Réalisation du CRC pour le conteneur MDS

Yann Guidon

EN DEUX MOTS **Avant d'étudier la compression des données, il faut d'abord se mettre d'accord sur un format de fichier-conteneur adapté. Celui-ci doit assurer l'intégrité des données, au moyen d'un algorithme de CRC que nous allons préalablement mettre au point. La détection des erreurs dans un flux de données est un sujet assez important, compliqué et général pour justifier un article séparé de celui sur la conception du conteneur MDS (à venir). Nous allons examiner les critères et justifier les différents choix, puis programmer et optimiser, afin d'obtenir des routines aussi efficaces que (raisonnablement) possible.**

Précisions sur MDS

Avant de commencer, quelques précisions contextuelles s'imposent. La mise au point du format MDS (c'est plus court que « *Multiplexed Data Stream* »), qui contiendra nos futurs flux de signaux compressés dans la suite de cette série, implique une détection des erreurs de flux. Cela peut être détecté par un bloc « non valide » (qui semble être incohérent ou mal formé), mais cela ne suffit pas puisqu'il est souvent déjà trop tard : la fin du bloc précédent a probablement été altéré mais a quand même été utilisé.

La raison majeure justifiant cette précaution est que les *codecs* utilisant ce conteneur seront généralement « conçus pour la vitesse » et n'effectueront qu'un minimum de vérifications sur la validité des données en entrée. Une donnée hors bornes (accidentellement ou non) peut conduire à des plantages ou des failles de sécurité dans le pire des cas.

Dans un flux MDS, l'utilisation de petits blocs de données permet de réduire (dans une certaine mesure) la perte d'information en cas de corruption locale, qui est la plus probable. S'il ne s'agit que de la modification d'un ou plusieurs bits, des outils particuliers permettront éventuellement de deviner ou retrouver les données initiales en corrélant les informations sur plusieurs couches de protocole (bien que la méthode la plus courante soit simplement d'ignorer le bloc entier).

L'objectif final étant de traiter des signaux en temps réel (et même plus vite si possible), la contrainte principale est d'utiliser un minimum de ressources tout en restant parfaitement portable sur les plateformes actuelles. Cela nécessite une grande parcimonie et une analyse détaillée de tous les mécanismes impliqués.

Un flux MDS est composé d'une succession de paires en-tête/données, comme des paquets dans un réseau de communication. Le CRC doit protéger aussi bien l'en-tête que la donnée, mais ils doivent être indépendants afin d'éviter la propagation d'une éventuelle erreur :

- Si la zone de données est corrompue, cela ne veut pas dire que l'en-tête est invalide, ce qui empêcherait la resynchronisation du programme avec le flux.
- Si l'en-tête est invalide, et seulement cela, il ne faut pas que la zone de données soit invalidée du même coup, ce qui empêcherait en pratique une récupération de données avec des outils spécialisés.

Donc, le même algorithme de CRC sera utilisé de deux manières différentes : d'une part, lorsqu'on examinera la validité d'un en-tête, d'autre part pour valider un bloc entier de plusieurs kilooctets. Une première version sous forme de `#define` (noyée facilement au milieu d'une fonction) est nécessaire pour le premier cas, alors qu'une version en boucle déroulée s'impose pour traiter un bloc plus long.

CRC contre « checksum »

Pour détecter une corruption (aucune modification ne pouvant être empêchée), la méthode générale consiste à combiner tous les octets dans une « signature » qui ne serait plus cohérente avec les données si un ou plusieurs bits (de la signature et/ou des données) étaient altérés.

Prenons par exemple un PC. Son BIOS effectue au démarrage un autotest de son propre programme d'une manière très simple : tous les octets sont additionnés entre eux (sans retenue), ce qui fournit un moyen facile, rapide et raisonnable de s'assurer que tout va bien.

Cependant, l'addition (appelée « *CheckSum* » ou « somme de contrôle » pour les francophonistes) n'est pas très fiable et est très facile à contourner manuellement. Certaines erreurs « simples » peuvent passer inaperçues : il suffit par exemple d'inverser la valeur de deux bits de poids identiques sur deux octets différents.

```
0xCD 0xEF : checksum = 0xBC
XOR 0x02 0x02
=> 0xCF 0xED : checksum = 0xBC (altération non détectée)
```

Cela ne se produit vraisemblablement pas dans une ROM de BIOS, mais, dans l'absolu, ce n'est pas impossible sur un fichier transmis par une mauvaise liaison série, par exemple.

Pour s'assurer avec certitude qu'aucune altération ne s'est produite, il existe des algorithmes de « qualité cryptographique » tels que SHA ou MD5 (quoique des attaques apparaissent).

Ceux-ci sont cependant gourmands et lents par rapport à une simple addition, donc hors de propos pour la majorité des applications. Et nous n'avons même pas besoin de sécurité cryptographique, nous désirons juste détecter les erreurs « possibles en pratique » : erreurs d'un ou quelques bits contigus ou alors bloc tronqué.

Une solution intermédiaire existe : les « *Cyclic Redundancy Checks* » ou « CRC » nécessitent une poignée d'opérations basiques par octet, ainsi qu'une petite table. Une altération aléatoire n'a quasiment aucune chance de passer inaperçue si l'algorithme est correctement réalisé et si les paramètres sont bien choisis. En réalité, la plupart des algorithmes utilisent les mêmes paramètres car très peu sont optimaux. Mais une fois ces paramètres choisis, le gros du travail commence.

Principes de base

Au lieu d'additionner tous les octets entre eux (sans tenir compte de la retenue, donc *modulo 256*), l'idée de base du CRC est de réaliser une division sur un « grand nombre » (qui est notre bloc de données à protéger ou vérifier). Le diviseur est appelé « polynôme générateur » (ou simplement « polynôme » pour aller plus vite) et la signature (le « CRC ») est le « reste » de la grande division. Pour vérifier que le CRC est correct, nul besoin d'effectuer l'opération inverse (donc pas de multiplication), il suffit de recommencer la division et de vérifier que le reste n'est pas différent.

L'astuce centrale est que la division n'est pas réalisée de manière classique, mais avec une arithmétique spéciale. La lente instruction de division habituelle est alors inutile, remplacée par des XOR et une petite table.

Le « polynôme » doit être choisi afin de laisser passer un minimum d'erreurs et seuls quelques-uns conviennent à cause de leurs propriétés mathématiques. En pratique, il en existe un ou deux par famille de CRC, ils sont donc réutilisés d'une implémentation à l'autre pour éviter de réinventer la roue.

Pour mieux comprendre comment tout cela fonctionne, je vous recommande de lire le texte (devenu une référence) écrit par Ross N. Williams, qu'il a versé dans le domaine public. Il est largement disponible sur Internet. Il a été utilisé précédemment lors de la conception de l'utilitaire gdups dont la réalisation est décrite dans GLMF n°61 de mai 2004.

Paramétrage

La grande famille des algorithmes de CRC nous offre plusieurs paramètres sur lesquels nous pouvons jouer pour obtenir le meilleur compromis entre performances et ressources (mémoire et processeur).

Le conteneur MDS peut être rapproché au conteneur Ogg (de **Xiph.org**) mais diffère par ses applications et donc ses choix techniques. Le CRC est un bon exemple du résultat des divergences fondamentales entre ces deux formats d'encapsulation et pour les illustrer, on va comparer dans cette partie leurs paramètres respectifs.

Pour commencer, la RFC n°3533 décrivant le conteneur Ogg, ainsi que les documents de Xiph.org, précisent qu'il utilise :

- Un CRC sur 32 bits (4 octets) ;
- De polynome `0x04c11db7` ;
- Un algorithme direct
- Avec une valeur initiale et un XOR nuls.

Nous allons voir maintenant en détail de quoi il s'agit (pour plus de renseignements sur les autres aspects de Ogg, des liens sont proposés après la conclusion).

Ces choix fonctionnent très bien pour Ogg mais MDS a une granularité plus fine (blocs de 4 KO maximum au lieu de 64 KO) ainsi que des contraintes de réalisation plus sévères (l'encodage et le décodage doivent être réalisables sur des plateformes trop petites pour Ogg/Vorbis). Le CRC ne peut donc pas être celui de Ogg, qui a été prévu pour être théoriquement inaltérable (il suffit de lire les spécifications pour voir qu'il a été prévu « très large »).

Au contraire, le conteneur MDS ne cherche pas à être « incassable », mais juste à être économe et solide « en pratique », c'est-à-dire dans la limite des altérations qui peuvent raisonnablement se produire dans la réalité : faux contact dans un connecteur, bruit induit par un téléphone portable...

- **Taille :**
  - D'abord, une table de CRC 32 bits nécessite 1024 octets. C'est relativement peu de nos jours, mais cela empiète tout de même sur les autres données à traiter, en particulier dans les processeurs dotés de peu de mémoire cache. Lorsque des traitements lourds et compliqués seront nécessaires, chaque octet de mémoire cache L1 comptera.
  - Ensuite, par rapport à Ogg, l'en-tête d'un bloc MDS est de petite taille, il y a donc proportionnellement moins de probabilité qu'une erreur purement aléatoire s'y produise, et dans cette éventualité, il y a beaucoup moins de chances qu'elle passe inaperçue.
  - De même, pour la zone de données qui est plus petite : une erreur « simple » (de quelques bits) est facilement détectée (même avec 16 bits) et une erreur « complexe » (échappant à la détection d'erreur) ne donne pas de données cohérentes pour la plupart des codecs.

Un CRC sur 16 bits convient donc pour MDS :

- Cela laisse une probabilité « théorique » d'erreur non détectée de 2^{-16} (une pour 65536) au lieu de 2^{-32}, en utilisant le modèle d'erreurs le plus défavorable. C'est largement suffisant alors que la taille de la table est divisée par deux.

- Un CRC16 peut théoriquement protéger au maximum 65536 bits, soit 8K octets, et la taille maximale des blocs de données dans MDS est de 4 K octets.
- Enfin, par rapport à CRC32, le CRC16 permet l'économie de 4 octets par bloc : deux dans l'en-tête et deux dans la zone de données.

▶Polynôme :
La taille étant différente, le polynôme doit être changé. Ici :

```
#define CRC16_poly  0x8005
```

C'est la valeur la plus courante, utilisée par ARC par exemple. Le polynôme pourrait être inversé sans changer son efficacité, mais cela n'a aucun intérêt.

▶XOR :
- En théorie, le XOR (ou complémentation) final sert à conserver la définition mathématique du CRC.
- En pratique, il semble n'avoir aucun effet : si le XOR final n'est pas appliqué lors du codage **et** du décodage, l'identité (ou l'altération) sera détectée. Ce XOR inutile pourrait donc être supprimé (comme c'est le cas pour Ogg).

Pour ce qui est de la réalité, les mesures dans la suite de cet article valent plus que des explications. Nous allons considérer pour l'instant que ce paramètre est définissable à la compilation pour faciliter les tests d'efficacité :

```
#ifdef CRC16_NEGATE
  return t ^ 0xFFFF;
#else
  return t;
#endif
```

▶Valeur initiale :
Ross Williams explique très bien que la valeur initiale du CRC a une grande importance sur la détection des erreurs dans le cas où elles se produisent au début d'un flux de zéros. Il est peu probable que cela se produise en pratique (en particulier sur un flux compressé, dont l'entropie est donc proche du maximum), mais on ne peut pas prétendre que cela soit impossible.

De plus, cela ne « coûte » rien d'initialiser le CRC à une valeur non nulle, puisqu'il faut bien donner une valeur de départ. Ainsi :

```
#define CRC16_seed  0xFFFF
```

En fait, toute valeur non nulle conviendrait, mais la valeur ci-dessus est une valeur standard.

▶Direction :
Il n'y a pas à chercher loin : l'algorithme nécessitant le moins d'opérations et de manipulations est à préférer. La version « réfléchie » est choisie car elle nécessite un seul décalage, contre deux pour la version « normale » (Voir les définitions dans le texte de Ross Williams).

Portabilité

Pour partir du bon pied et rendre la suite lisible, définissons quelques symboles qui vont éviter au code d'exploser lors de la compilation sur une autre machine. Ces symboles sont utilisés lorsque la taille joue un rôle particulier, les compteurs de boucles sont laissés en `int`.

```
#include <endian.h>
#include <sys/types.h>
#define U8 __U8_TYPE
#define U16 __U16_TYPE
#define U32 __U32_TYPE
#define U64 __U64_TYPE
#define PTR_CAST (long) /* supprime un warning
   lorsqu'on teste l'alignement d'un pointeur */
```

Du moins c'est ce que j'ai fait au début. Mais les fichiers `sys/types.h` ne contiennent pas la même chose sur tous les ordinateurs et selon les systèmes, en particulier les systèmes embarqués ou avec un compilateur incompatible. La version actuelle est donc codée ainsi :

```
#include <endian.h>
#define U8  unsigned char
#define U16 unsigned short int
#ifndef U32
 #if defined (__alpha) /* || defined(__amd64__) || defined (__x86_64__) */
  #define U32 unsigned int
 #else
  #define U32 unsigned long int
 #endif
#endif
#define U64 unsigned long long int
#define PTR_CAST (long)
```

La portabilité est difficile, quel que soit le moyen. S'il y en a un, je ne le connais pas et il est donc trop compliqué. Soit l'utilisateur doit vérifier que son fichier `sys/types.h` convient, soit il doit modifier le fichier source pour régler la taille des variables si elles ne conviennent pas. Dans la plupart des cas, les définitions fonctionnent (si on se base sur une écrasante majorité d'ordinateurs x86 32 bits), mais il est stupide qu'une chose aussi simple que la taille d'une variable soit si difficile à obtenir au moment de la compilation (et par le préprocesseur).

La table constante

Malgré l'apparence arbitraire, la table de CRC (on dit habituellement « LUT » qui vient du terme anglais *LookUp Table*) n'est pas difficile à fabriquer ; il faut juste faire très très attention. Cela consiste à effectuer les opérations de « division » bit à bit, pour réutiliser les résultats sur des octets par la suite. C'est la même démarche que de précalculer une table de sinus ou de multiplication.

ATTENTION

Le détail critique est que nous utilisons l'algorithme de consultation réfléchi mais avec un polynôme normal et des données normales. En conséquence, c'est l'ordre des octets de la LUT qui est inversé, non celui des bits. Il ne faut pas confondre cette table avec une table pour CRC réfléchi ou inversé.

L'algorithme « réfléchi » est conçu pour travailler sur des octets dont l'ordre des bits est inversé. Or, nos octets de données sont normaux. Pour comparer le présent algorithme avec un CRC réfléchi, il faut soit inverser l'ordre des bits des données, soit celui des bits de la LUT. Il y a de quoi se perdre, ce qui entretient le flou quantique autour des CRC en général.

En fait, j'ai trouvé tellement de variations du code sur Internet que je ne savais plus trop à quelle référence me fier. Les deux tiers des codes existants sont peut-être faux, ce qui ne les empêche pas de fonctionner convenablement puisqu'une table aléatoire est presque aussi efficace qu'une table idéale.

Pour valider la LUT, j'ai donc reconstruit l'algorithme de CRC16 « par division » (qui opère bit à bit) en me servant du texte de Ross Williams, puis déduit l'algorithme de génération de la table, et enfin comparé le fonctionnement de tous les codes qui utilisent la LUT (c'est là qu'on trouve les premiers bugs).

La seule différence entre l'algorithme de référence et ceux utilisés ci-après est que l'ordre des octets est inversé. Par commodité, cette inversion n'est pas opérée par la suite et le code de référence est donc modifié légèrement.

```
/* l'algorithme de référence */
U16 CRC16_reference(U8 *p, unsigned int count){
  unsigned int i, j;
  U32 ref = CRC16_seed;
  for (j=0; j<count; j++) {
    ref ^= (*(p++)) << 8 ; /* insertion */
    for (i=0; i<8; i++) {
      ref<<=1;
      if (ref & 0x10000)
        ref^=CRC16_poly;
    }
  }
  /* byteswap terminal */
#ifdef CRC16_NEGATE
  return ((ref >> 8) & 255) | ((ref & 255) << 8) ^ 0xFFFF;
#else
  return ((ref >> 8) & 255) | ((ref & 255) << 8);
#endif
}
```

Une fois compilée, la fonction de génération de la table prend moins de place que la table elle-même, c'est pourquoi je privilégie (pour économiser quelques ressources) la version « codée » à la place de la grosse table « constante ».

Mais durant la conception des algorithmes l'utilisant, un des bugs les plus frappants que j'ai commis est d'oublier d'appeler la routine de construction de la LUT. Comme le compilateur initialise toutes les variables à zéro, l'algorithme ne détectait aucune des erreurs que j'introduisais pour vérifier le fonctionnement.

Dans le fichier programmant le CRC de MDS, j'ai donc défini par défaut l'inclusion de la grande table constante, pour éviter que d'autres étourdis ne commettent la même erreur. La table a été obtenue directement par le code de génération pour garantir l'exactitude du résultat.

Le code est une adaptation de celui publié dans ces pages pour gdups (GLMF n°61, mai 2004), avec quelques corrections et une petite « protection anti-étourdis ».

```
/*
   Poly = 16,15,2,0
   CRC16 standard: (1)-1000-0000-0000-0101 = 0x8005
   init=0xFFFF
 */
#define CRC16_poly  (0x8005)
#define CRC16_seed  (0xFFFF)
#ifndef CRC16_SMALL_FOOTPRINT
TYPE_CRC16 CRC16_LUT[256]= {
  0x0000, 0x0580, 0x0F80, 0x0A00, 0x1B80, 0x1E00, 0x1400, 0x1180,
  0x3380, 0x3600, 0x3C00, 0x3980, 0x2800, 0x2D80, 0x2780, 0x2200,
   >8  >8  >8  >8  >8  snip plein de nombres >8  >8  >8  >8  >8
  0x2002, 0x2582, 0x2F82, 0x2A02, 0x3B82, 0x3E02, 0x3402, 0x3182,
  0x1382, 0x1602, 0x1C02, 0x1982, 0x0802, 0x0D82, 0x0782, 0x0202
};
#else
TYPE_CRC16 CRC16_LUT[256];
 /* Ne pas oublier d'appeler cette routine au début du programme ! */
void create_CRC16_LUT() {
  int i, j;
  U16 r;
  for (i=0; i<256; i++) {
    r=i << 8;
    for (j=0; j<8; j++) {
      if (r & 0x8000)
        r = (r << 1) ^ CRC16_poly;
      else
        r <<= 1;
    }
    CRC16_LUT[i] = ((r<<8) |(r>>8)) & 65535 ;
  }
}
#endif
```

On peut noter que par construction, la première case du tableau est à zéro, ce qui justifie l'initialisation du registre de CRC à une valeur non nulle.

Une mesure plus draconienne serait de mettre une valeur arbitraire (mais choisie pour être différente de toutes les autres) dans la case nulle de la LUT.

Cela compliquerait l'étude mathématique de l'algorithme sans beaucoup réduire son efficacité. Pour l'instant, cela n'est pas justifié mais cela mérite réflexion.

La routine de CRC

Le cœur du système de CRC est le calcul de la signature des blocs de données. Dans son texte, Ross Williams définit ainsi l'algorithme réfléchi :

```
unsigned long crc_reflected (blk_adr,blk_len)
unsigned char *blk_adr;
unsigned long  blk_len;
{
  unsigned long crc = INIT_REFLECTED;
  while (blk_len--)
    crc = crctable[(crc ^ *blk_adr++) & 0xFFL] ^ (crc >> 8));
  return crc ^ XOROT;
}
```

Cela correspond à la figure n°1.

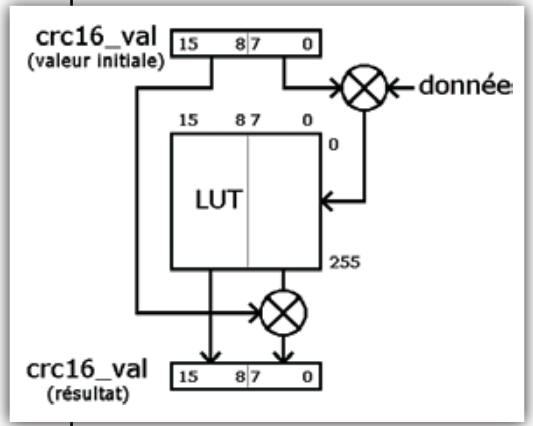

Fig. 1 : Structure d'une itération de l'algorithme de CRC16 réfléchi.

Appliqué à notre usage, cela s'écrit ainsi :

```
TYPE_CRC16 crc16_val;

#define CRC16_step(msg_in) \
   crc16_val = CRC16_LUT[(crc16_val & 255) ^ msg_in] ^ (crc16_val >> 8);
/* Attention : msg_in doit être un octet non signé */
```

La macro permet d'inclure le CRC au milieu du code de vérification d'intégrité des en-têtes. Ce dernier doit effectuer lui-même, le cas échéant, la complémentation des bits avant de les écrire ou de les comparer.

Cette manière d'écrire n'est pas « sûre » du tout, en particulier à cause de l'utilisation directe de `msg_in`, il faut donc faire très attention aux pointeurs postincrémentés par exemple. D'un autre côté, gcc n'aime pas les fonctions `inline` en l'absence d'option -O, ce qui ne facilite pas le développement préliminaire.

Mais surtout, utiliser des macros au lieu de fonctions `inline` force à « rester éveillé » tout en évitant de s'interroger sur la performance du code généré par le compilateur. Enfin, c'est ce que je pensais avant de mettre le nez dans ses petites affaires.

Optimisation

Les efforts d'optimisation ont une grande importance pour cet algorithme, car il consomme relativement beaucoup de temps CPU alors qu'il n'effectue aucun « traitement utile ». Chaque gain, même petit, a donc un effet important pour un algorithme si court, et permet de passer d'autant plus vite au traitement proprement dit.

Dépendances

Première mauvaise nouvelle : toutes les opérations dépendent de l'opération qui la précède. Aucun parallélisme n'est permis car, contrairement à l'opération d'addition, une étape de CRC n'est pas commutative ou associative (c'est aussi pour cela que le CRC est plus « sensible » que l'addition donc plus apte à détecter les erreurs).

Parallélisme

Pour accélérer la vérification d'un bloc de données sur un processeur « moderne » (lire : « complexe », avec réordonnancement des instructions), on pourrait exécuter 2 ou 4 CRC simultanés et indépendants pour « briser les dépendances » entre deux instructions consécutives d'un même CRC. Le résultat des différents CRC serait combiné par addition par exemple, pour donner les 16 bits attendus. L'amélioration sur un processeur comme le Pentium3 serait importante mais cela défavoriserait les processeurs les plus simples. Et comme nous venons de le voir, l'ordre des opérations est important et on ne pourrait pas obtenir le même résultat qu'avec un CRC simple.

Plusieurs octets à la fois

Autre solution pour doubler la vitesse de traitement : utiliser une LUT permettant de vérifier 2 octets à la fois. Cela est théoriquement très simple mais le résultat serait catastrophique sur certains processeurs à mémoire réduite.

C'est tout de même une éventualité à « garder au chaud » si jamais un développeur doit implémenter MDS sur un processeur sans cache et avec une faible pénalité d'accès à la mémoire centrale. Si celle-ci est suffisamment grande, on peut imaginer traiter 2 ou 3 octets en un seul cycle, à condition de disposer respectivement de 3×2^{16}=192KO ou de 4×2^{24}=64MO. En prenant comme exemple un de mes ordinateurs, cadencé à 500MHz et disposant de 2MO de cache externe, le temps d'accès à cette dernière est d'environ 10 cycles d'horloge. Dans le même temps, le cœur peut exécuter jusqu'à 40 instructions sur des nombres entiers. On voit donc tout de suite la limite de l'intérêt pratique de cette approche...

Importance de la taille des données

Il ne reste plus qu'à examiner attentivement les opérations qui sont effectuées. Pour le XOR entre le registre `crc16_val` et le nouveau caractère, la taille du résultat est sur un octet, comme l'indique le `& 255`. La taille des éléments de la LUT est 16 bits, et on retourne sur 8 bits pour le XOR final. En tout, la macro devrait utiliser 3 opérations ou trois instructions sur un x86.

Une nouvelle difficulté surgit, spécifique à la plateforme x86 : c'est l'emploi des préfixes de taille. Ceux-ci précèdent des instructions, en mode 32 bits, pour traiter des nombres 16 bits, et vice versa. Ils ont le fâcheux inconvénient de consommer un cycle de décodage chacun, alors que trois instructions peuvent normalement être décodées en même temps sur un Pentium II ou un Pentium III.

Pire : dans ces architectures, les données internes sont toutes sur 32 bits et le traitement d'octets ou de mots de 16 bits augmente le nombre de micro-opérations à exécuter. Le manuel d'optimisation d'Intel (248966-012.pdf pages 2-7) spécifie, au milieu d'une interminable liste de recommandations :

- *Avoid instructions that unnecessarily introduce dependence-related stalls: inc and dec instructions, partial register operations (8/16-bit operands).*
- *Avoid use of ah, bh, and other higher 8-bits of the 16-bit registers, because accessing them requires a shift operation internally.*

On a donc intérêt à utiliser au maximum un codage sur 32 bits ! Ainsi, pour peaufiner le code, il est possible de choisir la taille du registre de CRC :

```
#ifndef TYPE_CRC16
 #ifdef i386
  #define TYPE_CRC16 U32
 #else
  #define TYPE_CRC16 U16
 #endif
#endif
TYPE_CRC16 crc16_val, CRC16_LUT[256].....
```

On perd dans ce cas l'un des avantages du CRC16 par rapport au CRC32 (la taille plus réduite de la LUT), du moins pour une famille précise de processeurs. Un compromis temps-espace possible consiste à utiliser une table d'éléments sur 16 bits et une instruction `movzwl` qui convertit une donnée (registre ou mémoire) de 16 à 32 bits sans pénalité. Le coût est d'une instruction supplémentaire et d'un registre temporaire supplémentaire.

Optimisations « Peep-hole »

Manuellement, la manière la plus naturelle d'éviter explicitement les masquages est de travailler directement sur chaque partie de `crc16_val` :

```
union {
  struct {
#if !defined(__BYTE_ORDER)
# error Endian indéfini !
#else
# if __BYTE_ORDER == __LITTLE_ENDIAN
    unsigned char lsb, msb;
# else
#  if __BYTE_ORDER == __BIG_ENDIAN
    unsigned char msb, lsb;
#  else
#   error Endian indéfini !
#  endif
# endif
#endif
  } u;
  unsigned short int u16;
} crc16_val, crc16_val2;
/* msg_in doit être de type unsigned char */
#define CRC16_step_pair(msg_in) \
   crc16_val.u.lsb ^= msg_in; \
   crc16_val2.u16 = CRC16_LUT[crc16_val.u.lsb]; \
   crc16_val2.u.lsb ^= crc16_val.u.msb;
#define CRC16_step_impair(msg_in) \
   crc16_val2.u.lsb ^= msg_in; \
   crc16_val.u16 = CRC16_LUT[crc16_val2.u.lsb]; \
   crc16_val.u.lsb ^= crc16_val2.u.msb;
```

Le découpage de l'uniligne originale oblige à utiliser une variable temporaire `crc16_val2`. Pour réduire le nombre d'instructions et de mouvements inutiles, l'opération est effectuée alternativement sur deux codes légèrement différents pour retomber sur la variable du début après un nombre pair d'opérations.

Pourtant on est encore loin du résultat espéré. L'examen de la sortie de GCC indique que celui-ci optimise le masque 0xFF du code original (en travaillant directement sur les octets), mais il a plus de mal quand on l'« aide » en découpant tout. Malgré tous les flags d'optimisation essayés, il lui semble inconcevable d'allouer un registre à des valeurs temporaires que l'on a clairement identifiées, il tient absolument à tout sauver en mémoire.

Paradoxalement, l'optimiseur semble mieux fonctionner avec la version condensée, car il n'y a pas d'accès temporaires à la mémoire, il peut alors mieux gérer seul l'allocation des registres.

Voici un exemple frappant :

```
#define CRC16_step(msg_in) \
   crc16_val = CRC16_LUT[(crc16_val ^ msg_in) & 255]^(crc16_val << 8);
```

Compilé avec l'option -O3, le code correspondant utilise 6 instructions :

```
movb a(%ebx),%al            ; charge le nouveau caractère
xorb crc16_val,%al          ; XOR du caractère et
movzbl %al,%edx             ; crée l'index dans la LUT
movzbw crc16_val+1,%ax      ; effectue le décalage en prenant l'octet haut (+1)
xorw CRC16_LUT(,%edx,2),%ax ; XOR final
movw %ax,crc16_val          ; sauve le résultat
```

Non seulement GCC a optimisé le `&255` et le décalage par des opérations sur des octets, ce qui est une bonne performance, mais il a appliqué d'autres optimisations assez alambiquées. Cependant, bien que le code ne puisse pas raisonnablement être amélioré dans ce contexte précis (en réorganisant ou substituant les instructions

par exemple), il reste des défauts liés entre autres à sa focalisation sur les variables en mémoire au lieu des registres.

Assembleur x86

Traduire manuellement le code « découpé » en code assembleur pour x86 semble évident pour un habitué, surtout lorsqu'on sait que certains registres (les plus anciens) de cette architecture semblent conçus pour cet usage. En particulier, les registres AX, BX, CX et DX sont utilisables en tant que mots sur 16 bits (correspondant à `crc16_val.u16`), ou comme octets individuels :AL, BL, CL et DL sont assimilables à `crc16_val.u.lsb` tandis que AH, BH, CH et DH sont assimilables à `crc16_val.u.msb`. Le codage s'annonce donc très facile, avec un choix inhabituellement large de registres pour placer nos variables. Même si Intel nous dit qu'utiliser AH, BH etc. n'est pas bien. Le x86 est l'exemple ultime d'une architecture complexe et bancale (restons polis). La difficulté qui émerge en tentant d'assembler le code est que l'adressage sur un octet n'est pas permis.

Il faut élargir l'opérande sur 16 ou 32 bits (dans un autre registre pour ne pas écraser le MSB qui servira plus tard) pour obtenir un pointeur utilisable. En se servant uniquement des registres (puisqu'il y a assez de place), le code pourrait s'écrire ainsi (Tab. 1).

On voit que le CRC ne prend que trois instructions si la valeur d'entrée est préalablement étendue sur 32 bits. Pour la lecture d'un nouveau caractère avec post-incrémentation, ajoutez deux autres instructions, et encore deux instructions pour reboucler.

Une fonction complète ?

Après un essai sur une étape de CRC16, on peut envisager d'écrire une fonction contenant une boucle entière. Entre autres astuces, on peut alors « masquer » la deuxième variable par renommage des registres et réordonner quelques instructions pour profiter de la distributivité du XOR (Tab. 2).

La boucle aurait pu être déroulée quatre fois au lieu de deux si seulement il y avait eu quelques registres supplémentaires. En pratique, les dépendances sont tellement tendues entre les dernières instructions du corps de la boucle qu'un déroulage supplémentaire n'aurait que peu d'effet. Il faut aussi compter le code de prologue qui doit aboutir à un compteur de boucle pair. L'approche utilisée dans le corps de la boucle n'oblige pas en plus à aligner le pointeur sur une adresse paire, ce qui aurait encore compliqué le programme. Le prix à payer est deux instructions superflues.

Le code InfoZip

Le code précédent utilise en moyenne onze instructions pour traiter deux octets (soit 5,5 instructions par octet). Bien content de moi, je ne croyais pas qu'il serait possible de faire mieux. Pourtant, une simple recherche sur Internet (`google: addl+crc`) retourne le fichier source `crc_i386.S` du logiciel Info-ZIP. C'est une version très bien conçue et la plus rapide que je connaisse d'un CRC32, avec octuple déroulage et 3,75 instructions en crête par octet. Il aurait été possible d'enlever une des instructions en jouant sur le mode d'adressage mais c'est déjà un fabuleux morceau de code.

J'étais assez fier du réordonnancement d'un XOR, mais il est possible d'aller plus loin : on peut effectuer la lecture des nouveaux caractères dans la même instruction que le XOR, ce qui réduit encore le nombre d'opérations. Le code équivalent en C est le suivant :

```
/* Attention : buf est aligné sur 4 octets
   et la machine est "little endian" */
crc ^= *(U32 *)buf;
((U32 *)buf)++;
crc = (crc >> 8) ^ table[crc & 0xFF]
```

Tableau : 1

Allocation et fonction de chaque registre :
- ▶EAX : registre contenant le CRC précédent (entrée)
- ▶EBX : caractère en entrée puis offset de pointeur
- ▶ECX : CRC courant (sortie)
- ▶ESI : pointeur sur la donnée à vérifier

Label	pseudo-code	notation nasm	notation gas
Initialisation_du_CRC :			
	AX=CRC16_seed, MSB à zéro	`movzwl eax, CRC16_seed`	`movzwl $CRC16_seed,%eax`
Lors_de_la_lecture_de_BX :			
	EBX à zéro sauf BL=[ESI]	`movzbl ebx,[esi]`	`movzbl (%esi),%ebx`
	ESI+=1	`inc esi`	`incl %esi`
Calcul_de_CRC :			
	BL=BL^AL	`xor bl,al`	`xorb %al,%bl`
	ECX=LUT[EBX*2]	`movzwl ecx,[CRC16_LUT+ebx*2]`	`movzwl CRC16_LUT(,%ebx,2),%ecx`
	CL^=AH	`xor cl,ah`	`xorb %ah, %cl`

Tableau : 2

Allocation et fonction de chaque registre :

▶EAX : CRC (sortie)
▶EBX : caractère puis offset de pointeur
▶ECX : CRC temporaire
▶EDX : caractère puis offset de pointeur
▶ESI : pointeur sur le début des données à vérifier (entrée)
▶EDI : nombre d'octets à traiter, strictement > 0 (entrée)

Labels et commentaires	Pseudo-code	Notation nasm	Notation gas
Initialisation_du_CRC :			
	AX=CRC16_seed	`movzwl eax,CRC16_seed`	`movzwl $CRC16_seed,%eax`
;prologue d'alignement : une itération si le LSB du pointeur est à 1			
	LSB(EDI)	`test edi, byte 1`	`test $1,%edi`
	saut si ==0	`jz corps_de_la_boucle`	`jz corps_de_la_boucle`
	EBX à zéro sauf BL=[ESI]	`movzbl ebx,[esi]`	`movzbl (%esi),%ebx`
	ESI++	`inc esi`	`incl %esi`
	BL=BL^AL	`xor bl,al`	`xorb %al,%bl`
	EAX=[LUT+EBX*2]	`movzwl eax,[CRC16_LUT+ebx*2]` `movzwl CRC16_LUT(,%ebx,2),%eax`	
	AL^=CRC16_seed>>8	`not al`	`notb %al`
	EDI--	`dec edi`	`decl %edi`
	saut à la fin si fini	`jz fin_boucle`	`jz fin_boucle` `.p2align 4,,7`
Corps_de_la_boucle :			
;lecture			
	EBX à zéro sauf BL=[ESI]	`movzbl ebx,[esi]`	`movzbl (%esi),%ebx`
	EDX à zéro sauf DL=[ESI+1]	`movzbl edx,[esi+1]`	`movzbl 1(%esi),%edx`
	ESI+=2	`add esi,2`	`addl $2,%esi`
;CRC			
	BL^=AL	`xor bl,al`	`xorb %al,%bl`
	DL^=AH	`xor dl,ah`	`xorb %ah,%dl`
	ECX=[LUT+EBX*2]	`movzwl ecx,[CRC16_LUT+ebx*2]`	`movzwl CRC16_LUT(,%ebx,2),%ecx`
	DL^=CL	`xor dl,cl`	`xorb %cl,%dl`
	EAX=[LUT+EDX*2]	`movzwl eax,[CRC16_LUT+edx*2]`	`movzwl CRC16_LUT(,%edx,2),%eax`
	AL^=CH	`xor al,ch`	`xorb %ch, %al`
; mise à jour des compteurs :			
	EDI-=2	`sub edi, 2`	`subl $2,%edi`
	saut si !=0	`jnz corps_de_la_boucle`	`jnz corps_de_la_boucle`
Fin_boucle :			
; ici, EAX contient le CRC sur 16 bits			

```
crc = (crc >> 8) ^ table[crc & 0xFF]
crc = (crc >> 8) ^ table[crc & 0xFF]
crc = (crc >> 8) ^ table[crc & 0xFF]
```

Le code réalise vraiment la fonction de CRC, bien que l'opération de CRC ne soit pas commutative. Par contre, le XOR qui les compose est une opération distributive par excellence. Comme on le voit sur la figure 2, on peut donc réorganiser les XOR avec les données d'entrée, pour les réunir dans une seule instruction et économiser des opérations individuelles sur un octet (Fig. 2).

C'est ce que j'avais codé pour la boucle déroulée deux fois, sans aller jusqu'à fusionner les deux parties par souci de la lourdeur que le code du prologue d'alignement ajouterait. Le code InfoZip fait la fusion et déroule la boucle quatre fois, au prix d'un prologue et

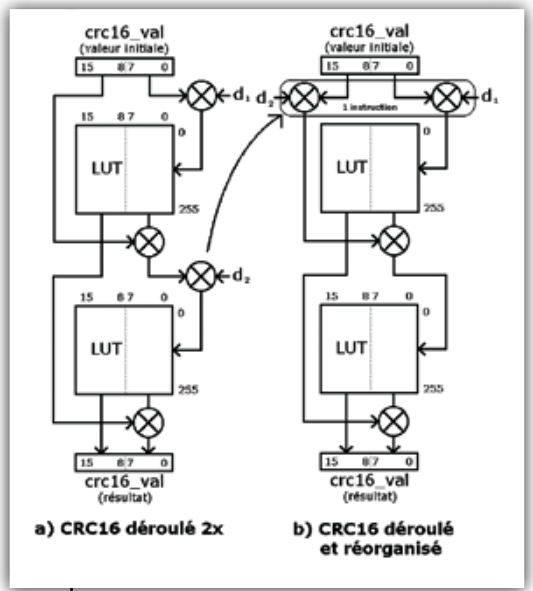

Fig. 2 : Voici la réorganisation des opérations pour 2 étapes de CRC16. La version InfoZip utilise la même méthode pour 4 itérations à la fois !

d'un épilogue plus long que le corps de la boucle. Le cœur de la version en assembleur (ici en syntaxe gas) est encore plus simple que la version octet par octet et utilise un minimum de registres :

```
; Assignation des registres :
;  eax: CRC
;  ebx: registre temporaire
xorl    (%esi), %eax                ; charge et XORe 4 caractères
d'un coup
addl    $4, %esi                    ; incrémente le pointeur de 4
movzbl  %al, %ebx                   ; tmp = c & 0xFF
shrl    $8, %eax                    ; c = (c >> 8)
xorw    CRC16_LUT(, %ebx, 2), %ax   ; c ^=table[tmp]
movzbl  %al, %ebx                   ; même chose, 3 fois
shrl    $8, %eax
xorw    CRC16_LUT(, %ebx, 2), %ax ;
movzbl  %al, %ebx
shrl    $8, %eax
xorw    CRC16_LUT(, %ebx, 2), %ax ;
movzbl  %al, %ebx
shrl    $8, %eax
xorw    CRC16_LUT(, %ebx, 2), %ax ;
```

L'inconvénient de ce type de code très optimisé est la nécessité d'ajouter un long prologue et un long épilogue pour aligner le pointeur sur une frontière de mot 32 bits. Comme les fichiers MDS sont des flux d'octets, on ne peut pas savoir à l'avance à quelle adresse sera le bloc à vérifier, et il faut prévoir en pratique tous les cas de figure.

Enfin du code acceptable !

J'ai beaucoup examiné le code généré par `gcc -S` pour différentes manières d'écrire le source et les combinaisons d'options d'optimisation. Le résultat était tellement décevant (par exemple, la sortie de gcc-2.95.3 était parfois meilleure que gcc-3.3.6) que j'étais prêt à me jeter tête baissée dans l'écriture d'assembleur *inline*, ce qui n'est ni portable, ni lisible, ni la garantie de quoi que ce soit. Au détour de la lecture d'un *hint* du projet LinuxFromScratch, je suis tombé sur la recommandation brûlante d'utiliser l'option -Os, qui optimise pour la taille. Avec l'aide de `-march=XXX` et `-fomit-frame-pointer`, le code est proche de ce que j'aurais pu écrire. Enfin, assez proche pour que je renonce à l'idée du codage direct dans des `__asm__()` ou simplement de jeter gcc. Le code ainsi généré oublie presque entièrement les accès aux variables en mémoire, au profit des registres. Bien sûr, on ne peut pas utiliser gdb (les variables sont lues en mémoire par gdb mais le code ne les met pas à jour), mais je peux m'en sortir en lisant le code avant assemblage. En partant de ce principe, j'ai écrit d'autres macros (pour l'examen dans les en-têtes) et une fonction (pour l'examen de blocs entiers). Le tout est contenu dans le fichier `mds_crc16.c`, qui dispose aussi d'une routine de test sommaire à activer à la compilation par `-DSTANDALONE_TEST_CRC16`.

Si vous avez tout lu jusqu'ici, vous retrouverez tous les ingrédients présentés dans les pages précédentes. La seule complication vient du prologue et de l'épilogue d'alignement : au début, il faut aligner le pointeur, et, à la fin, il faut s'aligner sur le compteur de boucle. Comme l'alignement du pointeur est inconnu dans l'épilogue (le reste étant sauté s'il y a moins de huit octets à traiter), le CRC est alors calculé octet par octet. La remarque précédente sur la taille du code est valable, même en C : l'alignement a un coût (en taille) important, mais il est absolument indispensable pour que la boucle puisse tourner à vitesse maximale (ici j'ai un peu déroulé la boucle interne).

ATTENTION

Le code suivant ne fonctionne pas sur des processeurs tels PowerPC, SPARC et certains MIPS, car il est nativement *Little Endian*. Sur les machines *Big Endian*, il y a plusieurs possibilités (par ordre d'efficacité et de complexité) :

- Utiliser une version « octet par octet », mais elle est plus lente (les résultats varient selon les machines)
- Insérer une instruction de « *byteswap* » (conversion LE/BE) entre le chargement et le XOR : le coût est réduit à une instruction par groupe d'octets. Le codage du byteswap n'est pas très portable en C, ou alors potentiellement complètement inefficace (vérifiez toutefois la sortie de votre compilateur, on n'est jamais à l'abri d'une bonne surprise).

Pour éviter le byteswap, il faut repasser en structure « directe » pour le registre à décalage du CRC. L'opération de masquage des MSB (pour obtenir l'index du tableau) est transformée alors en décalage à droite du registre. Il faut aussi inverser les octets de la LUT ainsi que le CRC final.

Ces dernières adaptations sont assez lourdes et pour l'instant, le fichier `mds_crc16.c` sélectionnera la première option (octet par octet) s'il détecte une machine Big Endian.

```
/* Trois macros qui prennent un entier : */
/* msg_in = unsigned char */
#define CRC16_step(msg_in) \
   crc16_val = CRC16_LUT[(crc16_val & 255) ^ msg_in] ^ (crc16_val >> 8);
/* msg_in = unsigned short int */
#define CRC16_stepX2(msg_in) \
   crc16_val ^= msg_in; \
   crc16_val = CRC16_LUT[crc16_val & 255] ^ (crc16_val >> 8); \
   crc16_val = CRC16_LUT[crc16_val & 255] ^ (crc16_val >> 8);
/* msg_in = unsigned long int */
#define CRC16_stepX4(msg_in) \
   { \
     U32 t = crc16_val ^ msg_in; \
             t = CRC16_LUT[t & 255] ^ (t >> 8); \
             t = CRC16_LUT[t & 255] ^ (t >> 8); \
             t = CRC16_LUT[t & 255] ^ (t >> 8); \
     crc16_val = CRC16_LUT[t & 255] ^ (t >> 8); \
   }
/* Une fonction universelle qui calcule la signature d'un bloc
quelconque : */
U16 CRC16_block(U8 *p, int count) {
  U32 t=CRC16_seed;
/* Prologue d'alignement : */
  if (count >= 8) {
    if ( PTR_CAST p & 1  )  {
      t = CRC16_LUT[ (0xFF & CRC16_seed) ^ (*p) ] ^ (CRC16_seed >> 8);
      p++;
      count--;
    }
    if ( PTR_CAST p & 2 ) {
      t ^= *(U16 *)p;
      t = CRC16_LUT[t & 255] ^ (t >> 8);
      p+=2;
      t = CRC16_LUT[t & 255] ^ (t >> 8);
      count-=2;
    }
    if ( PTR_CAST p & 4 ) {
      t ^= *(U32 *)p;
      t = CRC16_LUT[t & 255] ^ (t >> 8);
      count-=4;
      t = CRC16_LUT[t & 255] ^ (t >> 8);
      p+=4;
      t = CRC16_LUT[t & 255] ^ (t >> 8);
      t = CRC16_LUT[t & 255] ^ (t >> 8);
    }
/* Boucle principale : */
    while (count>=8) {
      t ^= *(U32 *)p;
      t = CRC16_LUT[t & 255] ^ (t >> 8);
      t = CRC16_LUT[t & 255] ^ (t >> 8);
      count -= 8;
      t = CRC16_LUT[t & 255] ^ (t >> 8);
      t = CRC16_LUT[t & 255] ^ (t >> 8);
      t ^= (*(U32 *)(p+4));
      t = CRC16_LUT[t & 255] ^ (t >> 8);
      p+=8;
      t = CRC16_LUT[t & 255] ^ (t >> 8);
      t = CRC16_LUT[t & 255] ^ (t >> 8);
      t = CRC16_LUT[t & 255] ^ (t >> 8);
    }
  }
/* Epilogue: */
  if (count & 4) {
    t = CRC16_LUT[(t & 255) ^ * p    ] ^ (t >> 8);
    t = CRC16_LUT[(t & 255) ^ *(p+1) ] ^ (t >> 8);
    t = CRC16_LUT[(t & 255) ^ *(p+2) ] ^ (t >> 8);
    t = CRC16_LUT[(t & 255) ^ *(p+3) ] ^ (t >> 8);
    p+=4;
    count -= 4;
  }
  if (count & 2) {
    t = CRC16_LUT[(t & 255) ^ * p    ] ^ (t >> 8);
    t = CRC16_LUT[(t & 255) ^ *(p+1) ] ^ (t >> 8);
    p+=2;
    count -= 2;
  }
  /* ici, count==0 ou ==1 par conception */
  if (count & 1)
    t = CRC16_LUT[(t & 255) ^  *p    ] ^ (t >> 8);
#ifdef CRC16_NEGATE
  return t ^ 0xFFFF;
#else
  return t;
#endif
}
#endif /* endian */
```

NOTES

Pour transformer l'algorithme CRC16 en CRC32, il suffit de changer :

- Le polynôme générateur (la valeur du CRC32 classique est `0x04C11DB7`) ;
- La LUT (le code source permet de la régénérer) ;
- La taille des données.

Le résultat de la routine généraliste est exactement le même qu'avec le code octet par octet. Il est encore possible d'améliorer un peu le code pour les machines 64 bits (Alpha et x86-64 en particulier) : les deux chargements 32 bits peuvent être réunis dans un seul chargement 64 bits. C'est ce que réalise la fonction `CRC16_bloc64()`.

Mais sur Alpha ou MIPS (ou les machines RISC en général, ne disposant pas de modes d'adressage complexes), impossible de descendre en dessous de 6 instructions par octet : la consultation de la LUT nécessite trois instructions dont deux pour calculer l'adresse.

A ce niveau de sophistication, il est difficile de trouver des améliorations importantes : la consultation de la table, hors calcul de l'adresse, consomme déjà plusieurs cycles d'horloge pour un processeur actuel, ce qui limite la vitesse maximale de l'algorithme.

Le pied collé au plancher

Chaque instruction dépendant du résultat de l'instruction directement précédente, les processeurs superscalaires ne peuvent les

exécuter en parallèle. Pour que ces derniers puissent fonctionner à plein rendement, il faut recréer le parallélisme d'une autre manière.

Toutes les optimisations connues et simples ont été essayées, sauf une : fusionner deux appels à la routine pour calculer simultanément la signature de deux blocs différents (ou éventuellement plus !).

Il faut tout de suite mettre en garde sur les inconvénients majeurs de cette approche :

- La duplication du code augmente d'autant la taille de la routine de signature.
- L'appel à la routine nécessite d'avoir deux blocs à signer (au lieu d'un), en d'autres termes : cela complique les routines de gestion de flux (qui doivent travailler par lots de plusieurs blocs).
- Les petits processeurs (par exemple pour l'embarqué comme l'ARM) sont exclus : la technique n'est efficace qu'avec les gros processeurs récents (à partir du Pentium Pro pour les x86) qui analysent les dépendances entre quelques dizaines d'instructions pour réorganiser leur exécution (permettant ainsi d'exécuter plusieurs opérations indépendantes à la fois).

La routine `CRC16_block_multi()` présente dans `mds_crc16.c` est donc deux fois plus grosse et pas nécessairement deux fois plus rapide. Comme elle consiste essentiellement en du copier/coller de la fonction optimisée précédente, elle n'est pas compliquée à examiner.

Le seul point particulier concerne le passage des deux valeurs de retour par des variables globales : la fonction n'est plus réentrante, mais cela évite d'utiliser des pointeurs qui auraient empêché l'allocation optimale des registres.

En parlant de ça, justement, les x86 posent un gros problème : on se heurte au nombre limité de registres. La boucle nécessite deux pointeurs vers les données, deux registres de CRC, deux compteurs de boucle : cela occupe déjà six registres alors que l'allocation n'est pas terminée.

Pour les deux pointeurs dans la LUT, il faut ruser et on utilise alors le septième et dernier registre restant, alternativement pour les deux CRC.

On compte sur le processeur (qui dispose de mécanismes internes de renommage des registres) pour deviner les fausses dépendances (et ainsi allouer d'autres registres internes invisibles) puisque la « durée de vie » des valeurs dans ce septième registre est de trois instructions.

```
; dans cet extrait de code assembleur généré par GCC,
; le registre ECX sert d'index partagé pour deux calculs indépendants

movzbl     %al,%ecx   ; Dans une première phase, le CRC est dans EAX
shrl       $8, %eax
xorl       CRC16_LUT(,%ecx,4), %eax

movzbl     %dl,%ecx   ; Dans la deuxième phase, le CRC est dans EDX
shrl       $8, %edx
xorl       CRC16_LUT(,%ecx,4), %edx
```

Évidemment, ce problème est absent pour la plupart des autres ordinateurs car ils disposent de plus de registres. L'évolution du x86 vers le monde 64 bits apporte d'ailleurs un doublement salvateur du nombre de registres.

L'étrange parenté avec les LFSR

Avec un peu de recul, le parallèle est frappant entre les algorithmes de CRC et de LFSR (« *Linear Feedback Shift Register* » soit en français : « Registre à Décalage à Rétroaction Linéaire » ou plus simplement : la forme la plus simple de générateurs de nombres pseudo-aléatoires).

D'abord, il y a la forme elle-même : un registre à décalage sur lequel on applique un XOR. Dans le cas du CRC, ce registre est « perturbé » par le message dont on veut calculer la signature, alors que le LFSR tourne tout seul en rond...

Ensuite, il y a en commun l'emploi d'un « polynôme » particulier, utilisant des mathématiques spéciales. Pour le CRC, c'est un « polynôme générateur », pour le LFSR c'est un « polynôme primitif ». Le nom change, car les fonctions et les propriétés sont différentes, mais on sent bien une parenté. Le soupçon se renforce car l'inversion des bits d'un polynôme donne un autre polynôme valide, pour un CRC comme pour un LFSR...

En fait, ces deux classes d'algorithmes utilisent les mêmes mathématiques (dans le domaine des **corps finis**, plus précisément dans **GF(2)** pour les curieux) et des armées de mathématiciens se sont penchées sur le sujet depuis... longtemps. Nous, humbles informaticiens, pouvons tranquillement nous servir des résultats existants et observer scrupuleusement les précautions indiquées dans cet article ou dans les livres traitant du sujet (les ouvrages de Knuth, Tanenbaum ou Schneier viennent en premier à l'esprit). On peut considérer un algorithme de CRC comme un LFSR « perturbé » par les données à signer. Puisqu'il y a une telle similitude, le codage d'un LFSR peut utiliser les mêmes techniques que celles développées pour le CRC. Partons de l'algorithme de LFSR 32 bits en configuration de Galois présenté par Bruce Schneier dans la deuxième édition française de son ouvrage *Cryptographie Appliquée* :

```
#define MASK 0x80000057

static unsigned long ShiftRegister = 1; /* tout sauf 0, la valeur
interdite */

int modified_RDRL(void)
{
```

```
  if (ShiftRegister & 1) {
    ShiftRegister = ((ShiftRegister ^ mask) >> 1) | 0x80000000;
    return 1;
  } else {
    ShiftRegister >>= 1;
    return 0;
  }
}
```

C'est le type de code simple, stable et portable auquel on s'attend dans un tel livre de référence. On peut l'utiliser sans crainte pour commencer à ébaucher une application. Pourtant, il y a un gros défaut : il y a un branchement.

Pas un branchement anodin mais équiprobable aléatoire, le cauchemar de l'unité de prédiction de branchement de votre microprocesseur... De nos jours, où toutes les instructions sont exécutées dans le désordre, ce détail est fatal pour la performance, c'est un coup à vous mettre le pipeline au chômage technique :-)

La solution est de ne pas utiliser de `if`. Heureusement, ce type de code se transforme très facilement. En passant par une petite table, on peut générer plusieurs bits consécutifs au moyen de quelques instructions, ce qui réduit le nombre d'invocations. Voici le résultat en syntaxe NASM :

```
%assign LFSR_POLY0 080000057h
%assign LFSR_POLY1 ((LFSR_POLY0>>1)|(LFSR_POLY0<>1)|(LFSR_
POLY1<>1)|(LFSR_POLY2<
```

Le code est tellement court (si on peut dédier un registre pour contenir la valeur courante du LFSR) qu'il n'est pas nécessaire de faire une fonction ; une macro ou du copier/coller suffit. On voit aussi qu'il est très facile à adapter pour un nombre particulier de bits. L'exemple fournit 4 bits et il suffit de modifier la table, le masque et le nombre de rotations.

Mais surtout, ce code est très rapide dans les processeurs actuels (à exécution dans le désordre) car la valeur qui nous intéresse est dans EAX, disponible immédiatement. Pendant que le programme va continuer à utiliser cette valeur, l'accès à la table et l'exécution du XOR puis du ROR vont continuer en « tâche de fond » (du moins, tant que EBP n'est pas lu).

Pour la version en C, le seul souci est le manque d'opérations de rotation (ce que je considère personnellement comme le défaut le plus impardonnable de ce langage). Pour rattraper cette situation, lorsqu'on présente la chose bien en évidence à `gcc -Os`, la bonne instruction est déduite du code (ou presque) :

```
lfsr4.c : (code source)
inline U32 lfsr4() {
  U32 t=LFSR_reg & 15;
  U32 u=LFSR_reg ^ LFSR4_TABLE[t];
  LFSR_reg = (u >> 4) | (u >> 28);
  return t;
}
lfsr4.s : (généré par gcc -Os -fomit-frame-pointer -march=pentium)
; dans un prologue de boucle :
       .......
         movl      LFSR_reg, %edx     ; charge le LFSR dans edx
       .......
```

```
; dans la boucle : edx=LFSR, eax=temporaire
       .......
         movl      %edx, %eax
         andl      $15, %eax
; ici, utilisation de eax           ; crée l'index dans la table
         xorl      LFSR4_TABLE(,%eax,4), %edx
         roll      $28, %edx
       .......

; épilogue :
       .......
         movl      %edx, LFSR_reg  ; sauve le LFSR
       .......
```

C'est pratiquement la même chose que le code assembleur initial. Cependant, malgré l'équivalence, je voudrais bien savoir pourquoi gcc choisit `roll $28` au lieu de `rorl $4`... Enfin non, j'ai peur de la réponse.

Cet algorithme reboucle au bout de 4 milliards (2^{32}-1) de bits, soit 1 milliard d'appels. Cela suffit pour des applications simples et peu regardantes sur la « sécurité » (comme des petits jeux ou d'autres applications à caractère informel). Pour générer des nombres en plus grande quantité et avec des propriétés statistiques bien meilleures, à la manière de `/dev/urandom`, d'autres techniques plus efficaces existent mais demandent des tableaux beaucoup plus grands. A l'extrême, on trouve par exemple les « *Mersenne Twister* »...

Tests d'efficacité

Ce petit LFSR rapide tombe très bien, car il va nous permettre de tester facilement la sensibilité de notre algorithme de CRC sur des millions de blocs de 4 K octets.

Normalement, on ne devrait pas pouvoir en tester autant (2^{32-15}=128K) mais comme le LFSR sert aussi à insérer des erreurs, les perturbations pseudo-aléatoires que cela produit créent d'autres blocs différents (décalés) après rebouclage. De plus, comme nous travaillons sur des octets, les 8 rotations de la séquence de 2^{32}-1 bits créent une séquence de 2^{32}-1 octets. Les rotations pseudo-aléatoires additionnelles permettent d'éviter de retester les mêmes blocs.

L'utilisation d'un tel LFSR a l'avantage de fournir des résultats déterministes et reproductibles, au cas où une erreur serait détectée. Pour en trouver la cause, on réinjecte les bonnes valeurs de départ et on utilise un *breakpoint* déclenché par la valeur d'erreur.

Utiliser des données (pseudo-)aléatoires est important lors des tests de CRC car la table et l'ordre des accès ont une influence sur le

reste du programme. En particulier, tester la détection des altérations sur des blocs vides n'a pas beaucoup de sens (« comment ça, ça sent le vécu ? »).

Plus tard, pour les tests de vitesse, il faudra garantir que la LUT de CRC est entièrement contenue dans la mémoire cache, au moyen d'un accès équiprobable à toutes les entrées.

Modèles d'erreurs

Il existe de nombreuses origines d'altérations accidentelles avec des propriétés statistiques différentes. Cependant, elles peuvent toutes être classées dans une des deux catégories suivantes :

- Modification (inversion) d'un ou plusieurs bits indépendants (bruit aléatoire) ;
- Zones de 0 ou 1 contigus (bruit en rafale).

Les algorithmes de CRC sont conçus pour détecter ces deux types d'erreurs qui peuvent se produire sur un support de stockage ou lors d'une transmission.

Si le flux est altéré par une ou des zones de zéros ou de uns, il y a une chance que le CRC, qui est accolé à la fin du bloc de données, soit lui aussi altéré et mis à 0x0000 ou 0xFFFF. Cela augmente les chances de détecter la corruption.

L'immunité aux erreurs en rafale est compréhensible, mais l'immunité au bruit aléatoire est plus difficile à évaluer car cela dépend encore plus du modèle statistique choisi.

En résumé, le banc de test crée un bloc pseudo-aléatoire de 4 K octets puis change progressivement tous les bits dans un ordre pseudo-aléatoire et revérifie le CRC à chaque fois ; les faux positifs sont inscrits dans un tableau pour examiner leur répartition plus tard.

Cela simule pour chaque bloc des conditions de transmission de plus en plus difficiles. C'est une approche « brute » qui permet de réduire le temps de calcul et d'évaluer le comportement du CRC16 dans des cas allant du plus simple au plus extrême.

La position de modification de chaque bit étant choisie par le LFSR, elle est donc équiprobable sur tout le bloc. Cela n'influence pas le modèle ou la détection car un CRC travaille avec des « multiples » (dans **GF(2)**) d'un polynôme. Un signal générant un faux positif peut être appliqué de manière « invariante par décalage » sur le message (la position absolue n'est pas importante). Et pour générer un faux positif, le signal doit être une combinaison par XOR du polynôme appliqué à différentes positions.

Diverses variations sont testées pour les comparer : la signature dans le bloc (donc altérée) ou non, LUT initialisée pseudo-aléatoirement, CRC inversé ou non.

L'objectif essentiel est d'éclairer les zones d'ombre du comportement de l'algorithme. En effet la littérature explique les types d'erreurs qu'un bon CRC doit pouvoir détecter, ainsi que celles qui sont indétectables (les « multiples » du polynôme). On sait que toutes les erreurs du type `2n+1` bits changés sont détectables, mais qu'en est-il des erreurs de type `2n` ? Le meilleur moyen de le savoir reste de le mesurer.

Erreurs dispersées

Le nombre d'essais a été choisi très grand pour pouvoir observer des probabilité inférieures à 1/65536 (la capacité théorique de détection d'un CRC sur 16 bits). Les premières simulations ont surtout détecté des fautes de codage du banc de test (dont les résultats étranges m'ont envoyé sur plusieurs fausses pistes). Le banc de test a finalement fourni des informations importantes synthétisées dans la table 3.

Taux de faux positifs	CRC normal		CRC complémenté - DCRC16_NEGATE	
	LUT normale	LUT aléatoire - DRND_LUT	LUT normale	LUT aléatoire -DRND_LUT
CRC dans le bloc -DCRC_inside	1/62467	1/54334	1/65659	1/57835
CRC hors du bloc	1/67694 *	1/17734	1/67694 *	1/17734
* = pas d'erreurs paires				

Tableau 3 : Statistiques majorantes collectées avec le banc de test mds_crc16_tst.c

▶Parité et répartition des faux positifs

Le banc de test fournit un histogramme mettant en relation le nombre de faux positifs avec le nombre d'erreurs injectées. Dans la plupart des cas, ils sont **équiprobables**, et donc il n'y a pas de différence entre des erreurs à nombres de bits pair et impair.

Toutes les erreurs impaires sont détectées quand le CRC est séparé du bloc et n'est pas altéré. Cela n'est pas réaliste puisqu'en pratique, il est concaténé à la fin de ce bloc et est autant altérable que ses voisins. De plus, le taux total de faux positifs est le même que lorsque le CRC est inclus, ce qui signifie que les erreurs sont « déplacées » et non supprimées. Les erreurs paires sont deux fois plus nombreuses...

Dans un sens, pouvoir reproduire le comportement théorique de l'algorithme permet de valider aussi bien le CRC16 que le banc de test.

Cependant, à moins de protéger le CRC par un code à correction d'erreur, ce comportement ne peut pas apparaître en pratique : bel exemple de théorie inutile :-/

▶Complémentation du CRC :
Elle n'a aucun intérêt, que le CRC soit altéré ou non : les probabilités sont les mêmes, proches du 2^{-16} théorique. Lorsque le CRC est séparé du bloc, les nombres sont identiques, ce qui indique que les mêmes erreurs sont détectées.

La discussion sur la complémentation est donc close : cette étape finale n'a aucun effet et est exclue de la définition du CRC16 de MDS.

▶LUT aléatoire
Cette mesure sort un peu du cadre de l'article mais tente d'expliquer pourquoi une LUT mal conçue a des chances de fonctionner presque aussi bien qu'une LUT normale. Les mathématiques laissent penser qu'une telle LUT reviendrait à effectuer une division avec un nombre irrationnel ou quelque chose d'encore plus compliqué, ce qui améliorerait potentiellement la sensibilité.

Le taux total de faux positifs n'est pas beaucoup plus élevé qu'un CRC16 normal, ce qui reste honorable. On peut noter la différence entre un CRC intégré et séparé. Appliqué au CRC32, cette différence est anecdotique, ce qui explique pourquoi gdups fonctionne correctement malgré une LUT dont les octets sont malencontreusement inversés.

Performance en situation

La campagne de mesure de sensibilité aux erreurs a été calculée sur des Pentium III à 500MHz. En divisant le nombre de signatures de blocs effectués par le temps écoulé, on arrive à environ 22000 signatures par seconde, ou 90M octets par seconde, en comptant le code d'altération des bits. C'est une performance raisonnable qui correspond à environ cinq cycles processeur par octet (sur ces architectures, cela n'a plus de relation directe avec le nombre d'instructions). Dans ces essais, tout se passait en mémoire cache interne, essentiellement en L1 (car l'unique bloc de 4 K octets tient dans les 16 K de L1 disponibles pour les données). En pratique, il y a de fortes chances que les données à lire soient aussi déjà en L1 (ou au moins en L2) :

- ▶Si les données sont à signer, elles ont certainement été écrites en mémoire il y a peu de temps, donc elles sont encore en cache (si la stratégie *writeback* est utilisée).
- ▶Si les données sont lues, elles sont probablement déjà présentes dans la cache du processeur (si la lecture du flux est effectuée par logiciel et non par DMA). Dans le cas où les données ne sont pas déjà toutes dans la mémoire interne, le calcul de la signature permet de « chauffer » la cache, agissant aussi comme un chargement dans le processeur.

A première vue, la vitesse de lecture de la mémoire externe est supérieure à la vitesse de calcul de la signature. On pourrait en profiter pour effectuer d'autres opérations que le calcul, pour profiter de la présence des données dans le processeur. Par exemple, on pourrait recopier et aligner les données pour qu'elles soient utilisables plus facilement par les codecs.

Idéalement, dans l'application finale, il faut fusionner au maximum les différentes opérations. C'est ce qui se passe pour la vérification d'en-tête MDS : les macros de CRC sont clairsemées au milieu du code et ne nécessitent qu'un seul accès à la mémoire par octet, permettant éventuellement au processeur d'entrelacer les instructions des deux fonctions (vérification du CRC **et** de

ATTENTION

Avec l'algorithme CRC16, la **probabilité maximale** de faux positif **à l'intérieur d'un flux aléatoire** est toujours de 2^{-16}, quelle que soit la taille du bloc signé. Pour un flux d'entropie maximale, cette probabilité **dépend uniquement du nombre de bits de la signature**. Si on calcule le CRC de blocs créés en lisant `/dev/urandom`, il y a une chance sur 65536 pour qu'il soit reconnu comme valide. C'est pourquoi CRC32 est préféré dans la plupart des applications actuelles. Cependant, le taux relativement élevé de faux positif est un problème qui ne nous concerne que dans le cas où le flux d'entrée a une entropie maximale, comme par exemple lorsque le flux est désynchronisé et le programme le lit avec un décalage. Là encore, la petite taille des blocs intervient : il y a une chance sur 65536 de faux positifs, mais dans un cadre hypothétique de resynchronisation **uniquement en utilisant les CRC des blocs de données**, il n'y a que 4096 (plus la taille d'en-tête) positions à essayer, soit une probabilité d'erreur de 1/16 au pire. Comme la resynchronisation s'effectue essentiellement en utilisant les en-têtes, dont la taille est très réduite et qui contiennent non seulement un CRC16, mais aussi d'autres informations séquentielles, la probabilité de se synchroniser sur un flux aléatoire est insignifiante. En pratique, la probabilité de faux positif est aussi proportionnelle à la densité du bruit perturbateur. Si cette densité moyenne est d'un bit sur 10^6, la probabilité de faux positif est de $2^{-16}\times10^{-6}$ soit un bloc pour 65 milliards.

De toute façon, les faux positifs ne sont qu'une partie du problème : quand une erreur survient, qu'elle soit détectée ou non, on ne peut pas en général récupérer l'information originale. Travailler sur de petits blocs permet de limiter les pertes. Les codes Reed-Solomon permettent de corriger un certain nombre de bits d'un bloc, mais cette technique est beaucoup trop compliquée pour MDS et rajoute trop d'informations redondantes, le but initial étant d'encapsuler des données compressées sans annuler le gain de place dû à la compression.

la cohérence des données). Les vérifications d'en-tête MDS sont donc théoriquement assez rapides. Une de ses particularités est que les octets sont souvent traités par paires, on peut donc exploiter les macros qui traitent 2 ou 4 octets d'un coup (après alignement).

Vitesse de CRC d'un bloc

Assez d'estimations, passons aux mesures. En partant des hypothèses précédentes, les calculs de signature sont effectués sur un unique bloc de 4 K octets, donc en cache L1. L'accès à la mémoire externe n'étant théoriquement pas un facteur limitant, on cherche dans un premier temps à mesurer la vitesse maximale de calcul. La signature est calculée par `chrono.c` un million de fois sur le bloc (ce qui représente quatre gigaoctets) et la durée d'exécution du programme est mesurée par la commande `/usr/bin/time -f "%U"`. Avec un million d'itérations, la précision est suffisante pour ne pas avoir recours à **RDTSC** qui est d'ailleurs spécifique aux processeurs x86. Le tableau 4 ressemble à un concours de *bogomips,* mais elle permet surtout de voir comment l'algorithme se comporte sur un très large éventail de machines. D'abord, on remarque que la vitesse de calcul évolue linéairement en fonction de la fréquence de travail, à quelques exceptions près. D'ailleurs, la vitesse relative (en cycles d'horloge par octet) permet presque de classer les architectures. Ce sont surtout les plateformes à faible performance qui sont intéressantes, car les plus semblables aux systèmes embarqués, sur lesquels MDS doit pouvoir fonctionner. L'AMD Winchip à 50MHz peut sans peine signer un flux de données sonores de haute qualité, sur la base de 150 à 600K octets par seconde, laissant un peu de marge pour effectuer des traitements sur les données (comme une compression ou décompression rudimentaire). On voit aussi que le nombre d'instructions par octet est indépendant du nombre de cycles par octets. Par exemple, l'Alpha EV6 nécessite six instructions par octet au minimum, ce qui est proche de son score de 7,22, alors que les x86 n'ont besoin que de 3 instructions qui prennent deux fois plus de cycles (entre 5 et 7).

Ces mesures ont été effectuées avec l'algorithme par défaut, mais ce n'est pas le seul disponible : la table 3 suivante met en lumière l'influence d'autres paramètres. La quantité totale de données signées est la même à chaque fois (4 G octets, même pour CRC16_block_multi) pour comparer les routines entre elles. Les différences d'architecture entre les processeurs sont ici les plus flagrantes. Le Pentium subit le plus de différences à cause de son organisation peu flexible, mais les paramètres par défaut pour x86 (-DCRC16=CRC16_block et -DCRC_TYPE=U32) sont adaptés à lui. Les processeurs plus récents peuvent réorganiser les instructions et masquer certaines latences, ce qui réduit par exemple l'influence de la taille de TYPE_CRC16. Par exemple, le Pentium III n'a pas de préférence très marquée entre l'algorithme par bloc ou octet par octet. Cela permet d'espérer que cette dernière

Contributeur	Processeur	bogomips	Temps user	Debit (MO/s)	Vitesse relative (cycles par octet)	Remarques
skc	AMD Winchip 486DX/2 50 MHz	24,93	12m 46,57s	5,3	9,43	Système embarqué
yg	Pentium 133MHz	266	204,36	20,04	6,63	Carte industrielle (SBC format PICMG)
twolife	MIPS32/200MHz	200	191,60s	21	9,52	routeur wrt54g, `-mips32 -mtune=mips32`
yg	Pentium MMX 200MHz	400	135,82	30,15	6,63	
yg	Pentium II 300MHz	600	72,94	56,15	5,34	P2 mobile : L2 intégrée
yg	Alpha EV6 466MHz	921	63,47	64,53	7,22	`-mcpu=21264`, U16 et CRC16_block64 par défaut
yg	Pentium III 500MHz	1000	43,88	93,34	5,35	Ordinateur portable
yg	Pentium III 700MHz	1400	31,29	130,90	5,34	Ordinateur portable
yg	AMD Athlon 850MHz	1700	25,37	161,45	5,26	
EDdy	Pentium-M Dahlias 1600MHz	3170,30	15,43	265	6,027	Ordinateur portable
chrisix	AMD Athlon XP 2000+ (1667MHz)	3309,56	14,67	279	5,97	
zephred	AMD Athlon XP 3200+ (2100MHz)	4136,96	12,15	337	6,23	
zephred	Pentium 4 à 2,4GHz	4771,02	10,76	380	6,31	
skc	Intel Xeon 3.00GHz		8,62	475	6,31	P4 en mode 64 bits

Tableau 4 : Mesures et calculs de performance de la routine CRC16_block()

Processeur	CRC16_block()		CRC16_ byte() (octet par octet)	CRC16_ block_multi() (2 blocs à la fois)	CRC16_byte_ multi() (2 octets à la fois)
	CRC_ TYPE=U16	CRC_ TYPE=U32			
Pentium 133MHz	265,98	204,36	308,23	209,85	321,78
Pentium MMX 200MHz	176,78	135,82	184,39	142,86	205,18
Pentium II 300MHz	74,20	72,87	82,20	40,57	75,37
Alpha EV6 466MHz	63,47 (CRC16_block64)	63,48	80,14	34.05	44.55
Pentium III 500MHz	44,50	43,69	49,30	24,33	45,21
Pentium III 700MHz	31,77	31,29	35,20	18,81	32,28
AMD Athlon 850MHz	30,21	25,38	32,40	13,08	36,83

Tablean 5 : Comparaison du temps d'exécution (en secondes) de différentes routines de CRC16 pour signer 4G octets

option, utilisée à défaut de mieux sur les machines Big Endian, ne sera pas excessivement lente sur des processeurs tels que le PowerPC. L'algorithme `CRC16_byte_multi()`, traitant simultanément deux blocs, remporte un succès unanime sur les processeurs récents (apportant presque un doublement de vitesse), ce qui encourage son emploi dans le futur. Une version octet par octet est donc née, appelée `CRC16_byte_multi()` et destinée à permettre aux machines Big Endian de rattraper un peu leur retard sur les autres. Cette fonction compile très mal sur x86 à cause du manque de registres, mais l'EV6 semble bien l'accepter, comme le témoigne le doublement de la vitesse par rapport à la version simple.

Le bug le plus sournois

« Tant qu'on ne mesure pas la vitesse d'un programme, on ne la connaît pas. » Tel est le leitmotiv de cette série d'articles qui donne lieu à des campagnes de mesures pour s'assurer que tout fonctionne comme prévu. C'est très bien si on n'est pas pressé car on tombe alors sur plein de choses inattendues. Les bugs, d'abord : on peut en deviner certains en regardant les résultats. Mais celui-là m'a vraiment interpellé : la vitesse de CRC chutait de moitié au bout d'un certain nombre de lancements du programme. J'ai tout essayé : la raison ne venait ni du compilateur (toutes les options y sont passées), ni de l'alignement des données, ni de conflits de bancs de mémoire cache, ni d'un mauvais codage...

Les symptômes ressemblaient à cela :

```
$ for i in a z e r t y ; do /usr/bin/time -f "%U" ./chrono ; done
18.83
18.83
37.81  <-- ralentissement
38.14
38.14
38.13
```

J'ai essayé d'intercaler différents programmes, pensant que le ralentissement provenait de mauvais alignements du code, puis des données, puis j'ai échafaudé plusieurs hypothèses invérifiables. J'ai même tenté d'exécuter une instruction `WBINVD` mais même en tant que *root*, elle est privilégiée (réservée au mode noyau). La vitesse normale revenait « au bout d'un certain temps » que je n'arrivais pas à reproduire...

L'heureux dénouement est venu au cours d'une discussion avec Benoît-Pierre, qui a suggéré que cela pouvait être causé par la gestion automatique d'énergie du processeur. En effet, j'ai effectué ces mesures étranges sur un ordinateur portable à base de Pentium 3 avec la « Technologie SpeedStep (R) » qui... réduit automatiquement la vitesse du processeur quand celui-ci commence à chauffer, pour éviter de faire du bruit avec le ventilateur. J'avais en effet activé cette fonction dans le BIOS pour faire tourner des mesures de sensibilité pendant la nuit et je n'avais pas compris pourquoi ce processeur à 700MHz était presque deux fois plus lent que les autres à 500MHz. L'autre portable à base de Pentium II à 300MHz donnait aussi des résultats variables à cause de la fonction « ventilateur désactivé ». Tout s'explique, donc ;-)

A creuser

Le code de CRC passe essentiellement son temps à consulter la LUT. Les trois instructions sur x86 se transforment en six instructions sur processeurs RISC à cause de leurs modes d'adressage très limités. La nature totalement sérielle de l'algorithme empêche les ordinateurs de profiter des pipelines superscalaires et le code « multiple entrelacé » double la taille du code. La question qui se pose est alors : existe-t-il des algorithmes intermédiaires, purement arithmétiques, entre le CRC et le checksum ? Ceux-ci devraient éviter tout accès à une table et pourraient alors exécuter sans peine 4 à 6 opérations arithmétiques et logiques par octet (dont certaines en parallèle).

Une voie serait l'utilisation de « générateurs congruentiels » : comme les LFSR, ce sont des générateurs de suites de nombres pseudo-aléatoires qui travaillent sur des entiers normaux (contrairement aux CRC). Ils sont définis par une fonction de récurrence du type `N = ((N * a) + b ) % c`. Dans l'arithmétique en **GF(2)**, le XOR correspond à l'addition et à la soustraction classique, et le décalage à la multiplication ou la division (selon la direction). On peut donc voir les LFSR comme des générateurs congruentiels dont `c` serait fixé à 2. Ou à l'inverse, les générateurs congruentiels seraient des LFSR généralisés à des bases non binaires. L'emploi de l'opérateur `modulo`, qui fait normalement appel à une division, est un facteur limitant la vitesse de calcul. Dans le cas qui nous concerne, `a`, `b` et `c` sont des constantes et on pourrait alors utiliser une multiplication par la réciproque de `c` (donc précalculée) au lieu de la division (c'est la technique utilisée dans les processeurs ne disposant pas d'unité de division entière, tel l'Alpha). L'opération de modulo s'écrit alors :

```
t = (x * RECIPROQUE) >> AJUSTEMENT; /* division */
reste = x - (c * t);    /* calcul du reste par différence avec x */
```

Même en évitant la division, l'opération reste lente. Mais surtout, la méthode n'a pas d'intérêt pratique tant que des valeurs de `a`, `b` et `c` ne garantissent pas une détection d'erreur aussi bonne qu'un CRC. La taille du champ de signature est limitée alors que la vitesse des processeurs ne l'est pas, il serait dommage de troquer la fiabilité pour un petit gain de vitesse à court terme.

Conclusion

A la fin de cette aventure, nous disposons d'un algorithme de CRC16 validé, portable, optimisé (à divers degrés de portabilité) et à la fiabilité mesurée. Il est disponible dans le fichier `mds_crc16.c` sur Internet. Son développement a été riche en enseignements !

- ▶ D'abord, il faut faire très attention à la définition d'un algorithme de CRC. Le meilleur moyen de vérifier qu'il fonctionne comme prévu est... de le faire tourner en comparant les résultats avec d'autres versions de code, même les moins évoluées.
- ▶▶ Réutiliser un code source sur Internet n'exonère pas d'apprendre comment s'en servir et pourquoi, ce qui évite de commettre des erreurs typiques de programmeur... L'algorithme de calcul de CRC n'est pas le plus difficile à réaliser, c'est la conception de la LUT et surtout la modélisation qui ont été difficiles.
- ▶ Privilégier l'option d'optimisation pour la taille : gcc -Os donne de bons résultats (sur x86), mais il faut encore surveiller l'alignement des instructions au début des boucles. Ici ce n'est pas critique car sur les processeurs utilisés, le temps d'exécution est largement supérieur au temps de décodage. Les options `fomit-frame-pointer` et `-march=(votre architecture)` aident aussi un peu.
- ▶ Écrire du code à la fois vraiment portable et efficace est toujours aussi difficile. Le langage C n'est pas l'idéal. Les versions optimisées restent réservées aux machines Little Endian.
- ▶▶ J'en ai profité pour tester mes nouveaux joujoux à base d'Alpha EV6, me permettant de préparer un portage sur x86-64 tout en évitant aux codes sources d'être complètement x86-centriques, j'ai même évité l'emploi de code assembleur !
- ▶▶ Une version Big Endian encore plus optimisée est possible au prix de nombreux autres efforts, y a-t-il des volontaires ?

Pour continuer l'étude de cet algorithme de CRC16, je vous invite à lire les différentes variations du code dans mds_crc16.c. Elles ne sont pas réservées à l'usage unique de MDS et peuvent servir à d'autres applications. Pour une protection encore plus solide, l'algorithme CRC32 nécessite juste deux octets de plus par signature. Diverses réalisations sont disponibles dans la `zlib` ainsi que d'autres Logiciels libres ou du domaine public. Et si vous êtes paranoïaque, vous disposez maintenant de suffisamment d'informations pour créer votre CRC64, CRC128... avant de passer à SHA ou MD5. Encore une fois, merci aux noctamoules qui m'ont gentiment aidé en collectant des mesures sur leurs machines. Des détails sur le format MDS ont transpiré dans cet article, le suivant va enfin lever le rideau.

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

La RFC n°3533 décrivant le conteneur Ogg «*The Ogg Encapsulation Format Version 0*» :
http://www.ietf.org/rfc/rfc3533.txt

La description détaillée du conteneur Ogg «*Ogg logical bitstream framing*» :
http://www.xiph.org/vorbis/doc/framing.html

L'utilitaire GDUPS : **http://f-cpu.seul.org/whygee/lm-gdups/**

La suite d'utilitaires Info-ZIP contient une version peaufinée de CRC32 : **http://info-zip.org/pub/infozip/**

Et dans votre noyau favori : crc32.h

Yann Guidon,