'un message :

- Soit le message $m = \text{"ENSAMCASABLANCA"}$.
- Convertissez chaque lettre en binaire sur 5 bits ($A = 00000$, $B = 00001$, ...).
- Générez un flux de bits avec le LFSR L1 (Polynôme $X^8 + X^4 + X^3 + X^2 + 1$, seed au choix).
- Réalisez le chiffrement par XOR : $c = m \oplus s$. (attention bit de poids faible est le plus à droite)

- **Partie** : Cryptanalyse et Robustesse

5. Attaque "Known Plaintext" (Message Clair Connu) : Supposons qu'un attaquant connaisse les 24 premiers bits du message clair m et les 24 premiers bits du texte chiffré c .

- L'objectif est de comparer le temps d'exécution de deux méthodes sur $n = 8$, puis 20, ...
- Déterminer les coefficients du LFSR par élimination de Gauss :
 - On suppose que la séquence interceptée est : $s_0, s_1, \dots, s_{2n-1}$
 - On cherche les coefficients $c_1, c_2, \dots, c_n$ du LFSR tels que :
 - $s_{k+n} = c_1 s_{k+n-1} \oplus c_2 s_{k+n-2} \oplus \dots \oplus c_n s_k$, pour $k = 0, 1, \dots, n-1$
 - puis, Construire le système linéaire suivant :
 - $s_n = c_1 s_{n-1} \oplus c_2 s_{n-2} \oplus \dots \oplus c_n s_0$,
 - $s_{n+1} = c_1 s_n \oplus c_2 s_{n-1} \oplus \dots \oplus c_n s_1$,
 - ...
 - $s_{2n-1} = c_1 s_{2n-2} \oplus \dots \oplus c_n s_{n-1}$,
 - Écrire ce système sous forme matricielle (A est une matrice de Hankel [ou Toeplitz selon l'orientation droite gauche]) :

$$A \cdot C = B \text{ dans } GF(2)$$
 - Résoudre ce système par élimination de Gauss modulo 2.
- De même, en utilisant l'algorithme de Berlekamp-Massey (cf annexe A).

6. Vers la robustesse : Composition de LFSR : Pour masquer la linéarité, on utilise souvent plusieurs LFSR combinés par une fonction non-linéaire (Ex: le générateur de Geffe).

- Implémenter trois LFSR avec des polynômes primitifs de votre choix.
- Générer les trois suites S_1, S_2 et S_3 respectifs
- Créer un flux combiné $S_F = (S_1 \wedge S_2) \oplus ((\neg S_1) \wedge S_3)$
- Déterminer expérimentalement les périodes de S_1, S_2, S_3 et S_F
- Comparer la période de S_F aux périodes individuelles.

Exercice 2 : Comparaison de chiffrements par flux

Objectif : Comparer la performance de différents générateurs de flux : LFSR, Trivium, Salsa20 et ChaCha20.

Étapes :

1. Génération du texte clair :
 - Créer un texte aléatoire de 1 Méga Octet à chiffrer.
2. Chiffrement par flux :

Chiffrer le texte avec les quatre méthodes suivantes :

 - LFSR non linéaire :
 - Vous pouvez composer plusieurs LFSR linéaires par une fonction non linéaire (exemple : $(S1 \text{ AND } S2) \text{ XOR } S3$) pour augmenter la robustesse.
 - Indiquez la taille de vos LFSR et la graine utilisée.
 - Trivium :
 - Implémentez le générateur Trivium en Python ou utilisez une implémentation éducative existante.
 - Utilisez une clé et un IV de 80 bits chacun.
 - Salsa20 (via PyCryptodome) :
 - `pip install pycryptodome`
 - Générer une clé de 32 octets et un nonce de 8 octets.
 - ChaCha20 (via PyCryptodome) :
 - `pip install pycryptodome`
 - Générer une clé de 32 octets et un nonce de 8 octets.
3. Déchiffrement :
 - Déchiffrer le texte avec la même clé/IV ou graine pour vérifier que le texte récupéré correspond bien au texte original.
4. Mesure des performances :
 - Mesurer le temps de chiffrement et de déchiffrement pour chaque algorithme.
 - Comparer les performances.

Annexe

Annexe A : Pseudo code de l'algorithme de Berlekamp–Massey

```
Entrée : sequence = [s0, s1, ..., s2n-1]    # bits interceptés
Sortie : C(x) = polynome de connexion
        L = degre du LFSR

Initialisation :
    C = [1]                # polynôme courant
    B = [1]                # polynôme précédent
    L = 0                  # degré courant
    m = -1                 # indice de dernier ajustement
    n = 0

Pour n de 0 à len(sequence)-1 :
    # Calculer la discrepancy d
    d = sequence[n]

    Pour i de 1 à L :
        d = d XOR (C[i] AND sequence[n-i])

    Si d == 1 :
        T = C.copy()

        # Calculer le décalage
        decalage = n - m

        # Étendre C si nécessaire
        if len(C) < len(B) + decalage:
            C = C + [0] * (len(B) + decalage - len(C))

        # C = C + x^(decalage) * B
        pour i de 0 à len(B)-1 :
            C[i + decalage] = C[i + decalage] XOR B[i]

        Si (2*L <= n) :
            L = n + 1 - L
            B = T
            m = n    # ← CORRECT: on stocke la position courante
    # Sinon : pas de changement

Retourner C(x), L
```