

Travaux Pratiques

Attaque sur GEA-1

Yann ROTELLA

28 avril 2026

GEA-1 est un algorithme de chiffrement à flots conçu par le groupe ETSI Security Group of Experts en 1998, à une époque où l'utilisation de la cryptographie était régulée, notamment en France où seul des algorithmes « faibles » cryptographiquement pouvaient être utilisés sans l'accord du premier ministre : tout algorithme ayant une sécurité supérieure à 40 bits devait être soumis à l'accord du premier ministre pour être utilisé civilement.

Le but de cette feuille consistera à programmer (en utilisant **C** et **Sage**) l'attaque trouvée en 2021 et publiée à EUROCRYPT.

LFSRs et Initialisation de GEA-1

Alors que les spécifications de GEA-1 étaient restées secrètes (ne respectant pas le principe de Kerckhoff), celles-ci ont été récemment retrouvées publiquement : les voici.

L'état interne dans GEA-1 est de taille 96 bits, séparé en trois registres A , B et C de tailles respectives 31, 32 et 33 bits et utilise une clef de 64 bits.

1. La clef secrète, le vecteur d'initialisation et un bit de direction sont « mélangés » dans une procédure dite d'initialisation au sein d'un registre S qui produit une suite (secrète) de 64 bits notée $s = s_{64}s_{63} \cdots s_0$. Nous ne détaillerons pas dans ce TP cette procédure.
2. Ces 64 bits sont ensuite injectés dans les registres (LFSRs) A , B et C . Chacun est initialisé à 0 et A reçoit $s_0, s_1, \dots, s_{63}$, B reçoit $s_{16}, s_{17}, \dots, s_{63}, s_0, \dots, s_{15}$ et C reçoit $s_{32}, s_{33}, \dots, s_{63}, s_0, \dots, s_{31}$. Après chaque bit injecté, la fonction de mise à jour (le polynôme de rétroaction) est appliqué.

À toi de jouer : regarde la structure LFSR dans le code fourni et la fonction `updateLFSR` et modifie celle-ci pour qu'elle soit correcte. Pour t'aider je te donne la figure qui représente GEA-1. Les bits de S sont injectés à gauche dans la figure.

À toi de jouer : modifie les deux lignes dans `init` de telle sorte que cela réalise l'initialisation comme indiqué au dessus.

Appliqué à $S = 1$, le code doit renvoyer les valeurs suivantes en hexadécimal pour les différents registres :

- Registre A : 0x58EC8DDD
- Registre B : 0xC3C529D5
- Regsitre C : 0xBEE39C47

À toi de jouer : Que s'est-il passé si $S = 1$ pour le registre A ? Pour t'aider, écris en binaire le masque pour le registre A et sa valeur en partant de la droite (du bit de poids faible) et regarde la figure et le registre A .

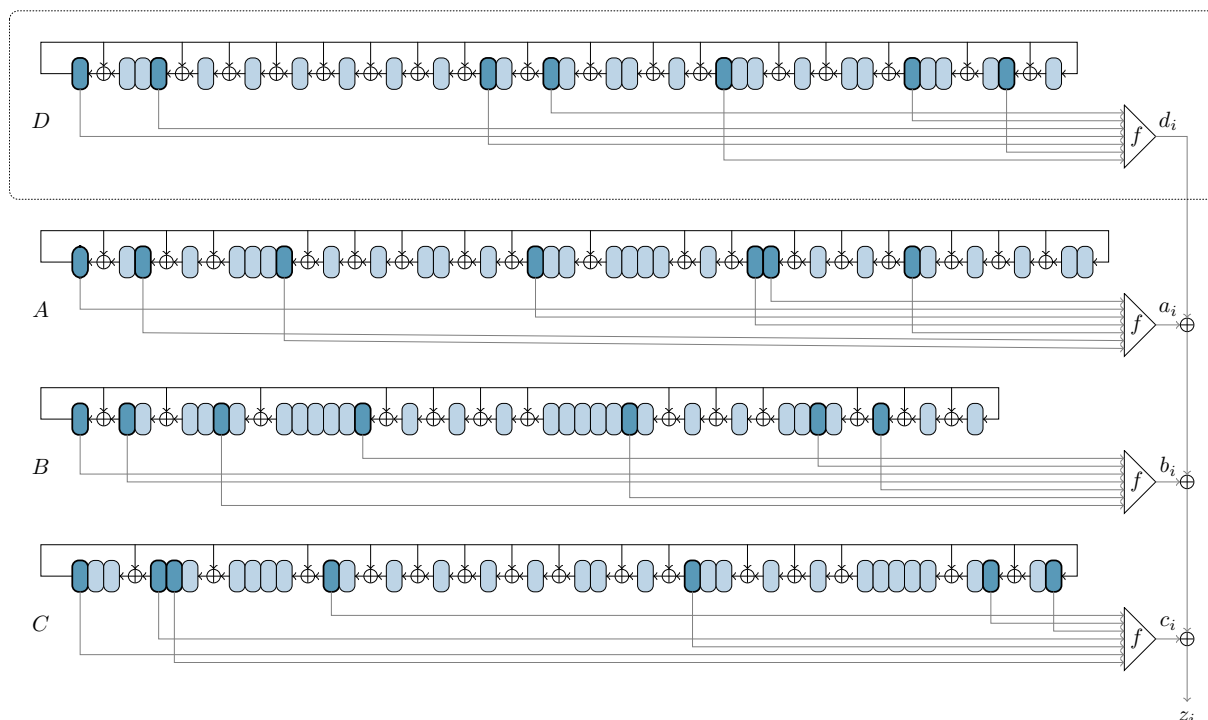


FIGURE 1 – Génération de la suite chiffrante pour GEA-1 et GEA-2. Le registre D n'est présent que pour GEA-2.

Génération de la suite chiffrante de GEA-1

Une fois l'initialisation terminée, la suite chiffrante dans GEA-1 est générée en appliquant une fonction booléenne $f : \mathbb{F}_2^7 \rightarrow \mathbb{F}_2$ à 7 bits bien choisis de chaque registre et en faisant un XOR de ces trois bits. À chaque fois les registres sont mis à jour.

À toi de jouer : Regarde la fonction `inputF`. Le calcul de la fonction f est donné au début du code. Enfin, remplis la fonction `stream` afin que celle-ci engendre 64 bits de suite chiffrante.

Optionnel : écris une fonction `encrypt` et `decrypt` qui est capable de chiffrer et de déchiffrer une suite de 8 caractères.

Question : Que peut-on dire de l'application $L_A : \mathbb{F}_2^{64} \rightarrow \mathbb{F}_2^{31}$ qui au secret S associe la valeur initiale du registre A ? Même question pour B et C .

Programmation : Calculez une représentation de ces trois applications avec le code en C. La fonction `printBinaire` sera utile.

Tu peux maintenant passer à l'analyse et à l'utilisation de SageMath !

Analyse de la faiblesse de GEA-1

Analyse : Vérifie ta représentation en ouvrant le fichier `matrices.py`.

Comme on a des matrices, on va les analyser. Pour cela **SageMath** est parfait et va permettre d'obtenir une analyse des espaces vectoriels définis par nos trois matrices.

Programmation : Lance **SageMath** et déclare les trois matrices. Une matrice en Sage peut se déclarer avec le constructeur `Matrix` qui prend deux éléments en entrée : le corps (ici $\mathbb{F}_2$ déclaré avec `GF(2)`) et un tableau de tableaux, donné dans le fichier `matrices`.

Programmation : Calcule les noyaux de ces trois matrices avec la méthode `.kernel()` ainsi que leurs dimensions avec la méthode `.dimension()`. Est-ce que ce que tu obtiens est normal ?

Programmation : Calcule les intersections de tous ces noyaux avec la méthode `.intersection()` puis calcule la dimension de ces trois intersections. Que remarques-tu ? Est-ce que ce phénomène doit avoir lieu si les matrices avaient été tirées au hasard (avec le même nombre de lignes et de colonnes) ?

Il va falloir exploiter cela. Nous allons réutiliser les notations qui peuvent être trouvées dans l'article original : T_{AC} dénote l'intersection du noyau de M_A avec le noyau de M_B . Dans ces conditions, on a

$$\mathbb{F}_2^{64} = T_{AC} \oplus \text{Ker}(M_B) \oplus V$$

Programmation : Vérifie que T_{AC} et $\text{Ker}(M_B)$ sont en somme directe.

Programmation : Calcule V de dimension 8 aligné sur la base canonique.

Programmation : Ecris une fonction qui transforme un vecteur de 64 bits en entier.

Programmation : Récupère en hexadécimal 24 vecteurs de base pour T_{AC} , 32 vecteurs pour $\text{Ker}(M_B)$ et 8 vecteurs de V . On pourra utiliser `hex()` et `.basis()`.

Programmation : Remets les 56 vecteurs pour T_{AC} et $\text{Ker}(M_B)$ dans le code en C sous des tableaux statiques en `uint64_t`. **Demande moi si tes vecteurs sont bons, ce serait dommage de devoir revenir dessus.**

Exploitation et attaque

La décomposition du secret S en somme directe de trois espaces vectoriels V , $T_{AC} = \text{Ker}(M_A) \cap \text{Ker}(C)$ et $\text{Ker}(B)$ permet de faire une attaque de type « Meet-in-the-middle ». En effet, pour tout $S \in \mathbb{F}_2^{64}$, il existe un unique triplet $(u, v, w) \in T_{AC} \times V \times \text{Ker}(B)$ tel que $s = u + v + w$.

Par ailleurs, la suite chiffrante dans **GEA-1** est calculée comme le XOR de trois fonctions appliquées à chaque registre indépendamment. Soit $s = u + v + w$ un secret donné et u , v et w les coordonnées de ce vecteur dans la somme directe et f_A , f_B et f_C les fonctions qui prennent en entrée l'état initial dans le registre A (respectivement B et C) et produisent la sortie a (respectivement b et c) de taille arbitraire.

Avec ces notations, la suite chiffrante z (de taille arbitraire) vaut

$$z = a \oplus b \oplus c = f_A(a_0) \oplus f_B(b_0) \oplus f_C(c_0)$$

où a_0 , b_0 et c_0 sont les états initiaux des trois registres après initialisation.

Par construction et propriétés du noyau, on a

$$a_0 = M_A(s) = M_A(u + v + w) = M_A(v + w)$$

$$b_0 = M_B(s) = M_B(u + v + w) = M_B(u + v)$$

$$c_0 = M_C(s) = M_C(u + v + w) = M_C(v + w)$$

Par conséquent, la suite chiffrante s'écrit comme $z = f(u + v) + g(v + w)$ pour deux fonctions f et g connues.

Algorithm 1 Attaque de GEA-1

Input: z un bout de suite chiffrante

Output: s le secret de 64 bits

```

for  $v \in V$  do
   $L \leftarrow \{M_B(u + v), u \in T_{AC}\}$ 
  Trier  $L$ 
  for  $w \in \text{Ker}(M_B)$  do
    Calculer  $r = z \oplus g(v + w)$ 
    Chercher  $r$  dans  $L$ 
    Renvoyer  $s = u + v + w$ 
  end for
end for

```

Questions : Quelle est la complexité de cet algorithme ? Pourquoi faut-il trier la liste L ?

Interlude : les codes de Gray

Pour énumérer toutes les valeurs dans un espace vectoriel E dont on connaît une base, on peut réaliser un produit matrice vecteur pour tous les vecteurs de la base canonique, en énumérant tous les entiers jusqu'à $2^{\dim(E)}$. Cependant, cela nécessite une multiplication matrice vecteur à chaque étape. En fait on peut faire beaucoup mieux avec l'utilisation des codes de Gray. Les codes de Gray sont une manière d'énumérer tous les entiers de k bits avec la propriété que la différence entre deux entiers consécutifs de la suite soit de 1 bit. Une fonction `grayCodes` dans le code en C permet d'engendrer les codes de Gray.

Dans notre cas, cela nous permet d'énumérer tous les éléments de nos espaces vectoriels sans faire de multiplication matrice-vecteur : on calcule à l'itération i de la boucle le i -ième code de gray, dont chaque bit indique le vecteur correspondant dans l'espace vectoriel ($5 = 101$ correspondra au vecteur $e[0] \oplus e[2]$ si $e[i]$ sont une base de notre espace vectoriel E).

Pour l'instant, on ne va pas modifier et parcourir tout l'espace V , on va se concentrer sur T_{AC} .

Programmation : En utilisant les codes de Gray, remplis dans la deuxième boucle du code de `attack` quelques lignes afin que le secret `uint64_t S` parcoure l'entiereté de T_{AC} . Vérifie en imprimant les premières valeurs parcourues. La fonction `__builtin_ctz` peut être utile car elle renvoie le nombre de zéros consécutifs en commençant par le bit de poids faible.

La liste L doit être triée et contient les sorties de **GEA-1**, mais nous nous intéressons à récupérer l'état S du secret. Ainsi, la liste doit contenir des paires de valeurs.

Programmation : Regarde la structure `TAB` puis rajoute quelques lignes afin qu'à la fin de cette boucle la liste L contienne toutes les valeurs nécessaires pour la suite de l'attaque.
Attention : pense à bien déclarer ton tableau au bon format et à allouer la mémoire correctement avec `malloc`.

Tri rapide en C : la fonction `qsort` prend en entrée 4 paramètres : le tableau à trier, sa taille, la taille des éléments des cellules et une fonction de comparaison.

Programmation : Rajoute une ligne après la boucle pour trier le tableau en fonction des valeurs de sortie. La fonction de comparaison t'est donnée dans le code.

Pour la deuxième partie, il va falloir chercher s'il existe un élément, en itérant maintenant sur $\text{Ker}(M_B)$. Pour t'aider, la recherche dichotomique est déjà codée et s'utilise comme suit.

```
— TAB *x = binary_search_TAB()
— if (x) {...}
```

Programmation : Réalise la deuxième boucle, en réutilisant l'astuce des codes de Gray, en itérant sur $\text{Ker}(M_B)$.

Objectif 1

Programmation : Peux-tu retrouver le secret S correspondant à la suite chiffrante `0x2D0E339CFD588D1B` ? **Important :** ce challenge a été généré afin qu'il soit trouvé dans les premières itérations des deux boucles : il n'est pas nécessaire d'aller jusqu'à 2^{24} et 2^{32} respectivement.

Objectif 2

Programmation : Peux-tu retrouver le secret S correspondant à la suite chiffrante `0x9AC76518F2322DA4` ?