GLMF, le Magazine du Manchot Libre et Productif, présente :

Sciences/Théorie de l'information : propriétés et dérivés des CRC et LFSR

Yann Guidon

EN DEUX MOTS **Cet été, la grande saga des algorithmes de CRC ne s'arrête pas. Pour continuer dans la lancée de l'article sur les corps de Galois, qui a expliqué en détail « pourquoi et comment fonctionnent les LFSR », nous allons maintenant faire le rapprochement avec les CRC, puis étudier un dérivé, que j'ai baptisé « pseudo-CRC ». Les applications pratiques sont trop longues pour tenir dans cet opus, qui complémente cependant les explications parues dans GLMF n°78 et n°81 : par référence à l'article sur les LFSR, celui-ci pourrait s'appeler « pourquoi et comment fonctionnent les CRC ? ».**

Résumé des épisodes précédents

Dans le cadre de la conception d'un format de flux multiplexé de données compressées, je cherche à mettre au point un algorithme calculant une *empreinte*, ou *signature* numérique de blocs, afin de détecter des altérations éventuelles lors de leur transmission ou de leur stockage. Cette signature (non cryptographique) est sur 16 bits, pour des blocs de données dont la taille est comprise entre 1 et 4096 octets. Le critère essentiel est bien sûr la vitesse d'exécution, mais sans compromettre la portabilité, l'efficacité et la facilité de codage.

L'article paru dans GLMF n°78 a justifié les choix des paramètres de notre CRC, puis passé en revue les techniques de codage d'un CRC 16 bits à base de table. L'algorithme a été validé par des tests, suivis d'une campagne de mesures de performance sur un éventail hétéroclite de plateformes. Ensuite, c'est le coup classique : en me relisant dans le magazine imprimé, je me suis rendu compte que j'avais oublié une méthode bien connue, utilisée par les générateurs de nombres pseudo-aléatoires à base de registres à décalage (« LFSR »).

Le numéro 81 de mars 2006 expose donc les bases théoriques et pratiques pour générer des quantités gigantesques de nombres pseudo-aléatoires à haute vitesse. Cela n'était pas une fin en soi et nous allons maintenant comprendre pourquoi cela peut aussi nous aider avec nos CRC, que je vais allègrement charcuter et hybrider...

Petits rappels

Les algorithmes de CRC (*Cyclic Redundancy Codes* ou Codes Redondants Cycliques en français) permettent de *signer* des blocs de données afin d'y détecter ultérieurement d'éventuelles altérations. C'est un peu comme une empreinte MD5 ou SHA, mais sans toutes les contraintes cryptographiques, remplacées par des exigences de vitesse et de compacité. Statistiquement, la probabilité qu'une ou des erreurs ne changent pas la signature (un *faux positif*) est inversement proportionnelle à l'exponentielle de la taille de la signature (en 2^-*n*), donc une signature longue est préférable.

Bien que le CRC 32 bits soit actuellement le plus utilisé (notablement par les trames Ethernet ou dans la Zlib, avec le polynôme 0x04C11DB7 déjà utilisé dans d'autres articles), la version 16 bits que nous étudions a de nombreuses applications (voir la **table 1**), en particulier quand les blocs à signer sont courts.

En effet, par rapport à des blocs longs (dizaines ou centaines de kilo-octets), la probabilité de capter une erreur accidentelle est plus faible, ce qui diminue la taille de la signature et économise un peu de place dans chaque bloc par la même occasion. D'autres facteurs sont en faveur d'une signature sur 16 bits (au lieu de 32 bits) : les erreurs de transmission ou de stockage sont rares en pratique, les conséquences ne sont pas critiques et le surcoût d'une signature longue peut contrebalancer les gains de compression.

Un algorithme de CRC est défini par plusieurs paramètres : sa taille (le nombre de bits de la signature), son *polynôme* (c'est la manière dont les bits sont mélangés entre eux), sa valeur initiale (ou *seed*) ainsi que d'autres détails optionnels qui ne nous concernent plus ici (ils ont été traités en décembre et nous n'en aurons plus besoin avec nos nouveaux algorithmes).

Nom	Polynôme	Utilisation
CRC-16	0x8005	en-têtes MPEG/MP3 (seed=FFFFh), ARC (seed=0000h)
CRC-CCITT	0x1021	X25 (seed=FFFFh), ATA/IDE en mode UDMA (seed=4ABAh)
CRC-CCITT réfléchi	0x8408	XMODEM (seed=0000h)
CRC-DNP	0x3D65	« *Distributed Network Protocol* » (réseaux industriels)
CRC-16T10-DIF	0xCBB7	

Table 1 : Quelques exemples de CRC16 et leurs applications

Pour schématiser grossièrement, un algorithme de CRC calcule une division un peu spéciale :

- le nombre à diviser est le flux de données à signer ;
- le diviseur est le fameux polynôme ;
- le reste est le résultat désiré (le « CRC ») ;
- le dividende est ignoré.

Ensuite, la division utilise une arithmétique spéciale, appelée GF(2) par les intimes, où l'addition et la soustraction que nous connaissons sont réalisées simplement par un XOR. Il en découle que la multiplication et la division dans GF(2) sont une succession de décalages et de XOR, très faciles à réaliser en matériel comme en logiciel.

Autre détail important, le diviseur a des propriétés mathématiques spéciales et il est représenté soit sous forme d'un *polynôme*, soit sous forme condensée numérique. Il n'est pas pris au hasard, car il doit répondre à certaines conditions, en particulier être *primitif* (notion que j'ai tenté d'expliquer dans l'article précédent de mars 2006), ce dont nous allons bientôt reparler.

D'un autre côté, un LFSR (*Linear Feedback Shift Register* ou Registre à Décalage à Rétroaction Linéaire en français, mais ça perd son charme anglo-saxon) permet de générer une séquence cyclique, d'une période arbitrairement longue et aux propriétés statistiques bien connues. C'est pratique lorsqu'on veut soumettre un programme à des tests intensifs et *a priori* aléatoires.

Comme la séquence est cyclique, elle n'est pas réellement aléatoire, mais c'est un substitut pratique et convenable dans certaines conditions : en raison de leur simplicité et de leur vitesse, on retrouve des LFSR, par exemple, dans les téléphones GSM et dans les GPS.

Théorie unifiée des registres à décalage

Nous avons vu dans les deux précédents articles que les LFSR et les CRC sont très liés. Ils utilisent les mêmes mathématiques, donc les mêmes polynômes, puisqu'un CRC est une division dans GF(2) et un LFSR peut aussi être vu comme une division (ou une multiplication, selon le bout par lequel on regarde).

Ils utilisent aussi les mêmes structures, donc les mêmes algorithmes : on peut réutiliser l'organisation *bit à bit* lente, ou une version plus rapide avec une table (comme dans l'article de décembre).

Pourtant, on ne peut pas réaliser directement un LFSR avec un CRC et vice versa. Il y a des règles à respecter (du moins, tant qu'on se préoccupe des conventions).

Admettons qu'aucun mathématicien fondamentaliste ne lise ceci, quelle est alors la différence essentielle (pour un informaticien) entre un LFSR et un CRC ? Revenons à leur fonction respective : le CRC calcule une empreinte d'une donnée, alors que le LFSR génère ses nombres en vase clos, sans entrée continuelle d'information.

Comme l'illustre la **figure 1**, l'unique différence structurelle tient à ce petit XOR du CRC, entre la donnée binaire à signer et la rétroaction. Au contraire, le LFSR fournit la valeur de cette rétroaction.

Si nous ne nous préoccupons pas des conditions d'initialisation ou de la consultation de la signature après calcul, il n'y a rien d'autre. Les mêmes contraintes de primitivité du polynôme sont à respecter, si on veut que le nombre de cycles avant rebouclage soit maximal.

Par exemple, le polynôme *(8,4,3,2,0)* utilisé sur la figure 1 est primitif et garantit que les (2^8)-1 états possibles seront visités.

Donc en gros, un CRC est un LFSR en configuration de Galois, « perturbé » par les données à signer.

Encore des histoires de configuration

Bon d'accord, ce n'est pas aussi simple que je voudrais le laisser penser, mais l'objectif est de *signer* des données, pas nécessairement avec un algorithme de CRC orthodoxe (que nous allons ici jeter aux orties).

Il existe en fait une petite différence concernant le registre à décalage en lui-même et la manière dont la rétroaction est appliquée. Un LFSR fonctionne aussi bien en configuration de Galois ou de Fibonacci, en donnant toutefois un résultat différent pour un même polynôme. Nous avons vu

Sans être absolument nécessaire, il est très fortement recommandé d'avoir lu et assimilé les articles présents dans GLMF 78 et 81 avant d'entamer la lecture de celui-ci.

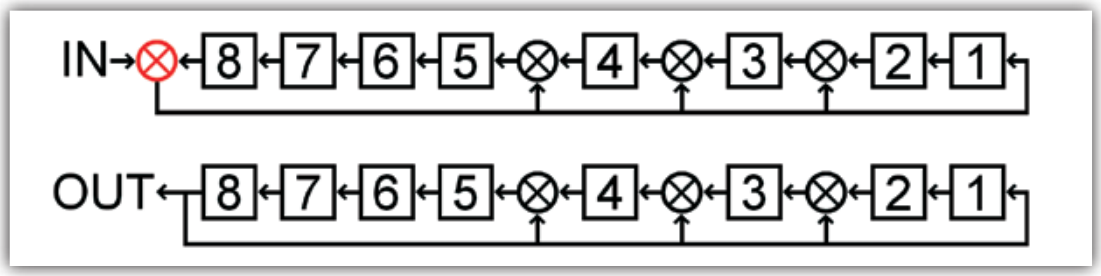

Fig. 1 : En un dessin, voici la différence entre un CRC (en haut) et un LFSR (en bas)

en mars qu'il suffit d'inverser l'ordre des facteurs du polynôme, ou bien l'ordre des bits de la séquence, pour retrouver une séquence identique. Cependant, un CRC ne se réalise qu'avec une configuration de Galois, en raison de son lien direct avec la division polynomiale.

Pourquoi s'intéresser à ce détail ? L'article de mars 2006 explique qu'une configuration de Fibonacci simplifie un peu le codage en enlevant des écritures.

Toute simplification est bonne à prendre, en particulier si cela ne change pas le résultat final (pour un LFSR). L'accélération est probablement minime, peut-être 10% après de nombreuses optimisations, mais c'est le type d'économie facile à réaliser, car elle intervient avant de commencer à coder.

PRÉCISION

L'effet réel dépend en fait de l'architecture du processeur qui exécute le programme. Les machines CISC comme le x86 peuvent coder l'opération complexe suivante :

```
a ^= b; /* a est en mémoire et b dans un registre*/
```

avec une seule instruction. Par contre, les processeurs RISC (par exemple MIPS, SPARC, PowerPC, ALPHA...) ont une architecture load-store et ne permettent pas d'effectuer des opérations complexes avec des variables en mémoire. Il faut donc découper l'opération précédente en trois morceaux pour travailler avec les registres, selon le pseudo-code suivant :

```
t=mémoire[a]; /* "load" */
t=t^b;        /* opération */
mémoire[a]=t; /* "store" */
```

Comme nous visons la portabilité sur le plus grand nombre possible de machines, nous ne pouvons pas compter sur la disponibilité d'instructions complexes avec la mémoire (souvent appelées read-modify-write).

La configuration de Fibonacci est préférée, car elle économise une opération d'écriture. Ce n'est pas désirable uniquement pour les processeurs RISC : les instructions CISC prennent moins de place, mais pas moins de temps à exécuter (il faut toujours effectuer la même chose, après tout).

Plus tard, le déroulage des boucles (et le stockage dans les registres qui en résulte) rendra éventuellement tout cela inutile... Mais l'important est que toutes les méthodes d'optimisation possibles soient employées.

Pré-titre fourni par l'auteur : «GLMF, le Magazine du Manchot Libre et Productif, présente...»

Quelle est l'importance réelle de la configuration pour un CRC ?

- La réponse officielle est qu'un CRC en configuration de Fibonacci n'est plus un CRC, puisque ce n'est plus une division polynomiale.
- La réponse pratique est que, en plus de donner des séquences d'états différentes, les deux configurations se comportent différemment dans les « cas pathologiques » (que nous essayons d'éviter). En particulier, il y a deux états dans lesquels un CRC en configuration de Fibonacci peut rester bloqué (c'est une configuration symétrique), contre un seul en configuration de Galois (asymétrique).

Ces cas particuliers peuvent être explorés au moyen du code suivant, dont les parties en surbrillance doivent être changées pour tester les 8 combinaisons.

```
/* fichier cycles.c
    Fonction : compte le nombre d'états du registre de CRC (Poly=0x8005)
     dans un cycle en partant d'un état et d'une configuration donnée
 */
#include <stdio.h>

#define CRC16_Galois(registre, data_in)   \
  registre = (registre+registre) ^ (0x8005 & \
    (((signed short int) (registre ^ ((data_in & 1) << 15))) >> 15));
#define CRC16_Fibonacci(registre, data_in) \
  registre = (registre+registre)           \
           ^ (1 & (data_in                 \
                 ^ (registre >> 1)         \
                 ^ (registre >> 14)        \
                 ^ (registre >> 15)  ));

#define SEED 0

int main(void) {
  int i=0;
  unsigned short int lfsr=SEED;

  do {
    i++;
    CRC16_Fibonacci(lfsr,1)
    printf("%05d\n",lfsr);
  } while (lfsr != SEED);

  printf(" %d itérations\n",i);
  return 0;
}
```

Si on ne se préoccupe pas trop de ce point, les autres caractéristiques principales d'un LFSR normal sont conservées.

EN RÉSUMÉ

- *Le rapport entre la configuration de Galois et la configuration de Fibonacci a été effectué en mars pour les LFSR : c'est la même chose pour nous, « à un ou deux détails près ».*
- *En unifiant les CRC et les LFSR, on peut déduire qu'il est possible de créer un CRC en configuration de Fibonacci, sans perdre trop de propriétés fondamentales.*
- *Mais mathématiquement, nous n'avons plus affaire à un CRC normal. Imaginons le terme de pseudo-CRC pour suggérer l'étrangeté de l'algorithme.*

Configuration	Seed	data_in	Longueur du cycle	Commentaires
Galois	0	0	1	Le registre reste à 0 : un CRC doit être initialisé à une valeur non nulle.
		1	65534	Les états 0 et 65535 sont visités
	0xFFFF	0	32767	L'état 0 n'est pas visité
		1	65534	Les états 0 et 65535 sont visités
Fibonacci	0	0	1	Le registre reste à 0
		1	65534	Les états 0 et 65535 sont visités
	0xFFFF	0	1	Le registre reste à 0xFFFF
		1	65534	Les états 0 et 65535 sont visités

Table 2 : Comportement d'un CRC en fonction des conditions initiales et de la configuration
Note 1 : 65534=32767×2, il n'y a donc pas d'erreur « off-by-one »
Note 2 : Les résultats dépendent aussi du polynôme utilisé (primitivité, nombre de termes...)

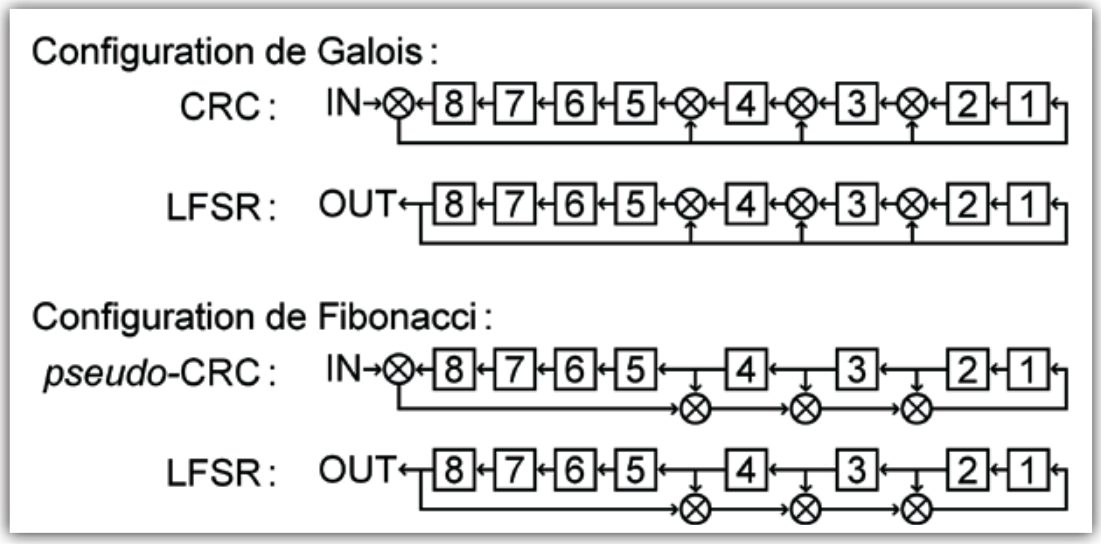

Fig. 2 : Variations sur le thème d'un registre à décalage

Détection et propagation d'altérations

Quand on joue avec le feu mathématique (comme je le fais ici), il ne suffit pas de dire que « ce n'est pas grave » et « cela nous suffit », il faut aussi vérifier. Un détail particulier me chiffonnait et j'ai donc repris le programme précédent pour comparer la capacité de chaque configuration à détecter et conserver une altération.

Le programme `avalanche.c` permet de visualiser la propagation d'une différence entre deux registres identiques, mais où un bit est modifié. La sortie du programme a été utilisée directement pour créer la **figure 3**.

```
/*
  fichier avalanche.c
  fonction : illustrer la différence d'effet d'avalanche
    entre une configuration de Fibonacci et de Galois
  compilation :
    gcc -Os -o avalanche avalanche.c
*/

char message_original[]="GNU/Linux Magazine France";
char message_modifie []="gNU/Linux Magazine France";

#define SEED 0xFFFF

..... snip! .....

int main(void) {
  int i, j;
  unsigned short int
    lfsr_galois=SEED,        lfsr_fibonacci=SEED,
    lfsr_galois_altere=SEED, lfsr_fibonacci_altere=SEED;
  char c, d;

  for (i=0; i<sizeof(message_original)-1; i++) {
    c=message_original[i];
    d=message_modifie[i];

    for (j=0; j<8; j++) {
      CRC16_Galois(lfsr_galois,c)
      CRC16_Galois(lfsr_galois_altere,d)
      CRC16_Fibonacci(lfsr_fibonacci,c)
      CRC16_Fibonacci(lfsr_fibonacci_altere,d)

      affiche_binaire(lfsr_galois^lfsr_galois_altere);
      affiche_binaire(lfsr_fibonacci^lfsr_fibonacci_altere);
```

```
      printf("\n");

      c>>=1;
      d>>=1;
    }
  }
}
```

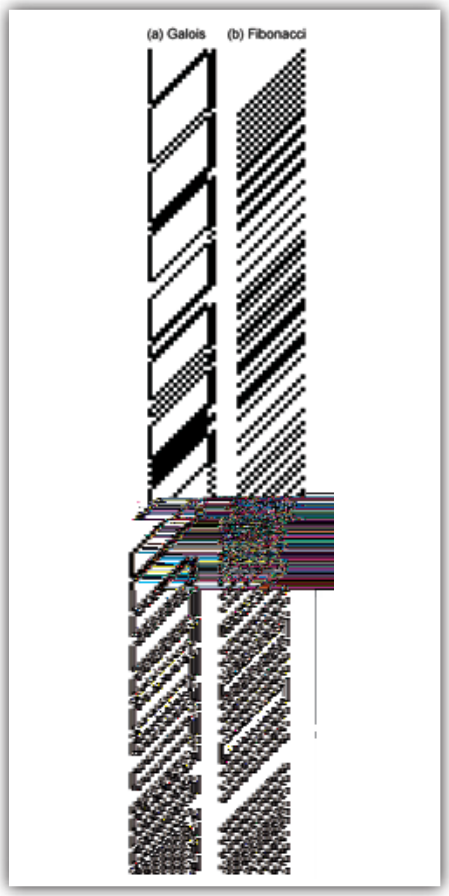

Fig. 3 : Propagation d'une altération dans le registre à décalage, en fonction de la configuration

Dans la plupart des cas, le *pseudo*-CRC propage très vite l'altération (en moyenne, 8 bits sur 16 sont modifiés) alors que le CRC classique exhibe des petits motifs presque cycliques, avec une différence de deux ou trois bits en moyenne.

Notre *pseudo*-CRC effectue son travail, et même mieux que je ne l'imaginais au début. Bien sûr, le nombre de bits de différence baisse rapidement, mais je pensais initialement que le CRC normal propagerait l'altération plus vite que notre *pseudo*-CRC, et le contraire se produit (d'où l'importance de vérifier).

On peut jouer longtemps avec ce programme, à créer plein de figures amusantes. Mais elles me rappellent fortement quelque chose... Vite, je modifie légèrement le source et je vois que le résultat est identique :

```
#define SEED 0 /* le seul changement imprévu */

..... snip! .......

    for (j=0; j<8; j++) {
      CRC16_Galois(lfsr_galois,c^d)
      CRC16_Fibonacci(lfsr_fibonacci,c^d)
      affiche_binaire(lfsr_galois);
      affiche_binaire(lfsr_fibonacci);
      printf("\n");

      c>>=1;
      d>>=1;
    }
```

En fait, étudier la propagation d'une différence entre deux messages est exactement la même chose que d'étudier la différence elle-même.

$$CRC(M) \text{ xor } CRC(M \text{ xor } \delta) = CRC(\delta)$$

C'est d'ailleurs une propriété connue puisqu'un CRC est le reste d'une division, on peut déduire la formule précédente de celle-ci :

$$(A+\delta \pmod x) - (A \pmod x) = \delta \pmod x$$

On reste en fait en pleine *arithmétique modulo x*. Donc, si on veut étudier la capacité à propager une différence, il suffit d'étudier le CRC de la différence elle-même, sans s'embarrasser avec deux CRC.

La question initiale concernait la détection et la conservation d'une altération. Nous pouvons maintenant y répondre en utilisant les formules précédentes ainsi que le **tableau 2**.

- Lorsqu'il n'y a aucune altération, le XOR entre les deux messages est nul. Nous nous trouvons dans le cas où SEED==0 et data_in==0 : le registre reste à 0 et donc aucune altération ne peut apparaître spontanément. C'est un état stationnaire, qui justifie pourquoi un registre de CRC ne doit pas être initialisé à 0 : il ne détecterait pas d'altération en début de bloc.
- Lorsqu'une erreur d'un bit survient, cela correspond à SEED==0 (pas d'erreur auparavant) et data_in==1 : le registre passe à un état non nul (1, dans la configuration de Fibonacci). C'est la transition vers l'état indiquant une altération.
- Lorsque la signature se poursuit sans subir d'autre altération, cela revient à SEED!=0 et data_in==0, donc à un LFSR, puisque A XOR 0 = A. Un LFSR de *n* bits a la propriété de passer par une suite cyclique de (2^n)-1 états, à l'exception de 0, l'altération ne peut donc pas s'effacer d'elle-même.

On en déduit qu'un *pseudo*-CRC, tout comme un CRC normal, détecte et conserve une altération d'un bit, quel que soit le nombre de bits à signer.

Altérations multiples

Cela devient un peu plus complexe lorsque plusieurs altérations surviennent. Les altérations multiples qui ne sont pas détectables par les CRC sont bien connues, mais un *pseudo*-CRC a des propriétés un peu différentes.

Leonardo Fibonacci était connu de son temps sous les noms de Leonardo Pisano ou encore Leonardo Bigollo. Mais son véritable patronyme était en réalité Guilielmi.

Pour les comprendre, repartons de la définition fondamentale de l'arithmétique modulo x :

$$A \pmod x = (A+\delta x) \pmod x, \text{ si } \delta \text{ est un entier relatif}$$

Si on ajoute un nombre entier de *x*, le résultat *modulo x* sera toujours le même. Appliqué à un CRC dont le polynôme est P sur un message M, on trouve :

$$CRC(M \text{ xor } \delta P) = CRC(M)$$

Dans ce contexte, $\delta \times P$ correspond aux erreurs qui ne sont pas détectables, puisqu'elles sont multiples (dans GF(2)) du polynôme utilisé. Ainsi, en effectuant un XOR du message signé avec la représentation binaire du polynôme, on peut fausser le message sans que cela ne soit détectable.

ATTENTION

En sélectionnant la bonne altération, on peut même « gommer » ou dissimuler des modifications arbitraires. Cette technique est très facile à appliquer pour corrompre des fichiers. En plus des modifications arbitraires, il faut aussi une zone de « compensation » de la signature, dont la longueur est égale à cette dernière. En effet, si le CRC est sur N bits, il suffit de contrôler N bits du message pour contrôler entièrement la signature et faire croire que rien ne s'est passé.

Il n'est même pas nécessaire de connaître la signature du fichier entier, celle du bloc qui sera altéré suffit. Si la signature de la zone modifiée reste la même, la signature finale ne changera pas non plus.

Moralité : quand la sécurité est en jeu, il faut uniquement faire confiance à des algorithmes de qualité cryptographique comme MD5 ou SHA (et les utiliser correctement).

Si un *pseudo*-CRC n'est pas une division dans GF(2), la forme d'un *faux positif* est-elle encore la même qu'avec un CRC normal ? Et est-il sensible aux mêmes erreurs ? Plus généralement, les mêmes mathématiques s'appliquent-elles ? Le seul moyen que j'ai trouvé pour répondre à ces questions existentielles est d'écrire un autre programme.

```
/*
  fichier positif.c
  fonction : étudier les formes des "faux positifs"
    avec les configurations de Fibonacci et de Galois
  compilation :
    gcc -Wall -Os -o positif positif.c
   Attention : Ne marche correctement que sur machine Little
Endian !
*/

..... snip! .......

char message[]="GNU/Linux Magazine France";

#define SEED 12345 (peu importe la valeur initiale)
#define DELTA 9 (voir la remarque dans le texte)

void calcul_CRC() {
  int i, j;
  unsigned short int
    lfsr_galois=SEED,
    lfsr_fibonacci=SEED;
  char c;

  for (i=0; i<sizeof(message)-1; i++) {
    c=message[i];

    for (j=0; j<8; j++) {
      CRC16_Galois(lfsr_galois,c)
      CRC16_Fibonacci(lfsr_fibonacci,c)
      c>>=1;
    }
  }

  printf("CRC Galois : 0x%04X, CRC Fibo : 0x%04X\n",
       lfsr_galois, lfsr_fibonacci);
}

int main(void) {
   /* calcul du CRC normal */
  calcul_CRC();

   /* modification du message (! little endian !) */
  *(long int *)message ^= DELTA *
#ifdef REVERSE
     ((1 << 16) | (1 << 14) | (1 << 1) | (1 << 0));
#else
     ((1 << 16) | (1 << 15) | (1 << 2) | (1 << 0));
#endif
  /* calcul du nouveau CRC */
  calcul_CRC();

  return 0;
}
```

La première remarque importante concerne l'ordre des bits du polynôme. Les macros de CRC travaillent en décalant les bits vers la droite (donc en partant du LSB), il faut leur fournir les bits du polynôme dans l'ordre inverse de la lecture. C'est pour cela que l'on se sert de #ifdef REVERSE.

L'autre détail important concerne le multiplicateur (DELTA). Il est fourni ici afin de montrer qu'on peut générer très facilement des faux positifs arbitraires. Mais le langage C et les ordinateurs usuels effectuent la multiplication *normale* (en propageant les retenues), donc tous les multiplicateurs ne conviennent pas.

Heureusement, le polynôme utilisé est très clairsemé et il est facile de trouver des multiplicateurs ne générant pas de retenue et donc équivalents dans GF(2). La multiplication classique est juste utilisée ici pour simplifier le codage...

Le résultat est facile à obtenir et à comprendre :

```
~# gcc -Os -Wall -o positif positif.c && ./positif (REVERSE n'est pas défini)
CRC Galois : 0x5372, CRC Fibo : 0xFEA3 (résultat original)
CRC Galois : 0xACA9, CRC Fibo : 0xFEA3 (faux positif après altération)
```

Dans cet exemple, REVERSE n'est pas défini, le polynôme subit donc une réflexion par rapport à l'ordre attendu et le *pseudo*-CRC rencontre un faux positif.

Si `REVERSE` est défini, c'est le CRC normal qui tombe dessus. Les réponses aux questions précédentes sont alors claires, même si leur justification n'est pas évidente :

- La forme d'un faux positif est « presque la même » que pour un CRC normal. **La seule différence est que le sens des bits du polynôme est inversé.**

Ce n'est pas surprenant, puisque nous avons observé, dans l'article de mars, qu'un LFSR en configuration de Galois fournit la même suite de bits qu'un LFSR en configuration de Fibonacci, mais dans le sens inverse.

- Puisque seul le sens des bits change, les mêmes types d'erreurs sont détectables. On peut aussi créer des faux positifs de la même manière, ce qu'il est facile de vérifier en jouant sur le paramètre `DELTA` (en faisant évidemment attention à la multiplication).

Ce qui est le plus frappant est que le *pseudo*-CRC n'est pas une division polynomiale, mais il en a les propriétés les plus importantes et intéressantes !

EN RÉSUMÉ

Après cette dernière vérification, il n'y a plus de doute : un pseudo-CRC est aussi valable qu'un CRC normal.

Mais qu'il s'agisse d'une division polynomiale, d'une multiplication martienne ou d'une exponentiation plutonienne, la seule chose qui nous importe est que nous comparerons simplement une signature reçue avec celle que nous venons de recalculer. Une différence d'un seul bit entre deux signatures est nécessaire et suffisante pour signaler une altération, l'efficacité de notre programme dépend de cette unique condition.

Genèse d'un polynôme

Même si nous n'avons plus affaire à un vrai CRC, le reste n'a pas changé pour autant. En particulier, notre bon vieux polynôme `0x8005` a été testé en décembre et, bien qu'il ne soit pas de toute première fraîcheur, ses propriétés nous arrangent beaucoup. Sa description exacte est *(16,15,2,0)*, ce qui correspond dans `GF(2^16)` à *x^16+x^15+x^2+1*.

Ce polynôme n'est pas arbitraire, mais a été construit très simplement. Dans leurs ouvrages respectifs, Tanenbaum et Ross N. Williams expliquent que pour détecter les altérations en nombre pair, il faut que le polynôme soit multiple de **11** (en représentation binaire, ou *x+1* en représentation polynomiale). Et en 1998, Terry Ritter écrivait dans un fil de discussion sur `news:comp.arch` :

« *So, no, CRC-CCITT is *not* irreducible. (Which also means it is *certainly* not primitive!). These first-generation standard CRC's (along with CRC-16) were constructed by taking an irreducible times the poly 11. The poly 11 effectively provides the equivalent of the parity used in the earlier systems. As I recall, the argument was that CRC would thus never be worse than parity.* »

Traduction : pendant la conception des premiers CRC, pour s'assurer qu'ils se comporteraient au moins aussi bien qu'un bit de parité, on prenait un polynôme primitif multiplié par le polynôme de parité *(1,0)*. Et quand on divise *(16,15,2,0)* par *(1,0)*, on obtient *(15,1,0)*, c'est-à-dire une version de notre trinôme favori !

IMPORTANT

J'avais écrit en mars :

« 'On sait que' le trinôme x^n+x+1 est primitif dans `GF(2^n)` pour

n = 1, 3, 4, 6, 9, 15, 22, 28, 30, 46, 60, 63, 127, 153, 172... »

En réalité, j'avais simplement recopié cette liste à partir du livre de Bruce Schneier, mis en confiance par certaines valeurs que je savais correctes. Les autres devaient donc être bonnes.

*Mais Nicolas, un lecteur attentif, a trouvé que certains nombres ne fonctionnaient pas. J'ai alors vérifié moi-même les nombres inférieurs à 30 et j'ai trouvé que seuls **3, 4, 6, 7, 15, 22** sont valides.*

Cela m'apprendra à recopier un livre. Même s'il s'agit d'une seule ligne, on tombe forcément sur une bourde. Dans ce cas, je me suis basé sur le livre Cryptographie appliquée (2è édition française) de Bruce Schneier, traduit en français par Laurent Viennot (qui lui-même a laissé plein de fautes de Français). Page 271, Chapitre 11.3, il y est écrit :

« Pour le corps de GALOIS Z/2^N, les spécialistes de cryptographie aiment utiliser le trinôme p(x)= x^n + x + 1 comme module car la longue série de coefficients nuls entre x^n et x rendent la réalisation de la multiplication très efficace [190]. Le trinôme doit être primitif, sinon cela ne fonctionne pas. Pour n inférieur à 1000, le trinôme x^n + x + 1 est primitif [1651, 1650] si n vaut :

1, 3, 4, 6, 9, 15, 22, 28, 30, 46, 60, 63, 127, 153, 172, 303, 471, 532, 865, 900 »

On peut penser qu'un trinôme primitif existe pour tous les champs de Galois d'une telle taille (un contre-exemple est `GF(2^8)` qui n'a pas de trinôme primitif du tout), mais ce n'est pas toujours le trinôme qui nous intéresse (c'est-à-dire (n,3,0) ou (n,4,0), mais pas (n,1,0)).

Comme tous les polynômes de la **table 1**, celui de CRC16 contient un nombre pair de termes et n'est donc pas primitif, sans que cela ne pose de problème réel : l'objectif d'un CRC n'est pas d'être de période maximale, on veut juste détecter un maximum d'erreurs. Nous venons d'ailleurs de voir que la capacité à détecter et conserver la trace d'une altération ne dépend pas de la période du LFSR correspondant.

Nous avons aussi vérifié en décembre que le polynôme 0x8005 fonctionne comme on s'y attend, avec *dans le pire des cas*, en moyenne un faux positif pour 2^16 signatures.

Un polynôme indétrônable

Par contre, nous avons aussi vérifié expérimentalement que dans le contexte qui nous intéresse (c'est-à-dire lorsque la signature est ajoutée au bloc de données et donc susceptible de subir les mêmes altérations), l'effet de la détection de parité est nul (cela ne change pas le taux de faux positifs). Et dans le cas où on empêche l'altération de la signature, toutes les erreurs de parité sont bien détectées, *mais* les autres erreurs ont un taux de faux positif double.

Quand j'ai lu ce résultat pour la première fois, j'ai d'abord cru à un autre bug dans mon programme de test, mais, avec le recul, cela semble maintenant logique. Le trinôme générateur principal *(15,1,0)* ne travaille que sur 15 bits, donc produit en moyenne un faux positif tous les 32768 essais, alors que la signature est sur 16 bits et donc permet le taux nominal de 1/65536 (le double). Le polynôme de parité a divisé par deux la capacité à détecter des erreurs purement aléatoires.

Mais cette considération n'est pas importante du tout, puisque nous ne sommes pas dans le cas où la signature est physiquement détachée du bloc concerné et protégée des altérations. On pourrait tenter de la protéger avec un autre code redondant comme celui de Hamming (il permet de détecter des erreurs de 2 bits et de corriger des erreurs d'un bit) ou son extension : les codes Reed-Solomon. Cela n'aurait pourtant aucun intérêt, puisque les bits redondants ainsi utilisés pourraient servir à la signature elle-même, sans augmenter la complexité du programme.

En résumé : pour notre usage, **le polynôme de parité est inutile.** D'ailleurs, il a été enlevé dans les générations suivantes de CRC (par exemple, le polynôme standard de CRC32 est primitif). Nous avons juste besoin d'un *polynôme primitif clairsemé*, c'est-à-dire avec le moins de termes possible pour que nos futurs programmes soient très courts, donc très rapides. N'importe quel polynôme primitif conviendrait, mais il y a un problème : il n'y a pas de trinôme primitif dans GF(2^16).

- Le premier indice vient du trinôme *x^n+x+1* qui est primitif pour n=15 mais pas pour n=16. On en déduit (grâce à la propriété de symétrie) que le trinôme *x^16+x^15+1* n'est pas non plus primitif.
- Les autres trinômes de la forme *x^16+x^i+1* (avec *i* variant de 2 à 8) ne sont pas non plus primitifs. J'ai testé leur période avec le petit programme suivant, écrit sur le pouce :

```
/*
  fichier gene16.c
  fonction : cherche un trinôme générateur pour GF(2^16)
*/

#include <stdio.h>
#include <string.h>

 /* en raison de la faible longueur du cycle (64K max),
   on peut utiliser directement un tableau au lieu
   d'une méthode plus complexe : */

unsigned char compteurs[65536];

int main(void) {
  unsigned long int i, iteration;
  unsigned short int lfsr, mask;

  for (i=1; i<16; i++) {
   /* initialise les variables pour un tour */
    memset(&compteurs, 0, sizeof(compteurs));
    iteration=0;
    mask = (1 << i) | 1;
   lfsr=1; /* seed */

    while (!compteurs[lfsr]) {
      compteurs[lfsr] = 1;
      iteration++;


         /* calcul d'un pas de LFSR */
      lfsr = (lfsr+lfsr)
           ^ (mask & (((signed short int)lfsr) >> 15));
    }

    printf("trinôme x^16 + x^%d + 1 : %d itérations\n",
      i, iteration);
  }

  return 0;
}
```

La sortie du programme ne contient pas le nombre 65535, bien qu'il y en ait un qui s'en approche. Remarquez que j'ai calculé tous les trinômes pour bien montrer la symétrie des polynômes primitifs.

```
trinôme x^16 + x^1 + 1 : 255 itérations
trinôme x^16 + x^2 + 1 : 126 itérations
trinôme x^16 + x^3 + 1 : 57337 itérations
trinôme x^16 + x^4 + 1 : 60 itérations
trinôme x^16 + x^5 + 1 : 16383 itérations
trinôme x^16 + x^6 + 1 : 434 itérations
trinôme x^16 + x^7 + 1 : 63457 itérations
trinôme x^16 + x^8 + 1 : 24 itérations
trinôme x^16 + x^9 + 1 : 63457 itérations
trinôme x^16 + x^10 + 1 : 434 itérations
trinôme x^16 + x^11 + 1 : 16383 itérations
trinôme x^16 + x^12 + 1 : 60 itérations
trinôme x^16 + x^13 + 1 : 57337 itérations
trinôme x^16 + x^14 + 1 : 126 itérations
trinôme x^16 + x^15 + 1 : 255 itérations
```

- Il n'y a donc pas de trinôme primitif dans GF(2^16). Peut-on alors chercher un polynôme primitif à quatre termes ? Inutile car un polynôme primitif a nécessairement un nombre impair de termes (cf. Schneier, page 400).
- En partant du programme précédent, on peut trouver tous les polynômes primitifs dans GF(2^16) en moins d'une minute (chez moi, 12 secondes à 500MHz, et sans rien optimiser), mais le résultat confirme la théorie : parmi les 2048 candidats trouvés (en comptant les doublons symétriques),

il n'y a aucun polynôme primitif dans `GF(2^16)` avec moins de 5 termes.

Donc, si nous restons dans `GF(2^16)`, **nous ne pouvons compter que sur notre cher vieux polynôme `0x8005` (à quatre termes)**. Son cousin direct, CRC16-CCITT (4 termes aussi), est utilisé dans l'interface ATA/IDE (le câble reliant les disques durs à la carte mère) puisqu'il minimise le nombre de portes logiques nécessaires à calculer la signature des blocs (ils sont longs de 512 octets). Comme quoi, nous en revenons toujours aux mêmes recettes.

L'intérêt du polynôme `0x1021` en électronique est que les termes sont espacés, ce qui permet de mieux répartir les portes logiques et d'atteindre des fréquences plus élevées. Mais en logiciel, `0x8005` est préférable, car nous cherchons à regrouper les termes pour réduire le nombre de variables intermédiaires. C'est critique dans une boucle déroulée, et pour notre cher polynôme *(16,15,2,0)*, le plus grand écart est de seulement un terme, entre le terme *2* et le terme *0* (note : pour arriver à ce résultat, le polynôme subit une rotation pour que tous les termes soient groupés).

Conclusion

Je pense que tous les aspects théoriques des CRC sont maintenant couverts. Concevoir un dérivé des CRC est une bonne manière de mettre en perspective tout ce que nous apprennent les livres à leur sujet. Le prochain article va recoller tous les morceaux exposés depuis décembre et passer directement au codage pratique.

Yann Guidon,

LIENS

- Miroir des sources : **http://ygdes.com**
- L'excellent, l'incontournable, le compréhensible, l'anglophone texte de Ross N. Williams, « *A painless guide to CRC error detection algorithms* » : **http://www.ross.net/crc/crcpaper.html** (à lire et à relire jusqu'à ce que cela semble évident).
- Les CRC expliqués par Terry Ritter dans Dr Dobb's : **http://www.ciphersbyritter.com/** Et sur comp.arch : **http://www.ciphersbyritter.com/crchash.htm** (voir en particulier le quiproquo sur la configuration d'un registre de CRC).
- « *A two-step computation of cyclic redundancy code CRC-32 for ATM networks* », par R. J. Glaise, IBM *Journal of Research and Development*, 1997, **http://www.research.ibm.com/journal/rd41-6.html,** décrit une méthode pour découper un CRC en deux parties, en jouant sur des multiples du polynôme ayant moins de termes.

Et dans votre noyau favori : `linux/include/net/checksum.h` et `linux/include/asm/checksum.h` (fonction `ip_fast_csum()`) qui calculent le *checksum* d'un paquet IP sur 32 bits, selon le principe donné dans la RFC791 (qui ne fait pas intervenir de corps de Galois).

ANNEXE

A la demande de relecteurs, voici un programme plus général et un peu mieux conçu permettant de trouver tous les polynômes générateurs dans `GF(2^n)` avec `n` entre 4 et 31. Les techniques suivantes sont employées :

- *La mémoire n'est plus utilisée pour détecter le rebouclage (un LFSR avec un polynôme non primitif reboucle sans passer par une étape intermédiaire non cyclique).*
- *Par contre, on utilise un « crible » binaire en mémoire pour sauter les tests que l'on sait inutiles, que le polynôme ait été vérifié comme primitif ou non (ce qui donne le même résultat final).*
- *Seuls les polynômes impairs sont testés (le LSB doit être à 1).*
- *Les polynômes à nombre pair de termes sont évités (par définition, ils ne sont par primitifs)*

En réduisant la quantité de nombres à tester (et surtout en évitant d'exécuter `memset()` à chaque fois), ce programme est beaucoup plus rapide que l'adaptation du précédent. Il reste néanmoins très rudimentaire, car des mathématiques plus puissantes permettent de ne pas avoir à faire tourner de registre.

Tant qu'on reste avec des polynômes de moins de 30 bits, cela n'est pas très important pour nous car le temps requis pour comprendre et coder les algorithmes complexes est supérieur au temps d'exécution de notre programme. En effet, la plupart des polynômes non primitifs ont des cycles courts et les polynômes primitifs ne sont pas majoritaires.

Cependant, le temps nécessaire à exécuter un milliard d'itérations d'un même LFSR est supérieur aux tests complexes et non exhaustifs de primitivité. On peut aussi noter que c'est un programme facilement parallélisable, puisque chaque nombre peut être vérifié indépendamment des autres.

```
/*
  fichier : primitifs.c
  fonction :
     cherche tous les polynômes générateurs pour GF(2^N)
     avec N < 32 et par test exhaustif (lent) mais assisté
     par quelques éliminations simples.
  compilation :
    gcc -DN=x -Os -march=pentium3 -fomit-frame-pointer -o primitifs primitifs.c
  Vitesse indicative :
    3,95s pour N=16 sur P3@500MHz
    mais le runtime croît un peu plus vite que 2^N
  Occupation mémoire :
    2^(N-4) octets, par exemple : 4K octets pour N=16,
    64K pour N=20, 128M pour 31. 32 et plus nécessitent un recodage.
*/

#include <stdio.h>
#include <string.h>

#ifndef N
#define N 16 /* le degré désiré, de 4 à 31 */
#endif

#define MaskN  (1UL<<N)
#define MaskN1 (MaskN-1UL)
#define SEED 1

char exclusion[1<<(N-4)];
 /* il faut 2^(N-3) octets mais on élimine en plus
  tous les nombres pairs. On pourrait encore éliminer
  les index avec un nombre pair de bits mais la mémoire n'est
  plus trop chère. */

char binaire[N+1]; /* attention, ici ya un truc :-) */

int affiche_binaire(unsigned int n) {
  int i, j=0;

  for (i=N; i>=0; i--) {
    if (n & 1) {
      j++;
      binaire[i] = '*';
    }
    else
      binaire[i] = ' ';
    n>>=1;
  }

  printf(binaire);
  return j;
}

/* initialisés au début du programme */
unsigned char table_bitrev[256],
              table_parite[256];

int main(void) {
  unsigned int i, j, rev,
      iteration, lfsr,
      nb_primitifs=0, tests=0;

  /* l'initialisation pourrait être optimisée mais n'est pas critique */
  for (i=0; i<256; i++) {
  /* initialisation de la table de bitrev */
    table_bitrev[i] = ((i & 1) <<7)
                    | ((i & 2) <<5)
                    | ((i & 4) <<3)
                    | ((i & 8) <<1)
                    | ((i & 16) >>1)
                    | ((i & 32) >>3)
                    | ((i & 64) >>5)
                    | ((i & 128) >>7);

  /* initialisation de la table de parité */
    table_parite[i] =  (i & 1)
                    ^ ((i & 2)  >>1)
                    ^ ((i & 4)  >>2)
                    ^ ((i & 8)  >>3)
                    ^ ((i & 16) >>4)
                    ^ ((i & 32) >>5)
                    ^ ((i & 64) >>6)
                    ^ ((i & 128)>>7);
  }

  /* Ne s'intéresser qu'aux "nombres" impairs,
     mais >1 car il n'existe pas de binôme primitif */
  for (i=3; i<MaskN; i+=2) {

    /* élimination des polynômes à nombre pair de termes */
    j=table_parite[ i        & 255]
#if N > 7
    ^(table_parite[(i >> 8 ) & 255])
#if N > 25
    ^(table_parite[(i >> 16) & 255])
#if N > 23
    ^(table_parite[(i >> 24) & 255])
#endif
#endif
#endif
 ;
    if (!j) {
/* recherche du polynôme réfléchi et autres éliminés */
      if (! ((exclusion[i>>4] >> ((i>>1)&7)) & 1)) {
        tests++;

        /* initialise les variables pour les tests */
        iteration=0;
        lfsr=SEED;
        do {
          iteration++;
          /* calcul d'un pas de LFSR (à améliorer !) */
          j=((signed int)(lfsr<<(32-N))) >> 31;
          lfsr = (lfsr+lfsr) ^ (i & j);
          lfsr &= MaskN1; /* enlève les MSB qui dépassent */
        } while (lfsr != SEED);

        /* inversion de l'ordre des bits */
        j=i|MaskN;
        j= (table_bitrev[ j        & 255] << 24)
#if N > 7
         | (table_bitrev[(j >> 8)  & 255] << 16)
#if N > 15
         | (table_bitrev[(j >> 16) & 255] << 8)
#if N > 23
         |  table_bitrev[(j >> 24) & 255]
#endif
#endif
#endif
        ;
        rev= ( ((unsigned) j) >> (31-N) ) & MaskN1;

        /* écriture du flag empêchant le test
           de la réflexion du polynôme courant : */
        exclusion[rev>>4] |= 1<< ((rev>>1) & 7);

        if (iteration==MaskN1) {
          nb_primitifs++;
          printf("mask=0x%08X (", i );
          printf("-%d termes) ",affiche_binaire(i|MaskN));
          printf("bitrev: 0x%08X(",rev);
          affiche_binaire(rev|MaskN);
          printf(")\n");
        }
      }
    }
  }
  printf("%d primitifs (sans les réfléchis), %d tests\n",
    nb_primitifs, tests);

  return 0;
}
```