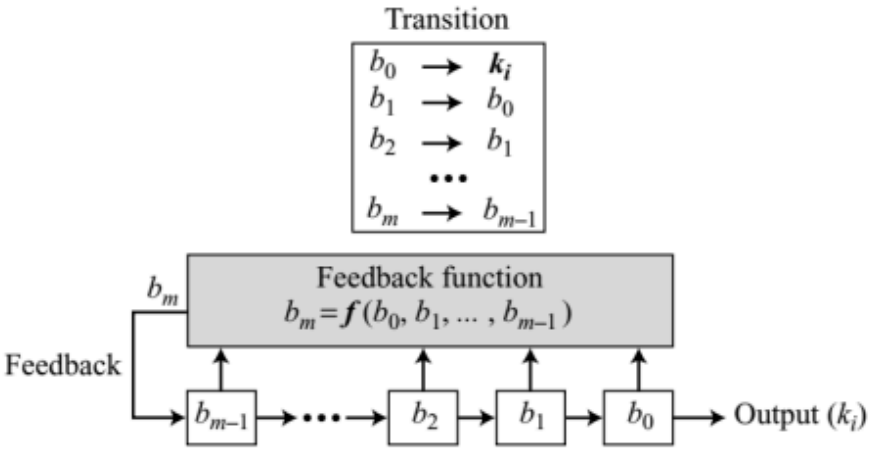
Bilel Rahmouni - Cheikh Ahmadou Bamba Diouf

TP2- PNRG et Chiffrement a Flot

Exercice 1 : LFSR

Un LFSR est un générateur de nombre aléatoire de la forme suivante :



La fonction f est linéaire et elle est de la forme suivante :

$$b_m = c_{m-1}\, b_{m-1} \oplus \quad \cdots \quad \oplus c_2\, b_2 \oplus c_1\, b_1 \oplus \; c_0\, b_0 \qquad (c_0 \neq 0)$$

1. Créer un LFSR de 4 cellules avec b4 = b1 ⊕ b0 et afficher les états après 20 transitions

LFSR : Un registre à décalage linéaire à rétroaction

4 bits : b0, b1 , b2 et b3

État initial : (0001)2

Transition	Etat (b3 – b2 – b1 – b0)	Sortie
Initial	0001	
1	1000	1
2	0100	0
3	0010	0

4	1001	0
5	1100	1
6	0110	0
7	1011	0
8	1101	1
9	1110	1
10	1111	0
11	0111	1
12	1011	1
13	1101	1
14	1110	1
15	1111	0
16	0111	1
17	1011	1
18	1101	1
19	1110	1
20	1111	0

2. Le LFSR est-il périodique et quelle est sa période ?

Oui, le LFSR est périodique. Car, on peut observer que l'état "1011", "1101", "1110" et "1111" se répète. La période de ce LFSR est donc 4, car à partir de l'état 7, le même cycle se reproduit.

3. Quelle est la période maximale d'un LFSR ?

La période maximale d'un LFSR est 2^n -1, où n est le nombre de cellules du LFSR. Pour un LFSR de 4 cellules, la période maximale est donc 2^4-1=15. Cela se produit uniquement si le LFSR est maximal, c'est-à-dire que la fonction de rétroaction est bien choisie.

4. Quelles sont les attaques sur les LFSR ? Comment les contrer ?

Les LFSR sont vulnérables aux attaques telles que :

- Attaque par corrélation : Cette attaque exploite le fait que certaines sorties d'un LFSR sont corrélées avec l'état interne. Pour contrer cette attaque, il est courant d'utiliser plusieurs LFSR en parallèle (générateurs combinés) ou de combiner les LFSR avec une fonction non-linéaire.

- Attaque par berlekamp-massey : Cette attaque permet de retrouver la séquence génératrice en analysant une suite suffisante de bits de sortie. La contre-mesure consiste à utiliser des constructions non linéaires pour masquer la structure linéaire de l'algorithme.

- Attaque par observation : Si un attaquant peut observer plusieurs bits de sortie, il peut remonter jusqu'à l'état initial. Pour se protéger, on peut introduire une fonction de combinaison non linéaire ou utiliser des techniques comme la permutation des bits.

En résumé, les techniques pour contrer ces attaques incluent l'utilisation de fonctions de rétroaction non linéaires, l'augmentation du nombre d'étages de l'algorithme, ou l'utilisation de plusieurs LFSR combinés ensemble.

Exercice 2 : RC4

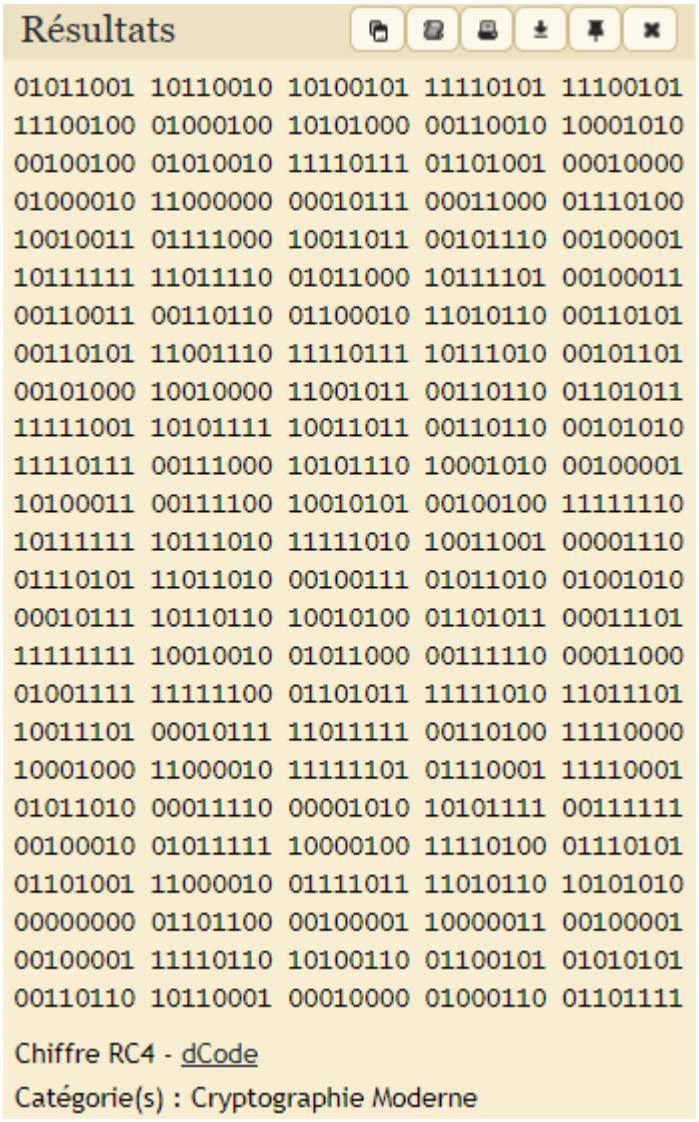
1. Générer une suite aléatoire de 1000 bits en utilisant l'algorithme RC4

RC4 (Rivest Cipher 4): un algorithme de chiffrement à flot

On utilise dcode :

Et on chiffre un message de 125 caractères car chaque caractère correspond à 8 octet et 8 x 125 est égal à 1000 bits.

On obtient donc ce résultat :



Résultats

01011001 10110010 10100101 11110101 11100101
11100100 01000100 10101000 00110010 10001010
00100100 01010010 11110111 01101001 00010000
01000010 11000000 00010111 00011000 01110100
10010011 01111000 10011011 00101110 00100001
10111111 11011110 01011000 10111101 00100011
00110011 00110110 01100010 11010110 00110101
00110101 11001110 11110111 10111010 00101101
00101000 10010000 11001011 00110110 01101011
11111001 10101111 10011011 00110110 00101010
11110111 00111000 10101110 10001010 00100001
10100011 00111100 10010101 00100100 11111110
10111111 10111010 11111010 10011001 00001110
01110101 11011010 00100111 01011010 01001010
00010111 10110110 10010100 01101011 00011101
11111111 10010010 01011000 00111110 00011000
01001111 11111100 01101011 11111010 11011101
10011101 00010111 11011111 00110100 11110000
10001000 11000010 11111101 01110001 11110001
01011010 00011110 00001010 10101111 00111111
00100010 01011111 10000100 11110100 01110101
01101001 11000010 01111011 11010110 10101010
00000000 01101100 00100001 10000011 00100001
00100001 11110110 10100110 01100101 01010101
00110110 10110001 00010000 01000110 01101111

Chiffre RC4 - dCode

Catégorie(s) : Cryptographie Moderne

Le message :

a9JkQmZ7xY2BcVwR8nT5pL0vK4sXeFgH1U6MbPqCjDrSyNzW3tIuOhEdAlG0a9JkQmZ7xY2BcVwR8nT5pL0vK4sXeFgH1U6MbPqCjDrSyNzW3tIuOhEdAlG025417

1000 bits :

01011001101100101010010111110101111001011110010001000100101010000011001010001010001001000101001011110111011010010001000010000010111000000000101110001100001110100100100111011110001001101100101110000100001101111111101111001011000010111101001000110011001101100110110110101100011010100110101111001110111011101110100101101001010001001000001100101100110110011010111110011010111110011011000101010111110111000101010111010000101000101010001111100100101010010011111111101011111110111010111110101001100100000111001110101101101000100111010110110100100101000101111011011010010011010001011011000111011111111111001001001011000000111110001100001001111111100011010111110101101110110011101000101111011111001101001111000100001

10001100001011111101011100011111000101101000011110000010101011110011111001000100101111100001000111101000111010101101001110000101111011101011010101010100000000001101100001000011000001100000111110110101001100101010101001101101011000100010000010000110011011111

2. Chercher le document NIST qui décrit les 15 tests statistiques

Le document NIST qui décrit les 15 tests statistiques pour mesurer la qualité aléatoire d'une séquence est le NIST SP 800-22 intitulé "*A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*". Ce document présente une batterie de tests utilisés pour vérifier la qualité des séquences générées.

Quelques exemples de tests du NIST SP 800-22 :

- Test de fréquence (monobit test) : Ce test vérifie si le nombre de '1' et de '0' dans la séquence est approximativement égal, ce qui est attendu pour une séquence aléatoire.

- Test de longueurs des suites : Ce test vérifie la longueur des séquences successives de 0 ou de 1. Il s'assure qu'il n'y a pas trop de séquences trop longues ou trop courtes de bits identiques dans la séquence.

- Test de compressibilité : Ce test mesure dans quelle mesure la séquence peut être compressée. Si une séquence peut être fortement compressée, elle contient une structure répétitive et donc n'est pas suffisamment aléatoire.

3. Chercher un outil qui implémente ces tests. Exécutez-le pour tester la séquence de la première question.

Nous utilisons le logiciel Dieharder afin d'effectuer ces tests. Son installation se fait directement sur ubuntu avec une commande classique : sudo apt install dieharder

On crée ensuite un fichier .bin pour implémenter la suite de 1000 bits de la question précédente, puis on exécute la commande qui initie les tests de séquence.

dieharder -a -f binary-suite.bin

```
[~/Documents]
rahmonex ⊟ dieharder -a -f binary-suite.bin
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |           filename             |rands/second|
        mt19937|                binary-suite.bin|  1.03e+08  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.82106969|  PASSED
      diehard_operm5|   0|   1000000|     100|0.83803446|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.09666615|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.59312285|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.48828533|  PASSED
        diehard_opso|   0|   2097152|     100|0.70639466|  PASSED
        diehard_oqso|   0|   2097152|     100|0.85541500|  PASSED
         diehard_dna|   0|   2097152|     100|0.15143602|  PASSED
diehard_count_1s_str|   0|    256000|     100|0.60707606|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.38605585|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.72516127|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.20708943|  PASSED
```

Puisqu'il n'y a que quelques résultats faibles, la séquence peut être considérée comme suffisamment aléatoire. Cependant, pour des applications très sensibles à la sécurité, il pourrait être judicieux de générer une nouvelle séquence ou d'augmenter la taille pour améliorer l'aléatoire.
Quelques tests comme `rgb_bitdist` et `rgb_lagged_sum` ont affiché des résultats "WEAK", ce qui peut nécessiter une investigation supplémentaire, mais ces résultats sont généralement acceptables dans des contextes non critiques.

4. Etudier les faiblesses de l'utilisation du RC4 dans WEP, puis décrire comment un attaquant peut
les exploiter dans la pratique. Quelle est la durée nécessaire en moyenne pour réussir l'exploit. Quels sont les enseignements que vous pouvez tirer de cet exemple.

Le protocole WEP utilise l'algorithme RC4, mais il présente plusieurs vulnérabilités importantes :

1. **Vecteur d'initialisation (IV) court** : WEP utilise un IV de 24 bits qui est trop court et finit par se répéter, permettant aux attaquants d'observer des motifs dans les paquets.
2. **IV transmis en clair** : Comme l'IV est envoyé en clair dans chaque paquet, un attaquant peut facilement l'analyser et découvrir la clé secrète RC4.
3. **Attaque FMS** : Les IV faibles permettent une attaque statistique appelée attaque FMS, qui aide à deviner la clé WEP.

Pour casser WEP, un attaquant commence par capturer des paquets sur le réseau. En utilisant des outils comme Aircrack-ng, il peut exploiter la réutilisation des IV et lancer une attaque FMS. En

moyenne, un attaquant peut réussir à casser la clé WEP après avoir capturé environ 4 à 6 millions de paquets, ce qui peut prendre de quelques minutes à quelques heures selon le trafic réseau.

Cet exemple montre que l'utilisation de clés statiques et de vecteurs d'initialisation courts est dangereuse. Cela souligne aussi l'importance de mettre à jour les protocoles de sécurité pour éviter ce type de failles.

5. Expliquer comment l'algorithme TKIP a corrigé les failles de la question précédente.

L'algorithme TKIP (Temporal Key Integrity Protocol) a été introduit pour corriger les failles de WEP tout en restant compatible avec le matériel existant. Voici comment il a amélioré la sécurité :

1. **Rotation des clés** : Contrairement à WEP qui utilise une clé statique, TKIP génère une nouvelle clé pour chaque paquet en combinant une clé principale avec un numéro de séquence unique. Cela évite la réutilisation des IV.
2. **Vecteur d'initialisation plus long** : TKIP utilise un IV de 48 bits, beaucoup plus long que celui de WEP (24 bits), réduisant considérablement les chances de répétition.
3. **Contrôle d'intégrité (MIC)** : TKIP introduit un code d'intégrité des messages (MIC, Message Integrity Check) pour vérifier que les paquets n'ont pas été modifiés, ce qui prévient les attaques par falsification.

Ces améliorations ont corrigé les faiblesses majeures de WEP, notamment en empêchant les attaques basées sur la réutilisation des IV et la prédiction des clés.

6. En 2013, d'autres failles ont été publiées ce qui a mis fin à l'utilisation de l'algorithme RC4 dans TLS et HTTPS. Cherchez l'article qui explique cette faille et faites un petit résumé. En 2015, l'IETF a radié RC4 de TLS. Quel est l'algorithme qui l'a remplacé ?

Failles publiées en 2013 sur RC4 dans TLS/HTTPS

En 2013, des chercheurs ont découvert des failles critiques dans l'utilisation de RC4 dans les protocoles TLS et HTTPS. Une des attaques notables est appelée "RC4 NOMORE". Cette attaque exploite des biais statistiques dans le flux généré par RC4. Après avoir observé un grand volume de données chiffrées, ces biais permettent de récupérer des fragments de texte en clair, notamment dans les sessions sécurisées.

En résumé, RC4 présente des faiblesses dans la génération de ses flux de clés, et lorsqu'il est utilisé sur de longues sessions, ces biais deviennent suffisamment prévisibles pour permettre des attaques de décryptage partiel. Cela a mené à la décision d'abandonner RC4 dans les communications sécurisées.

Remplacement de RC4

En 2015, l'IETF a officiellement déprécié RC4 dans TLS à travers le RFC 7465. L'algorithme qui a remplacé RC4 est AES (Advanced Encryption Standard), utilisé principalement dans les modes CBC (Cipher Block Chaining) et GCM (Galois/Counter Mode) pour renforcer la sécurité dans TLS. AES offre une bien meilleure résistance aux attaques cryptographiques.

Exercice 3 :

Linux obtient le caractère aléatoire des ressources physiques suivantes :

```
void add_keyboard_randomness(unsigned char scancode);
void add_mouse_randomness(__u32 mouse_data);
void add_interrupt_randomness(int irq);
void add_blkdev_randomness(int major);
```

1. Expliquer la signification de ces sources

`add_keyboard_randomness(unsigned char scancode)` : Cette fonction collecte l'aléa généré par les événements clavier, comme les frappes de touches. Chaque appui sur une touche produit un code de scan (scancode), qui est utilisé pour ajouter de l'entropie (de l'aléa) au pool aléatoire.

`add_mouse_randomness(__u32 mouse_data)` : Cette fonction ajoute des données aléatoires provenant des mouvements de la souris. Les mouvements de la souris sont considérés comme une bonne source d'aléa en raison de leur caractère imprévisible.

`add_interrupt_randomness(int irq)` : Cette fonction utilise les interruptions matérielles (IRQ) comme source d'entropie. Chaque interruption déclenchée par un périphérique (comme un disque dur ou une carte réseau) ajoute des données au pool aléatoire.

`add_blkdev_randomness(int major)` : Cette fonction collecte des informations provenant des dispositifs de blocs (comme les disques durs). Les opérations d'E/S (entrées/sorties) sur ces dispositifs fournissent des données imprévisibles qui sont ajoutées à l'entropie du système.

2. Expliquer la notion de l’entropie, puis afficher sa valeur courante dans votre système. Afficher sa valeur avec la commande watch.

L'entropie, dans le contexte de la cryptographie, fait référence à la quantité de désordre ou d'incertitude. Une haute entropie indique un niveau élevé de hasard, ce qui est essentiel pour créer des clés cryptographiques sécurisées et imprévisibles.

Sur Linux, l'entropie est accumulée à partir de diverses sources matérielles, telles que les frappes au clavier, les mouvements de la souris, et les interruptions matérielles (comme expliqué dans la première partie). Le noyau stocke cette entropie dans un "pool aléatoire". Plus ce pool contient de l'entropie, plus le système peut fournir des nombres aléatoires sécurisés via des interfaces comme `/dev/random` et `/dev/urandom`.

Pour vérifier la quantité actuelle d'entropie disponible dans ton système, tu peux consulter le fichier spécial `/proc/sys/kernel/random/entropy_avail`

```
rahmonex  cat /proc/sys/kernel/random/entropy_avail
256
```

On obtient donc le résultat 256.

3. Linux stocke les données collectées de ces sources dans un « random pool », puis utilise deux fichiers pour stocker les nombres aléatoires : /dev/random et /dev/urandom. Le fichier /dev/random est bloquant.
a. Expliquer cette notion. Exécuter la commande cat /dev/random | hexdump afin d'illustrer cette notion.
`/dev/random` est bloquant, ce qui signifie qu'il fournit des nombres aléatoires uniquement si suffisamment d'entropie est disponible dans le système. Si l'entropie est insuffisante, la lecture se bloque jusqu'à ce que le pool d'entropie se remplisse à nouveau.

La commande "cat /dev/random | hexdump"montre le comportement bloquant.
Si le système manque d'entropie, la commande peut se bloquer et attendre que de nouvelles sources d'entropie soient générées

```
2a0cb10 ff16 4cc4 87dd d20b 275f 19bb fc33 d4d1
2a0cb20 4921 8a24 d9f9 d1f5 dfae 381b 42d0 9d03
2a0cb30 ddc8 c165 f1b0 2866 d944 64b8 08d7 4c15
2a0cb40 a34a 6fc1 285f 7ff8 acd9 543b 2e34 3a91
2a0cb50 f04c a8ca 85f2 59e0 5cc9 41c4 6f7b e6ca
2a0cb60 1843 8126 fa39 8731 9ca7 1734 c6e7 7664
2a0cb70 054e b9bd f013 54a5 cb81 c332 459c f98d
2a0cb80 19e6 caa8 e1fe b5ca fff4 d013 b264 1091
2a0cb90 135c 0273 3095 cf1c 2d8c 98b1 1318 659b
2a0cba0 be18 5b36 4312 2c3d e31c 850e 765a 82eb
2a0cbb0 3a18 37ee fb40 af2b f616 2b5f 0590 3578
2a0cbc0 8304 1a8e dbdf 73e7 cabe 1539 5b43 2209
2a0cbd0 31c0 3681 f880 06a6 d2f4 edd1 310d a605
2a0cbe0 78ad 5735 be04 f124 b003 b05f bf23 8e3a
2a0cbf0 3b9d 6769 08af b660 c421 0529 d5f9 a7e9
2a0cc00 49bd 73d8 d2a4 b1c0 7854 cb25 cb10 0d21
2a0cc10 09e1 3d2a 6b50 a3e3 ed78 a0b6 1825 19d1
2a0cc20 5978 0560 8cb3 002f 1213 88fc e80c 0a08
2a0cc30 4aa7 bed1 4001 a1a2 4586 2057 bffa 0fa4
2a0cc40 8d6c 1142 0db2 20b1 eed4 cee8 ba29 3f8f
2a0cc50 d7d6 ef7d 76a5 0c14 315d bbae bed3 eec4
2a0cc60 cc60 d719 9f5b 7819 a1f8 4eb3 8a75 a07e
2a0cc70 4467 55ba 551f 4d03 837d af15 2a1a d094
2a0cc80 c1f9 889c 81a1 a580 87e8 a25b a9c0 78c5
```

b. Si un serveur utilise /dev/random pour générer les clés de session, expliquer comment ceci peut être utilisé par un attaquant pour mettre le serveur en DoS.

Si un serveur utilise **`/dev/random`** pour générer des clés de session cryptographiques, il peut devenir vulnérable à une attaque par déni de service (DoS) en cas d'insuffisance d'entropie.

En effet, lorsque le serveur utilise `/dev/random` et que l'entropie est insuffisante, la génération de clés se bloque. Si l'entropie disponible dans le système est faible (par exemple, peu d'activité clavier/souris), le serveur attend indéfiniment pour générer une clé.

Un attaquant pourrait exploiter cette situation en créant de nombreuses requêtes, forçant ainsi le serveur à générer plusieurs clés de session. Si l'entropie s'épuise, les nouvelles connexions seront bloquées, entraînant un DoS (Denial of Service), car le serveur ne peut plus traiter de nouvelles connexions.

Une solution envisageable serait d'utiliser /dev/urandom(non bloquant) pour éviter ce problème.

Exercice 4 :

Pour générer une clé de session, nous devons commencer par un seed qui est aléatoire ; sinon, le résultat sera tout à fait prévisible.
Le programme suivant utilise l'heure actuelle comme graine pour générer une clé de 128 bits :

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define KEYSIZE 16
void main()
{   int i;
    char key[KEYSIZE];
    printf("%lld\n", (long long)time(NULL));
    srand(time(NULL));
    for (i = 0; i < KEYSIZE; i++)
    {   key[i] = rand() % 256;
        printf("%.2x", (unsigned char)key[i]);}
    printf("\n");
}
```

1. Exécuter ce programme plusieurs fois successives. Quelle est votre remarque ?

Lorsqu' on exécute ce programme plusieurs fois successivement, tu remarques probablement que les clés générées sont souvent identiques ou très similaires si les exécutions sont proches dans le temps. Cela est dû au fait que la graine utilisée pour générer les nombres aléatoires (time(NULL))

est l'heure actuelle en secondes. Si tu exécutes le programme à des intervalles de moins d'une seconde, la graine sera la même et donc, les mêmes valeurs pseudo-aléatoires seront générées.

```
rahmonex  main  gcc -o keygen Exo4-keyGen.c


[~/Documents/EFREI-Crypto-M1/TP 2 - PNRG Et Chiffrement A Flot]
rahmonex  main  ./keygen

1726153583
45d343e26ca1d1f0dbc693f44c26c48a

[~/Documents/EFREI-Crypto-M1/TP 2 - PNRG Et Chiffrement A Flot]
rahmonex  main  ./keygen

1726153585
16bed1dd12599f15700d6d8b27c3315b

[~/Documents/EFREI-Crypto-M1/TP 2 - PNRG Et Chiffrement A Flot]
rahmonex  main  ./keygen

1726153586
490c45b0b10b42a0450e734b13c4608e
```

2. Commenter la ligne srand(time(NULL)); puis réexécutez le programme plusieurs fois. En déduire le rôle de srand.

```
[~/Documents/EFREI-Crypto-M1/TP 2 - PNRG Et Chiffrement A Flot]
rahmonex  main  gcc -o keygen Exo4-keyGen.c


[~/Documents/EFREI-Crypto-M1/TP 2 - PNRG Et Chiffrement A Flot]
rahmonex  main -  ./keygen

1726208974
67c6697351ff4aec29cdbaabf2fbe346

[~/Documents/EFREI-Crypto-M1/TP 2 - PNRG Et Chiffrement A Flot]
rahmonex  main -  ./keygen

1726208974
67c6697351ff4aec29cdbaabf2fbe346

[~/Documents/EFREI-Crypto-M1/TP 2 - PNRG Et Chiffrement A Flot]
rahmonex  main -  ./keygen

1726208976
67c6697351ff4aec29cdbaabf2fbe346
```

En enlevant le srand(time(NULL), on remarque que la clé générée est toujours la même à chaque exécution.

- **`srand()`** est utilisé pour initialiser (ou "semer") le générateur de nombres pseudo-aléatoires avec une valeur de départ, appelée "seed". Sans appeler srand(), le générateur de nombres aléatoires utilise une valeur de seed par défaut, ce qui signifie qu'il produit toujours la même séquence de nombres aléatoires à chaque exécution du programme.
- **`srand(time(NULL))`** initialise le générateur de nombres aléatoires avec l'horloge actuelle en secondes, ce qui garantit une séquence différente à chaque exécution, car la valeur de `time(NULL)` change à chaque seconde.

Sujet :
Le 17 avril 2018, Alice a terminé sa déclaration de revenus et elle l'a sauvegardée (un fichier PDF) sur son disque. Pour protéger le fichier, elle a chiffré le fichier PDF à l'aide d'une clé générée à partir du programme décrit dans la question 1. Elle a noté la clé dans un cahier, qui est stockée en toute sécurité dans un coffre-fort. Quelques mois plus tard, David est entré par effraction dans son ordinateur et a obtenu une copie de la déclaration de revenus chiffrée. Comme Alice est PDG d'une grande entreprise, ce dossier est très précieux.

David ne peut pas obtenir la clé de chiffrement, mais en regardant autour de l'ordinateur d'Alice, il a vu le programme de génération de clé, et a soupçonné que la clé de chiffrement d'Alice peut être générée par le programme. Il a également remarqué l'horodatage du fichier, qui est « 2018-04-17 23:08:49 ». Il a deviné que la clé peut être générée dans une fenêtre de 24 heures avant la création du fichier.

Puisque le fichier est un fichier PDF, la partie initiale de son en-tête est toujours le numéro de version. À l'époque où le fichier a été créé, PDF-1.5 était la version la plus courante, c'est-à-dire que l'en-tête commence par %PDF-1.5, soit 8 octets de données. Les 8 octets suivants des données sont également assez faciles à prévoir. Par conséquent, David a facilement obtenu les 16 premiers octets du texte en clair. Sur la base des métadonnées du fichier crypté, il sait que le fichier est crypté à l'aide de aes-128-cbc. Étant donné que AES est un chiffrement de 128 bits, le texte en clair de 16 octets se compose d'un bloc de texte en clair, donc David connaît un bloc de texte en clair et son texte chiffré correspondant. De plus, David connaît également le vecteur initial (IV) du fichier chiffré (IV n'est jamais crypté). Voici ce que David sait :

```
Plaintext: 255044462d312e350a25d0d4c5d80a34
Ciphertext: d06bf9d0dab8e8ef880660d2af65aa82
IV: 09080706050403020100A2B2C2D2E2F2
```

Votre travail consiste à aider David à trouver la clé de chiffrement d'Alice, afin que vous puissiez déchiffrer l'ensemble du document. Vous devez écrire un programme pour essayer toutes les clés possibles. Si la clé a été générée correctement, cette tâche ne sera pas possible. Cependant, comme Alice a utilisé time() pour semer son générateur de nombres aléatoires, vous devriez pouvoir trouver sa clé.

3. Trouver la clé du document d'Alice.

L'idée est d'exploiter le fait qu'Alice a utilisé la fonction `time()` comme graine pour son générateur de nombres aléatoires lors de la génération de la clé de chiffrement. Le problème avec `time()` est qu'il est basé sur l'horodatage (nombre de secondes depuis le 1er janvier 1970), ce qui le rend prévisible dans une certaine fenêtre de temps.

Informations dont nous disposons :

- Horodatage du fichier : 2018-04-17 23:08:49. Nous savons qu'Alice a généré la clé dans sur une plage de 12 heures avant et 12 heures après (43200 secondes de part et d'autre de l'horodatage).
- En-tête du fichier PDF (texte en clair sur 16 octets) :
    - Texte en clair : `255044462d312e350a25d0d4c5d80a34` (en hexadécimal).
- Texte chiffré correspondant (AES-128-CBC) :
    - Texte chiffré : `d06bf9d0dab8e8ef880660d2af65aa82` (en hexadécimal).
- Vecteur d'initialisation (IV) : 09080706050403020100A2B2C2D2E2F2.

AES en mode CBC (Cipher Block Chaining) chiffre les blocs de 16 octets. Chaque bloc est chiffré en combinant le texte en clair avec le bloc chiffré précédent (ou l'IV pour le premier bloc). Nous connaissons le texte en clair et chiffré du premier bloc, ce qui nous permet de tester différentes clés pour tenter de les retrouver.

Schéma cours : —————————— ici

On a aussi 2018-04-17 23:08:49 en timestamp UNIX donne : 1524004129

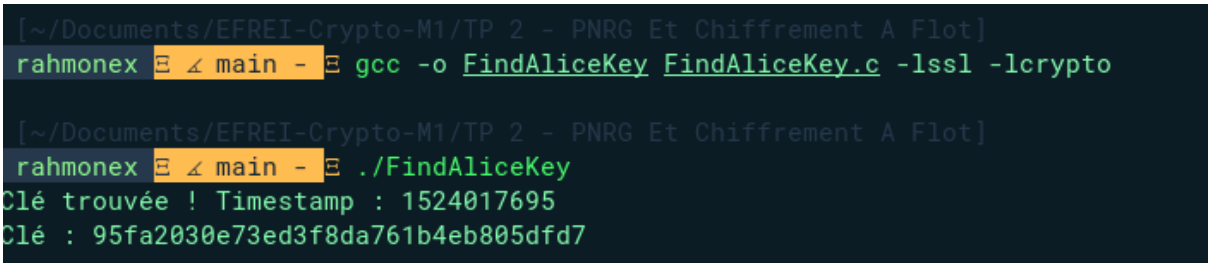
Pour exécuter le code on doit tout d'abord installer OpenSSL sur Linux :

```
sudo apt update
sudo apt install libssl-dev
```

Et ensuite on exécute le code avec cette commande ou l'on doit lier les bibliothèques libssl et libcrypto.

```
gcc -o FindAliceKey FindAliceKey.c -lssl -lcrypto
```

On obtient :



Donc pour ce Timestamp : 1524017695 on trouve la bonne clé.
Clé : 95fa2030e73ed3f8da761b4eb805dfd7

Le timestamp 1524017695 correspond à la date et heure suivante en UTC : 18 avril 2018, 03:34:55 UTC

Démarche du code :

Génération de la clé : Le programme génère des clés en utilisant une graine (`seed`) qui correspond à un horodatage (timestamp) spécifique. La graine est utilisée par la fonction `srand()` pour initialiser le générateur de nombres pseudo-aléatoires.
Chiffrement : Le texte en clair connu est chiffré avec AES-128-CBC en utilisant la clé générée et le vecteur d'initialisation (IV) fourni.
Comparaison : Le texte chiffré généré est comparé au texte chiffré connu. Si les deux blocs correspondent, cela signifie que la clé correcte a été trouvée.
Recherche dans une plage de temps : Le programme teste toutes les graines possibles dans une plage de **24 heures avant** et **24 heures après** l'horodatage du fichier PDF (timestamp), ce qui correspond à la fenêtre où Alice aurait pu générer la clé.
Sortie : Si la clé est trouvée, elle est affichée avec son horodatage. Si aucune clé n'est trouvée dans la plage de temps donnée, un message d'échec est affiché.