Les fonctions de hachage différentielles, application à la génération de graphes d'états

Bernard Cousin

To cite this version:

Bernard Cousin. Les fonctions de hachage différentielles, application à la génération de graphes d'états . Colloque francophone sur l'ingéniérie des protocoles (CFIP'93), Sep 1993, Montréal, Canada. pp. 525-541. hal-01490723

HAL Id: hal-01490723
https://hal.science/hal-01490723

Submitted on 15 Mar 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Les fonctions de hachage différentielles : application à la génération de graphes d'états

Bernard Cousin

IRISA - Université de Rennes-I
Campus universitaire de Beaulieu
35042 - Rennes cédex
FRANCE

Résumé. *Les fonctions de hachage différentielles possèdent un procédé de calcul différentiel qui permet d'optimiser le temps d'exécution des fonctions de hachage vis-à-vis du procédé usuel tout en obtenant exactement la même valeur de hachage. Nous définissons la propriété d'être différentielle pour une fonction de hachage, puis nous montrons que toutes les fonctions de hachage ne possèdent pas cette propriété, enfin nous proposons une caractérisation de l'ensemble des fonctions qui la possède. Nous nous attachons ensuite à montrer le gain en performance induit par l'utilisation d'algorithmes différentiels de quatre fonctions de hachage trouvées dans la littérature. Les gains observés étant proportionnels à la taille de la clef, ils peuvent être extrêmement importants. Nous utilisons alors une fonction de hachage différentielle pour la génération de graphes d'états issus de modèles décrit au moyen de réseaux de Petri.*
Mots-clefs : fonctions de hachage, techniques de vérification, étude de performance, graphe d'accessibilité, réseau de Petri.
Abstract. *Differential hashing functions have a differential computation process which enables hashing function processing time to be optimized. Formal definition of the differential property for hashing functions is proposed, and we show that not all hashing functions have this property, then we propose a characterization of the differential hashing function set. Next, we show the performance acceleration produced by the differential algorithms applied to five hashing functions found in common applications. The observed accelerations can be significant because they are proportional to key length. Last we study the performances of differential hashing functions for the reachability graph exploration of distributed systems specified by a Petri net.*
Key words : hashing functions, validation technique, performance study, reachability graph, Petri net.

1. Introduction

De nombreux outils de validation sont basés sur l'établissement de graphes d'états qui permettent de décrire tout ou partie de l'évolution de systèmes répartis modélisés formellement. De très nombreux graphes ont été développés : arbres de couverture [Karp 69], graphes réduits ou minimaux [Itoh 83, Zhao 86, Finkel 91],

composantes, il peut être plus performant de déduire la nouvelle valeur de hachage de la première valeur que d'appliquer la fonction de hachage à toutes les composantes de la nouvelle clef.

Au procédé de calcul différentiel on associe une famille de *fonctions mono-différentielles de calcul*. Chaque fonction permet de calculer la valeur de hachage d'une clef connaissant la valeur de hachage d'une autre clef qui ne *diffère* de la première clef que de la valeur d'un seul composant donné. Par composition, on prouve aisément que les fonctions mono-différentielles permettent de calculer la valeur de hachage d'une clef à partir de la valeur de hachage d'une autre clef différant de la première de plus d'une composante.

Une première question est soulevée par notre proposition : Est-il possible d'associer un procédé de calcul différentiel à toute fonction de hachage ? Au cours du deuxième chapitre, après une définition formelle des fonctions de hachage différentielles, nous prouverons que la réponse à cette question est négative. Cependant nous présenterons une caractérisation de l'ensemble des fonctions de hachage auxquelles il est possible d'associer un procédé de calcul différentiel et nous verrons par la suite que cet ensemble recouvre la majorité des fonctions de hachage habituellement utilisées.

Le chapitre suivant (le troisième) sera l'occasion pour nous de répondre à une question supplémentaire : Le procédé de calcul différentiel est-il plus performant que le procédé usuel pour toutes les fonctions de hachages différentielles ? Nous verrons que ce n'est pas toujours le cas et nous préciserons dans quelles conditions. Toutefois cette réponse n'est pas définitive car l'algorithme utilisé lors de l'implantation, tant pour le procédé différentiel que pour le procédé usuel, a une influence prépondérante sur leurs performances. Nous verrons qu'un implantation naïve produit déjà une très bonne accélération de traitement, et qu'une implantation compacte autorisant un codage très efficace, réduit le temps de calcul considérablement.

Le quatrième chapitre sera pour nous l'occasion de mesurer les performances des fonctions de hachage différentielles pour la génération de graphes d'états utilisée par un outil de validation basé sur la description formelle de systèmes répartis au moyen de réseau de Petri. Cet exemple nous permettra de mettre en évidence les avantages et les limites de notre technique.

2. Les fonctions de hachage différentielles

2.1 Définitions

Une fonction de hachage peut être décrite comme une application h à N variables d'un produit d'ensemble $\prod_{k=1}^{N} A_k$ vers un ensemble B. C'est à dire :

$$\forall <x_1,...,x_i,...,x_N> \in \prod_{k=1}^{N} A_k, \exists\ h(<x_1,...,x_i,...,x_N>) \in B.$$

Nous appellerons $h(<x_1,...,x_i,...,x_N>)$ la valeur de hachage de la clef $<x_1,...,x_i,...,x_N>$.

Toutes les fonctions de hachage n'admettent pas de fonctions de calcul différentiel. Par exemple la fonction de hachage "⊗" construite sur un produit d'ensembles binaires et définie par l'opération binaire "et logique" n'est pas différentielle. En effet, la fonction de calcul différentiel associée au premier composant ne peut pas être déterminée : $F_1(0,0,1)$ peut être égale soit à $F_1(\otimes<0,1>,0,1)=\otimes(<1,1>)=1$, soit à $F_1(\otimes<0,0>,0,1)=\otimes(<1,0>)=0$, ce qui est en contradiction avec la définition d'une fonction qui assure l'unicité de l'image.

Propriété :

Les fonctions de hachage, qui admettent une fonction de calcul différentiel pour leur $i^{ème}$ composant, sont caractérisées par la propriété suivante :

$$\forall <x_1,...,x_i,...,x_N> \in \prod_{k=1}^{N} A_k,\ \forall <y_1,...,y_i,...,y_N> \in \prod_{k=1}^{N} A_k, \qquad [2]$$

$$h(<x_1,...,x_i,...,x_N>)=h(<y_1,...,x_i,...,y_N>) \Leftrightarrow$$
$$h(<x_1,...,y_i,...,x_N>)=h(<y_1,...,y_i,...,y_N>).$$

Nous allons prouver que la propriété [2] est une condition nécessaire et suffisante pour que la fonction de hachage admette une famille complète de fonctions de calcul mono-différentiel.

Prouvons d'abord que c'est une condition nécessaire. Soit h une fonction de hachage possédant une fonction de calcul différentiel pour sa $i^{ème}$ composante, alors d'après la définition [1] on a la propriété suivante : la fonction de calcul différentiel F_i de $B x A_i x A_i$ vers B est telle que : $\forall <x_1,...,x_i,...,x_N> \in \prod_{k=1}^{N} A_k$, $\forall y_i \in A_i$,

$$h(<x_1,...,x_i,...,x_N>)=F_i(h(<x_1,...,y_i,...,x_N>),\ y_i,\ x_i)$$

Donc $\forall <x_1,...,x_i,...,x_N> \in \prod_{k=1}^{N} A_k$, $\forall <y_1,...,y_i,...,y_N> \in \prod_{k=1}^{N} A_k$, d'après la définition [1] on peut écrire que : $h(<x_1,...,x_i,...,x_N>) = F_i(h(<x_1,...,y_i,...,x_N>),\ y_i,\ x_i)$.

Si on suppose les prémices de la condition [2] :

$$h(<x_1,...,y_i,...,x_N>) = h(<y_1,...,y_i,...,y_N>)$$

alors on en déduit que $h(<x_1,...,x_i,...,x_N>) = F_i(h(<y_1,...,y_i,...,y_N>),\ y_i,\ x_i)$.

On peut écrire aussi d'après la définition [1] que $\forall <y_1,...,y_i,...,y_N> \in \prod_{k=1}^{N} A_k$,

$$\forall x_i \in A_i,\ h(<y_1,...,x_i,...,y_N>) = F_i(h(<y_1,...,y_i,...,y_N>),\ y_i,\ x_i) .$$

On en déduit immédiatement la conclusion de la condition [2] :

$$h(<x_1,...,x_i,...,x_N>) = h(<y_1,...,x_i,...,y_N>). \text{ (cqfd)}$$

Ensuite nous prouvons que c'est une condition suffisante. Soit une fonction de hachage h ayant la propriété suivante :

$$\forall <x_1,...,x_i,...,x_N> \in \prod_{k=1}^{N} A_k,\ \forall <y_1,...,y_i,...,y_N> \in \prod_{k=1}^{N} A_k,$$

3. Etude des performances

3.1 Présentation

Nous allons montrer dans ce chapitre l'augmentation des performances que l'on peut attendre de l'utilisation du procédé de hachage différentiel. Pour ce faire nous allons comparer les performances obtenues par des fonctions de hachage en utilisant leur algorithme usuel à celles obtenues par l'utilisation d'un algorithme différentiel.

Dans la littérature on peut trouver de nombreuses études des performances des méthodes de hachage [Froidevaux 90, Wirth 87, Knott 75, Knuth 73, Ulmann 72, Lum 71, Morris 68, etc]. Notre étude se focalise sur l'étude des performances des fonctions de calcul de la valeur de hachage (adresse) sans nous préoccuper des fonctions d'accès et de résolution des conflits. En effet notre technique s'adapte sans aucun problème à toutes les fonctions d'accès et de résolution des conflits, l'augmentation des performances constatées lors de l'utilisation d'un algorithme différentiel pour implanter les fonctions de calcul de la valeur de hachage sera donc intégralement conservée.

Nous avons choisi 5 fonctions de hachage parmi celles trouvées dans la littérature, cet échantillon ne couvre pas toutes les fonctions de hachage existantes, toutefois ces 5 fonctions couvrent un spectre de fonctions très fréquemment utilisées. Nous ne cachons que la fréquence d'utilisation n'est pas le seul critère qui soit intervenu dans le choix de ces cinq fonctions, la simplicité de conception des algorithmes usuels de ces fonctions de hachage, gage de bonnes performances, est aussi intervenue comme critère. En effet, un algorithme simple et de code court est souvent plus rapide qu'un algorithme complexe et long. Actuellement des études supplémentaires sont entreprises pour rechercher l'algorithme différentiel de nouvelles fonctions de hachage, et ainsi augmenter le spectre de notre étude.

Nos fonctions de hachage utilisent les opérations de base suivantes : multiplication/division, modulo, addition/soustraction, pliage ("folding"), extraction de bits particuliers [Knott 75]. D'autres opérations peuvent être proposées : élévation à la puissance/racine, changement de base, calcul polynomiaux, etc [Lum 71]. Ces autres opérations ont souvent l'inconvénient d'être difficile à calculer, c'est pourquoi nous avons choisi de ne pas les utiliser dans un premier temps. Pour répartir équitablement sur l'ensemble des valeurs de hachage possibles, les opérations de base peuvent faire intervenir des coefficients multiplicatifs qui sont généralement choisis parmi les nombres premiers. Les cinq fonctions de hachage proposées utilisent généralement une combinaison de plusieurs opérations de base.

La première fonction (n°1) utilise une technique de pliage basée sur l'utilisation de l'opération logique "ou exclusif". La deuxième fonction (n°2) étudiée effectue la somme de tous les composants. La troisième fonction (n°3) utilise une technique d'extraction de quelques groupes de bits parmi la suite des bits formant la clef. La quatrième fonction (n°4) est une somme pondérée des composants par des coefficients premiers. La dernière fonction (n°5) désignée dans la littérature sous le nom "hpjw" combine des opérations logiques de rotation et d'addition.

Nous allons voir que nous sommes parvenus à trouver les algorithmes différentiels pour chacune de ces cinq fonctions, toutefois si des accélérations

très grand nombre de collisions dégradant de manière considérable les performances globales de la méthode. Toutefois, il ne faudrait pas en conclure que les résultats obtenus sur les temps d'exécution des fonctions de hachage sont incorrects. En effet, le temps d'exécution des fonctions de résolutions des collisions étant, en général, indépendant de la taille des clefs, il tend à devenir négligeable pour des grandes clefs vis-à-vis du temps d'exécution des fonctions de hachage [Deudon 92].

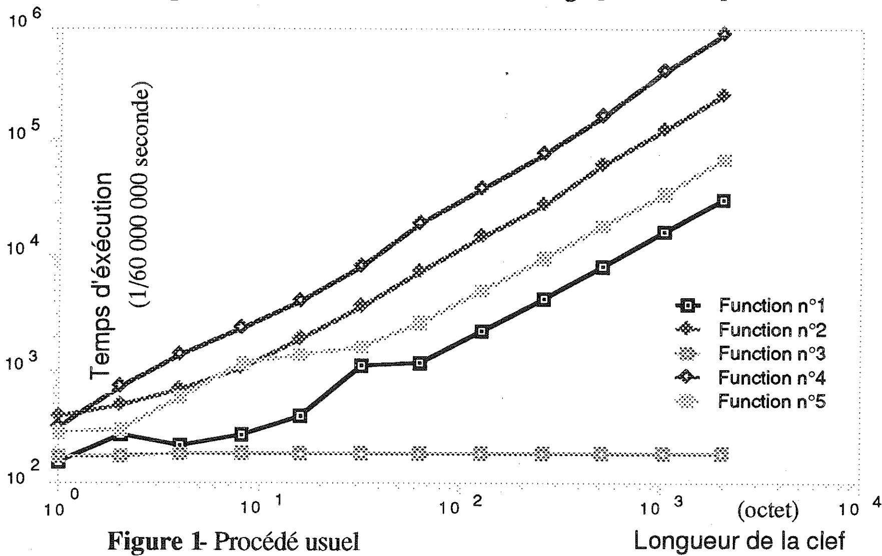

Figure 1- Procédé usuel

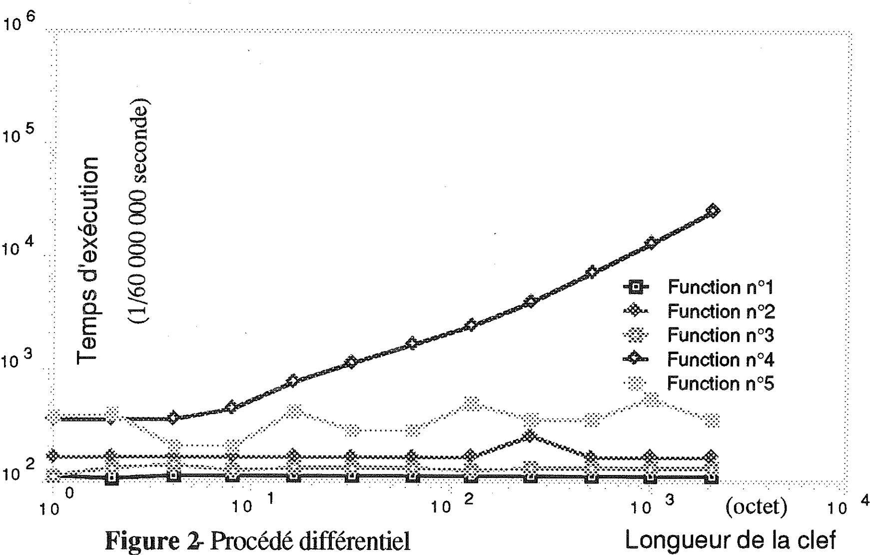

Figure 2- Procédé différentiel

Remarquons que certaines fonctions (notamment la fonction n°1) propose un procédé de calcul différentiel multiple dont la durée d'exécution de l'algorithme est proche de celle de l'algorithme associé au procédé mono-différentiel. C'est cette fonction de hachage munie de ce procédé de calcul de la valeur de hachage que nous utiliserons par la suite.

4. Application à la génération de graphes d'états

Le codage des états nécessite une taille de stockage importante. En effet, cette taille est fonction du parallélisme et de la précision de la modélisation du système étudié. Dans la littérature on peut trouver les chiffres suivants : 1916 octets pour la modélisation du "P-channel protocol" [Doldi 92], typiquement une centaine d'octets pour Holzmann [Holzmann 91], notre propre étude tend à montrer que pour des modèles d'applications ou de protocoles réels il est fréquent d'obtenir des états de taille conséquente : plusieurs centaines d'octets.

Certains logiciels de génération de graphes d'états préfèrent utiliser une technique de recherche où la structure de stockage des données est un arbre binaire (et équilibré) ou encore une structure hiérarchique spécifique [Chiola 87, Crubillé 88]. Ces techniques ne sont pas les plus fréquemment utilisées car elles présentent, soit une trop grande spécificité (c'est à dire qu'elles ne sont adaptées qu'à une certaines classe de propriétés), soit un surcoût tant en espace (pointeur d'indirection) qu'en temps d'accès (parcours des indirections), vis-à-vis de la technique de recherche par hachage.

Nous avons effectué l'étude des performances à l'aide de l'outil Bouster [Campergue 92] sur un jeu de plusieurs modèles décrit au moyen de réseaux de Pétri ayant un nombre de places variant de 2 à 50000, un nombre de transitions de 1 à 50000, un nombre d'arcs de 1000 à 100000 et générant plusieurs dizaines de milliers d'états. Cette étude montre que les fonctions de calcul de la clef de hachage ("f_hashing()"), de recherche et d'insertion d'un nouveau noeud ("put_in_table()") et de comparaison de chaîne de caractères ("bcmp()") sont les trois fonctions les plus intensément utilisées : 15 à 30 % du temps de calcul du processeur en mode utilisateur pour chacune d'entre-elles en fonction des différents modèles et dans un ordre variable. L'ordre et la valeur du taux d'utilisation de ces trois fonctions sont variables car leur performance dépend du nombre de collisions, qui lui même dépend à la fois de la fonction de hachage choisie et de la taille de la table de hachage. Toutes les autres fonctions sans exception utilisent chacune moins de 10% du temps du processeur et pour la grande majorité d'entre-elles moins d'un millième.

Ces chiffres montrent à la fois l'importance et les limites des gains que l'on peut espérer avec notre méthode. En effet, le processeur passe en moyenne près de 40% de son temps dans le code des fonctions chargées d'effectuer le calcul de la clef de hachage et d'appliquer l'algorithme de recherche et d'insertion d'un nouveau noeud. Une fonction de hachage rapide, judicieuse et équilibrée devrait être capable de réduire de manière conséquente ce temps en minimisant d'une part la fréquence des collisions et d'autre part le temps de calcul de la clef. Toutefois l'accélération des performances induites par ce type de technique ne peut pas réduire de manière magique la complexité inhérente au modèle étudié, notamment le nombre gigantesque de noeuds

que celui du procédé de calcul usuel, troisièmement cette technique de calcul différentiel ne s'applique raisonnablement que si l'obtention des composantes modifiées entre une clef et la suivante est d'un coût faible voire nul. L'application proposée pour mettre en oeuvre les fonctions de hachage différentielles possède toutes les caractéristiques qui font que l'on obtient un gain en performance conséquent.

Remerciements

Je remercie Christian Houillon et Guillaume Deudon pour leur réalisation et leur étude des performances des fonctions de hachage différentielles sous une forme préliminaire [Deudon 92]. Mes remerciements aussi à Christian Ronse pour sa participation à la caractérisation des fonctions différentielles.

Bibliographie

[Aho 75] A.V.Aho, M.J.Corasick, "Efficient string matching : an aid to bibliographic search". Communications of the ACM vol 18 n°6, June 1975, p333-343.

[Algayres 91] B.Algayres,L.Doldi, H.Garavel, Y.Lejeune, C.Rodriguez, "Vesar : a pragmatic approach to formal specification and verification", Computer Network and ISDN Systems : special issue on tools for FDT's, vol 25 n°7, February 1991.

[Boyer 77] R.S.Boyer, J.S.Moore, "A fast string searching algorithm". Communications of the ACM vol20 n°10, 1977, p762-772.

[Campergue 92] C.Campergue, C.Nouaille, "Bouster : génération parallèle du GMA associé à un réseau de Petri", rapport de recherche, Bordeaux-France, Juin 1992.

[Cacciari 92] L.Cacciari, O.Rafiq, "On improving reduced reachability analysis", proceedings of 5th international conference on formal description techniques (FORTE'92), Lannion-France, 13-16 octobre 1992.

[Chiola 90] G.Chiola, "Symbolic reachability graph", 11h international workshop on theory and applications of Petri nets, Paris-France, 1990.

[Cousin 93] B.Cousin, "Differential hashing functions", 5th international conference on computing and information (ICCI'93), Sudbury - Canada, 27-29 May 1993.

[Crubillé 88] P.Crubillé, "Mec : un outil de clacul sur les systèmes de transitions", Colloque francophone sur l'ingéniérie des protocoles (CFIP'88), Bordeaux-France, Septembre 1988, Eyrolles, 1988.

[Deudon 92] G.Deudon, C.Houillon, "Techniques de hachage", rapport de recherche ENSERB, Bordeaux-France, Juin 1992.

[Dimitrijevic 89] D.D.Dimitrijevic, M.S. Chen, "Dynamic state explosion in quantitative protocol analysis". Proceedings of PSTV-IX, Twente-Netherland, 6-9 June 1989.

[Doldi 92] L.Doldi, P.Gauthier, "Veda-2 : power to the protocol designers", proceedings of FORTE'92, Lannion-France, 13-16 octobre 1992.

[Dutheillet 89] C.Dutheillet, S.Haddad, "Regular stochastic Petri nets", Workshop on Petri nets and its application, Bonn - Germany, 1989.

[Finkel 91] A.Finkel, C.Johnen, "Construction efficace du graphe de couverture minimal : application à l'analyse de protocole", Colloque francophone sur l'ingéniérie des protocoles (CFIP'91), Pau-France, Septembre 1991, Hermès, 1991.

[Froidevaux 90] C.Froisdevaux, M-C.Gaudel, M.Soria, "Types de données et algorithmes", McGraw-Hill, 1990.

Annexe A

```
/*** Fichier fct_hash.c ***//*** version 1.2 - du 11/12/93 ***/
extern int n;   /* Taille du tableau de hachage */
extern char* key;      /* La clef */
extern int l;          /* Longueur de la clef (en octet)*/

int function_1() {
/* "ou_exclusif entre composants de 2 octets        */
register int i; register unsigned result = 0;
int x=0;
        for (i=1; i<l; i+=2){
                x = key[i-1]<<8;
                x ^= key[i];
                result ^= x;
        }
        if (l % 2) result ^= key[l-1];
        return (result % n);
} /* function_1() */

int function_2() {
/* somme ponderee de tous les composants  */
 register int i; register unsigned result = 0;
        for (i=1; i<=l; i++) result += i*key[i-1];
        return(result % n);
} /* function_2() */

int function_3() {
/* extraction des premier troisieme et dernier octets */
 register unsigned result = 0;
        result = key[0] ^ key[1]<<3 ^ key[l-1]<<7;
        return(result % n);
} /* function_3() */

int function_4() {
/* technique multiplicative et somme pondérée       */
/*      par des nombres premiers                    */
 register int i; register unsigned result = 0,g;
static  int  coef[19]={67,61,59,53,47,43,41,37,31,29,23,  17,13,11,9,7,5,3,1};
        for (i=0; i<l; i++){
                result = result * 5 + key[i] * coef[i % 19];
                if ((g = result & 0xF0000000) != 0) {
                        result ^= g ; result += (g>>28) ;
                }
  }
  return (result % n);
}/* function_4() */
```

```
int function_4_diff(int result_prec, int indice, char ancien_octet, char nouvel_octet){
/* technique multiplicative et somme pondérée        */
/*       par des nombres premiers                    */
register int result = 0; register int g;
static  int  coef[19]={67,61,59,53,47,43,41,37,31,29,  23,17,13,11,9,7,5,3,1};
register int x;  int i;
        x=1;
        for (i=1; i<=l-1-indice; i++) {
                x *= 5;
                f ((g = x & 0xF0000000) != 0) {
                        x ^= g ; x += (g>>28) ;
                }
        }
        x *= coef[indice%19];
        if ((g = x & 0xF0000000) != 0) {
                x ^= g ; x += (g>>28) ;
        }
        x *= nouvel_octet - ancien_octet;
        result = result_prec + x;
        if ((g = result & 0xF0000000) != 0) {
                result ^= g ; result += (g>>28) ;
        }       return(result % n);
} /* function_4_diff() */

int function_5_diff(int result_prec, int indice, char ancien_octet, char nouvel_octet) {
/*      fonction hpjw              */
register unsigned int result; register int x_haut, x_bas;
        switch((l-1-indice) % 7) {
        case 0 : result = result_prec ^ (nouvel_octet ^ ancien_octet); break;
        case 1 : result = result_prec ^ (nouvel_octet ^ ancien_octet)<<4; break;
        case 2 : result = result_prec ^ (nouvel_octet ^ ancien_octet)<<8; break;
        case 3 : result = result_prec ^ (nouvel_octet ^ ancien_octet)<<12; break;
        case 4 : result = result_prec ^ (nouvel_octet ^ ancien_octet)<<16; break;
        case 5 : result = result_prec ^ (nouvel_octet ^ ancien_octet)<<20; break;
        case 6 : x_bas = ((nouvel_octet ^ ancien_octet) & 0x0f)<<24;
                 x_haut = ((nouvel_octet ^ ancien_octet) & 0xf0)>>4;
                 result = result_prec ^ (x_haut ^ x_bas);
                break;
        }       return(result % n);
}/* function_5_diff() */
```