Groupe de travail Réseau
Request for Comments : 3174
Catégorie : Information
Traduction Claude Brière de L'Isle

D. Eastlake, 3rd, Motorola
P. Jones, Cisco Systems
septembre 2001

Algorithme 1 de hachage sécurisé (SHA-1)

Statut de ce mémoire
Le présent mémoire apporte des informations pour la communauté de l'Internet. Il ne spécifie aucune sorte de norme de l'Internet. La distribution du présent mémoire n'est soumise à aucune restriction.
(La présente traduction incorpore les errata 328 et 329 du 20/03/2004)

Résumé
L'objet du présent document est de rendre l'algorithme de hachage SHA-1 *(Secure Hash Algorithm 1)* facilement disponible à la communauté de l'Internet. Les États Unis d'Amérique ont adopté l'algorithme de hachage SHA-1 décrit ici comme norme fédérale de traitement de l'information. La plus grande partie du texte présenté ici par les "auteurs" a été tirée de FIPS 180-1. Seule la mise en œuvre en code C est "originale".

Remerciements

La plus grande partie du texte présenté ici est tirée de [FIPS 180-1]. Seule la mise en œuvre du code C est "originale" mais son style est similaire à celui des RFC précédemment publiées sur MD4 et MD5, les [RFC1320] et [RFC1321].

SHA-1 se fonde sur des principes similaires à ceux utilisés par le professeur Ronald L. Rivest du MIT lorsque il a conçu l'algorithme de résumé de message [MD4] et il est modélisé d'après cet algorithme de la [RFC1320].

Des commentaires utiles qui ont été incorporés ont été reçus de Tony Hansen et de Garrett Wollman. Qu'ils en soient ici remerciés.

Table des matières

1. Généralités

Note : Le texte qui figure ci-dessous est principalement tiré de [FIPS 180-1] et les assertions qu'il contient sur la sécurité de SHA-1 sont faites par le Gouvernement des USA et par l'auteur de [FIPS 180-1] et non par les auteurs du présent document.

Ce document spécifie un algorithme de hachage sécurisé (SHA-1, *Secure Hash Algorithm*) pour calculer une représentation condensée d'un message ou d'un fichier de données. Lorsque un message d'une longueur quelconque < 2^64 bits est entré,

SHA-1 produit un résultat de 160 bits appelé un résumé de message. Le résumé de message peut alors, par exemple, être l'entrée d'un algorithme de signature qui génère ou vérifie la signature pour le message. Signer le résumé de message plutôt que le message améliore souvent l'efficacité du processus parce que le résumé de message est habituellement beaucoup plus petit que le message. Le même algorithme de hachage doit être utilisé par le vérificateur d'une signature numérique comme il a été utilisé par le créateur de la signature numérique. Tout changement du message en transit va, avec une très forte probabilité, résulter en un résumé de message différent, et la vérification de la signature va échouer.

SHA-1 est dit décurisé parce qu'il est impossible par le calcul de trouver un message qui corresponde à un résumé de message donné, ou de trouver deux messages différents qui produisent le même résumé de message. Tout changement à un message en transit va, avec une très forte probabilité, résulter en un résumé de message different, et la vérification de la signature va échouer.

La Section 2 ci-dessous définit la terminologie et les fonctions utilisées comme éléments de la construction de SHA-1.

2. Définitions des chaînes binaires et des entiers

La terminologie suivante sera utilisée pour les chaînes binaires et les entiers :

a. Un chiffre hexadécimal est un élément de l'ensemble {0, 1, ... , 9, A, ... , F}. Un chiffre hexadécimal est la représentation d'une chaîne de 4 bits. Exemples : 7 = 0111, A = 1010.

b. Un mot égale une chaîne de 32 bits qui peut être représentée comme une séquence de 8 chiffres hexadécimaux. Pour convertir un mot en 8 chiffres hexadécimaux chaque chaîne de 4 bits est convertie en son équivalent hexadécimal, comme décrit en (a) ci-dessus.
Exemple : 1010 0001 0000 0011 1111 1110 0010 0011 = A103FE23.

c. Un entier entre 0 et 2^32 - 1 inclus peut être représenté comme un mot. Les quatre bits de moindre poids de l'entier sont représentés par le chiffre hexadécimal le plus à droite de la représentation du mot.
Exemple : l'entier 291 = 2^8+2^5+2^1+2^0 = 256+32+2+1 est représenté par le mot hexadécimal, 00000123.

Si z est un entier, $0 \le z < 2^{64}$, alors $z = (2^{32})x + y$ où $0 \le x < 2^{32}$ et $0 \le y < 2^{32}$. Comme x et y peuvent être respectivement représentés comme les mots X et Y, z peut être représenté comme la paire de mots (X,Y).

d. Un bloc est une chaîne de 512 bits. Un bloc (par exemple, B) peut être représenté comme une séquence de 16 mots.

3. Opérations sur les mots

Les opérateurs logiques suivants sont appliqués aux mots :

a. Opérations de mot logiques au bit près

X ET Y = ajout logique au bit près de"X et Y.

X OU Y = "ou inclusif" logique au bit près sur X et Y.

X OUX Y = "ou exclusif" logique au bit près sur X et Y.

NON X = "complément" logique au bit près de X.

Exemple :

```
      01101100101110011101001001111011
OUX   01100101110000010110100110110111
      --------------------------------
  =   00001001011110001011101111001100
```

b. L'opération X + Y est définie ainsi : les mots X et Y représentent les entiers x et y, où $0 \le x < 2^{32}$ et $0 \le y < 2^{32}$. Pour les entiers positifs n et m, soit n mod m le reste de la division de n par m. On calcule $z = (x + y) \bmod 2^{32}$.

Alors $0 \le z < 2^{32}$. Convertir z en un mot, Z, et définir Z = X + Y.

c. L'opération de permutation circulaire à gauche S^n(X), où X est un mot et n est un entier avec $0 \leq n < 32$, est définie par

S^n(X) = (X << n) OU (X >> 32-n).

Ci-dessus, X << n est obtenu comme suit : éliminer les n bits de gauche de X puis bourrer le résultat avec n zéros sur la droite (le résultat fera encore 32 bits). X >> n est obtenu en éliminant les n bits de droite de X puis en bourrant le résultat avec n zéros sur la gauche. Donc S^n(X) est équivalent à une permutation circulaire de X de n positions vers la gauche.

4. Bourrage du message

SHA-1 est utilisé pour calculer un résumé de message pour un message ou fichier de données qui est fourni en entrée. Le message ou fichier de données devrait être considéré comme une chaîne binaire. La longueur du message est le nombre de bits dans le message (le message vide a une longueue 0). Si le nombre de bits dans un message est un multiple de 8, pour le rendre compact, on peut représenter le message en hexadécimal. L'objet du bourrage de message est de rendre la longueur totale d'un message bourré multiple de 512. SHA-1 traite en séquence les blocs de 512 bits lors du calcul du résumé du message. Ce qui suit spécifie comment devra être effectué ce bourrage. En résumé, un "1" suivi par des "0" suivis par un entier de 64 bits sont ajoutés à la fin du message pour produire un message bourré de longueur 512 * n. L'entier de 64 bits est la longueur du message d'origine. Le message bourré est alors traité par SHA-1 comme des blocs de 512 bits.

Supposons qu'un message a une longueur $l < 2^{64}$. Avant qu'il soit entré dans SHA-1, le message est bourré à droite comme suit :

a. "1" est ajouté. Exemple : si le message d'origine est "01010000", il devient avec le bourrage "010100001".

b. Les "0" sont ajoutés. Le nombre de "0" va dépendre de la longueur d'origine du message. Les derniers 64 bits du dernier bloc de 512 bits sont réservés pour la longueur *l* du message d'origine.

Exemple : Supposons que le message d'origine soit la chaîne binaire
01100001 01100010 01100011 01100100 01100101.

Après l'étape (a), cela donne 01100001 01100010 01100011 01100100 01100101 1.

Comme $l = 40$, le nombre de bits ci-dessus est 41 et 407 "0" sont ajoutés, faisant un total de 448. Cela donne (en hex)
61626364 65800000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000.

c. Obtenir la représentation de *l* en deux mots, le nombre de bits dans le message d'origine. Si $l < 2^{32}$ le premier mot est alors tout de zéros. Ajouter ces deux mots au message bourré.

Exemple : Supposons que le message d'origine est comme dans (b). Alors $l = 40$ (noter que *l* est calculé avant tout bourrage). La représentation en deux mots de 40 est l'hexadécimal 00000000 00000028. Donc, le message bourré final est en hexadécemal :
61626364 65800000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000028.

Le message bourré va contenir 16 * n mots pour toute n > 0. Le message bourré est considéré comme une séquence des n blocs M(1) , M(2), des premiers caractères (ou bits) du message.

5. Fonctions et constantes utilisées

Une séquence des fonctions logiques f(0), f(1),..., f(79) est utilisée dans SHA-1. Chaque f(t), $0 \leq t \leq 79$, opère sur trois mots B, C, D de 32 bits et produit un mot de 32 bits en résultat. f(t;B,C,D) est défini comme suit :

pour les mots B, C, D,

f(t;B,C,D) = (B ET C) OU ((NON B) ET D)	($0 \leq t \leq 19$)
f(t;B,C,D) = B OUX C OUX D	($20 \leq t \leq 39$)
f(t;B,C,D) = (B ET C) OU (B ET D) OU (C ET D)	($40 \leq t \leq 59$)
f(t;B,C,D) = B OUX C OUX D	($60 \leq t \leq 79$).

Une séquence de mots constants K(0), K(1), ... , K(79) est utilisée dans SHA-1. En hexadécimal, ils donnent :

K(t) = 5A827999 $0 \le t \le 19$)
K(t) = 6ED9EBA1 $20 \le t \le 39$)
K(t) = 8F1BBCDC $40 \le t \le 59$)
K(t) = CA62C1D6 $60 \le t \le 79$).

6. Calcul du résumé de message

Les méthodes indiquées en 6.1 et 6.2 ci-dessous donnent le même résumé de message. Bien que l'utilisation de la méthode 2 économise soixante quatre mots de 32 bits de mémoire, elle va vraisemblablement rallonger le temps d'exécution à cause de la complexité accrue des calculs d'adresse pour la { W[t] } à l'étape (c). Il y a d'autres méthodes de calcul qui donnent des résultats identiques.

6.1 Méthode 1

Le résumé de message est calculé en utilisant le message bourré comme décrit à la section 4. Le calcul est décrit avec l'utilisation de deux mémoires tampon, chacune consistant en cinq mots de 32 bits, et une séquence de quatre vingt mots de 32 bits. Les mots de la première mémoire tampon de cinq mots sont étiquetés A, B, C, D, E. Les mots de la seconde sont étiquetés H0, H1, H2, H3, H4. Les mots de la séquence de 80 mots sont étiquetés W(0), W(1),..., W(79). Une mémoire tampon TEMP d'un seul mot est aussi employée.

Pour générer le résumé de message, les blocs M(1), M(2),..., M(n) de 16 bits définis à la section 4 sont traités dans cet ordre. Le traitement de chaque M(i) implique 80 étapes.

Avant de traiter un bloc, les H sont initialisés comme suit, en hexadécimal :

H0 = 67452301
H1 = EFCDAB89
H2 = 98BADCFE
H3 = 10325476
H4 = C3D2E1F0.

Ensuite M(1), M(2), ... , M(n) sont traités. Pour traiter M(i), on procède comme suit :

a. Diviser M(i) en 16 mots W(0), W(1), ... , W(15), où W(0) est le mot le plus à gauche.

b. Pour $16 \le t \le 79$ soit W(t) = S^1(W(t-3) OUX W(t-8) OUX W(t-14) OUX W(t-16)).

c. Soit A = H0, B = H1, C = H2, D = H3, E = H4.

d. Pour $0 \le t \le 79$ faire TEMP = S^5(A) + f(t;B,C,D) + E + W(t) + K(t) ; E = D ; D = C ; C = S^30(B) ; B = A; A = TEMP ;

e. Soit H0 = H0 + A, H1 = H1 + B, H2 = H2 + C, H3 = H3 + D, H4 = H4 + E.

Après le traitement de M(n), le résumé de message est la chaîne de 160 bits représentée par les 5 mots H0 H1 H2 H3 H4.

6.2 Méthode 2

La méthode qui précède suppose que la séquence W(0), ... , W(79) est mise en œuvre comme une matrice de quatre vingt mots de 32 bits. Cela est efficace du point de vue de la minimisation du temps d'exécution, car les adresses de W(t-3), ... , W(t-16) dans l'étape (b) sont calculées facilement. Si l'espace est plus important, une autre solution est de considérer { W(t) } comme une file d'attente circulaire, qui peut être mise en œuvre en utilisant une matrice de seize mots de 32 bits W[0], ... W[15]. Dans ce cas, soit MASK = 0000000F en hexadécimal. On traite alors les M(i) comme suit :

a. Diviser M(i) en 16 mots W[0], ... , W[15], où W[0] est le mot le plus à gauche.

b. Soit A = H0, B = H1, C = H2, D = H3, E = H4.

c. Pour 0 ≤ t ≤ 79 faire s = t ET MASK ;
si (t ≥ 16) W[s] = S^1(W[(s + 13) ET MASK] OUX W[(s + 8) ET MASK] OUX W[(s + 2) ET MASK] OUX W[s]) ;
TEMP = S^5(A) + f(t;B,C,D) + E + W[s] + K(t) ;
E = D ; D = C ; C = S^30(B) ; B = A ; A = TEMP ;

d. Soit H0 = H0 + A, H1 = H1 + B, H2 = H2 + C, H3 = H3 + D, H4 = H4 + E.

7. Code C

Ci dessous figure une mise en œuvre de démonstration de SHA-1 en langage C. Le paragraphe 7.1 contient le fichier d'en-tête, le paragraphe 7.2 le code en langage C, et le paragraphe 7.3 un pilote d'essais.

7.1 Fichier .h

```
/*
*  sha1.h
*
* Description :
* Fichier d'en-tête pour le code qui met en œuvre l'algorithme 1 de hachage sécurisé tel que défini dans la norme FIPS
* PUB 180-1 publiée le 17 avril 1995. Beaucoup des noms de variables de ce code, en particulier les noms de caractères
* seuls, ont été utilisés parce que ces noms sont utilisés dans la norme publiée. Pour plus d'informations, prière de se
* reporter à la lecture du fichier sha1.c.
 */

#ifndef _SHA1_H_
#define _SHA1_H_

#include <stdint.h>
/*
* Si vous n'avez pas le fichier d'en-tête stdint.h de la norme ISO, vous devrez alors taper ce qui suit :
*  name                 signification
*  uint32_t             entier non signé de 32 bits
*  uint8_t              entier non signé de 8 bits (c'est-à-dire, un caractère non signé)
*  int_least16_t        entier ≥ 16 bits
*
*/

#ifndef _SHA_enum_
#define _SHA_enum_
enum
{
   shaSuccess = 0,
   shaNull,                    /* paramètre pointeur Nul */
   shaInputTooLong,            /* données d'entrée trop longues */
   shaStateError               /* appel d'entrée après résultat */
};
#endif
#define SHA1HashSize 20

/*
*  Cette structure va contenir les informations de contexte pour l'opération de hachage SHA-1
*/
typedef struct SHA1Context
{
   uint32_t Intermediate_Hash[SHA1HashSize/4];          /* Résumé de message */

   uint32_t Length_Low;                                 /* Longueur du message en bits */
   uint32_t Length_High;                                /* Longueur du message en bits */

/* Indice dans la matrice de bloc de message */
```

```
    int_least16_t Message_Block_Index;
    uint8_t Message_Block[64];                  /* Blocs de message de 512 bits */

    int Computed;                               /* Le résumé est-il calculé ? */
    int Corrupted;                              /* Le résumé de message est-il corrompu ? */
} SHA1Context;

/*
 * Prototypes de fonction
 */

int SHA1Reset(  SHA1Context *);
int SHA1Input(  SHA1Context *,
                const uint8_t *,
                unsigned int);
int SHA1Result( SHA1Context *,
                uint8_t Message_Digest[SHA1HashSize]);

#endif
```

7.2 Fichier .c

```
/*
*  sha1.c
*
* Description:
* Ce fichier met en œuvre l'algorithme 1 de hachage sécurisé tel que défini dans la norme FIPS PUB 180-1 publiée le
* 17 avril 1995. SHA-1 produit un résumé de message de 160 bits pour un certain flux de données. Il faudra environ
* 2**n étapes pour trouver un message avec le même résumé que celui de ce message et 2**(n/2) pour trouver deux
* messages avec le même résumé, lorsque n est la taille en bits du résumé. Cet algorithme peut donc servir de moyen pour
* donner une "empreinte digitale" d'un message.
*
* Questions de portabilité :
* SHA-1 est défini en termes de "mots" de 32 bits. Ce code utilise <stdint.h> (inclus dans "sha1.h" pour définir des types
* d'entier non signé de 32 et 8 bits. Si votre compilateur de langage C n'accepte pas les entiers non signés de 32 bits,
* ce code n'est pas approprié.
*
* Avertissements :
* SHA-1 est conçu pour fonctionner avec des messages de moins de 2^64 bits. Bien que SHA-1 permette que soit généré
* un résumé de message pour des messages de tout nombre de bits inférieur à 2^64 bits, cette mise en œuvre ne
* fonctionne qu'avec les messages d'une longueur multiple de la taille d'un caractère de 8 bits.
*
*/

#include "sha1.h"

/*
* Definit la macro SHA1 permutation circulaire à gauche
*/
#define SHA1CircularShift(bits,word) \ (((word) << (bits)) | ((word) >> (32-(bits))))

/* Prototyptes de fonction locale */
void SHA1PadMessage(SHA1Context *);
void SHA1ProcessMessageBlock(SHA1Context *);

/*
*  SHA1Reset
*
* Description :
* Cette fonction va initialiser le SHA1Context pour préparer le calcul d'un nouveau résumé de message SHA1.
*
```

```
*  Paramètres : context: [in/out]     /* C'est le contexte à réinitialiser. */
*
*  Résultat : Code d'erreur sha.
*
*/

int SHA1Reset(SHA1Context *context)
{
   if (!context)
   {
      return shaNull;
   }

   context->Length_Low             = 0;
   context->Length_High            = 0;
   context->Message_Block_Index    = 0;

   context->Intermediate_Hash[0]   = 0x67452301;
   context->Intermediate_Hash[1]   = 0xEFCDAB89;
   context->Intermediate_Hash[2]   = 0x98BADCFE;
   context->Intermediate_Hash[3]   = 0x10325476;
   context->Intermediate_Hash[4]   = 0xC3D2E1F0;

   context->Computed   = 0;
   context->Corrupted  = 0;

   return shaSuccess;
}

/*
*  SHA1Result
*
* Description:
* Cette fonction va retourner le résumé de message de 160 bits dans la matrice Message_Digest fournie par l'appelant.
* Noter que le premier octet du hachage est mémorisé dans l'élément 0, le dernier octet du hachage dans l'élément 19.
*
* Paramètres :
*      context: [in/out]                *  C'est le contexte à utiliser pour calculer le hachage SHA-1.
*      Message_Digest: [out]            *  Où le résumé sera renvoyé.
*
* Résultat : Code d'erreur sha.
*
*/

int SHA1Result( SHA1Context *context, uint8_t Message_Digest[SHA1HashSize])
{
   int i;

   if (!context || !Message_Digest)
   {
      return shaNull;
   }

   if (context->Corrupted)
   {
      return context->Corrupted;
   }

   if (!context->Computed)
   {
      SHA1PadMessage(context);
      for(i=0; i<64; ++i)
      {
```

```
                                                    /* le message peut être sensible, l'éliminer */
            context->Message_Block[i] = 0;
        }
        context->Length_Low = 0;                    /* et éliminer la longueur */
        context->Length_High = 0;
        context->Computed = 1;

    }

    for(i = 0; i < SHA1HashSize; ++i)
    {
        Message_Digest[i] = context->Intermediate_Hash[i>>2] >> 8 * ( 3 - ( i & 0x03 ) );
    }

    return shaSuccess;
}

/*
* SHA1Input
*
* Description :
* Cette fonction accepte une matrice d'octets comme portion suivante du message.
*
* Paramètres :
*     context: [in/out]        * C'est le contexte SHA à mettre à jour *
*     message_array: [in]   * Matrice de caractères qui représentent la prochaine portion du message. *
*     length: [in]              * Longueur du message dans message_array
*
* Retours :                    * Code d'erreur sha.*
*/

int SHA1Input(    SHA1Context    *context,
                  const uint8_t  *message_array,
                  unsigned       length)
{
    if (!length)
    {
        return shaSuccess;
    }

    if (!context || !message_array)
    {
        return shaNull;
    }

    if (context->Computed)
    {
        context->Corrupted = shaStateError;

        return shaStateError;
    }

    if (context->Corrupted)
    {
         return context->Corrupted;
    }
    while(length-- && !context->Corrupted)
    {
    context->Message_Block[context->Message_Block_Index++] =
                (*message_array & 0xFF);

    context->Length_Low += 8;
    if (context->Length_Low == 0)
```

```
    {
      context->Length_High++;
      if (context->Length_High == 0)
      {
        /* Message is too long */
        context->Corrupted = 1;
      }
    }

    if (context->Message_Block_Index == 64)
    {
      SHA1ProcessMessageBlock(context);
    }

    message_array++;
    }

    return shaSuccess;
}

/*
*  SHA1ProcessMessageBlock
*
* Description :
* Cette fonction va traiter les 512 prochains bits du message mémorisés dans la matrice Message_Block.
*
* Paramètres :      * Aucun.
*
* Retour :          * Rien.
*
* Commentaires :
* Beaucoup des noms de variables de ce code, en particulier ceux des noms de caractères seuls, ont été utilisés parce que
* ce sont ceux qui sont utilisés dans la norme publiée.*
*/

void SHA1ProcessMessageBlock(SHA1Context *context)
{
    const uint32_t K[] =    {                       /* Constantes définies dans SHA-1   */
                        0x5A827999,
                        0x6ED9EBA1,
                        0x8F1BBCDC,
                        0xCA62C1D6
                        };
    int         t;                                  /* compteur de boucle      */
    uint32_t    temp;                               /* valeur de mot temporaire */
    uint32_t    W[80];                              /* séquence  de mot */
    uint32_t    A, B, C, D, E;                      /* mémoires tampons de mots */

/*
* Initialise les 16 premiers mots dans la matrice W
*/
    for(t = 0; t < 16; t++)
    {
        W[t] = (uint32_t)(context->Message_Block[t * 4]) << 24;
        W[t] |= (uint32_t)(context->Message_Block[t * 4 + 1]) << 16;
        W[t] |= context->Message_Block[t * 4 + 2] << 8;
        W[t] |= context->Message_Block[t * 4 + 3];
    }

    for(t = 16; t < 80; t++)
    {
       W[t] = SHA1CircularShift(1,W[t-3] ^ W[t-8] ^ W[t-14] ^ W[t-16]);
    }
```

```
    A = context->Intermediate_Hash[0];
    B = context->Intermediate_Hash[1];
    C = context->Intermediate_Hash[2];
    D = context->Intermediate_Hash[3];
    E = context->Intermediate_Hash[4];

    for(t = 0; t < 20; t++)
    {
        temp =  SHA1CircularShift(5,A) + ((B & C) | ((~B) & D)) + E + W[t] + K[0];
        E = D;
        D = C;
        C = SHA1CircularShift(30,B);

        B = A;
        A = temp;
    }

    for(t = 20; t < 40; t++)
    {
        temp = SHA1CircularShift(5,A) + (B ^ C ^ D) + E + W[t] + K[1];
        E = D;
        D = C;
        C = SHA1CircularShift(30,B);
        B = A;
        A = temp;
    }

    for(t = 40; t < 60; t++)
    {
        temp = SHA1CircularShift(5,A) + ((B & C) | (B & D) | (C & D)) + E + W[t] + K[2];
        E = D;
        D = C;
        C = SHA1CircularShift(30,B);
        B = A;
        A = temp;
    }

    for(t = 60; t < 80; t++)
    {
        temp = SHA1CircularShift(5,A) + (B ^ C ^ D) + E + W[t] + K[3];
        E = D;
        D = C;
        C = SHA1CircularShift(30,B);
        B = A;
        A = temp;
    }

    context->Intermediate_Hash[0] += A;
    context->Intermediate_Hash[1] += B;
    context->Intermediate_Hash[2] += C;
    context->Intermediate_Hash[3] += D;
    context->Intermediate_Hash[4] += E;
    context->Message_Block_Index = 0;
}


/*
 *  SHA1PadMessage
 *

* Description:
* Conformément à la norme, le message doit être bourré pour faire 512 bits. Le premier bit de bourrage doit être un '1'.
```

```
* Les 64 derniers bits représentent la longueur du message d'origine. Tous les bits entre devraient être à 0.
* Cette fonction bourre le message conformément à ces règles en remplissant en conséquence la matrice Message_Block.
* Elle va aussi invoquer la fonction ProcessMessageBlock fournie de façon appropriée.
* Quand elle a fini, on peut supposer que le résumé de message a été calculé.
*
* Paramètres :
*     context: [in/out]              Le contexte à bourrer
*     ProcessMessageBlock: [in]      C'est la fonction SHA*ProcessMessageBlock appropriée
* Retour :     Rien.
*
*/

void SHA1PadMessage(SHA1Context *context)
{
/*
* Vérifier si le bloc de message actuel est trop petit pour contenir les bits et la longueur du bourrage initial.
* Si oui, on va bourrer le bloc, le traiter, puis continuer le bourrage dans un second bloc.
*/

   if (context->Message_Block_Index > 55)
   {
      context->Message_Block[context->Message_Block_Index++] = 0x80;
      while(context->Message_Block_Index < 64)
      {
         context->Message_Block[context->Message_Block_Index++] = 0;
      }

      SHA1ProcessMessageBlock(context);

      while(context->Message_Block_Index < 56)
      {
         context->Message_Block[context->Message_Block_Index++] = 0;
      }
   }
   else
   {
      context->Message_Block[context->Message_Block_Index++] = 0x80;
      while(context->Message_Block_Index < 56)
      {

         context->Message_Block[context->Message_Block_Index++] = 0;
      }
   }

/*
* Mémorise la longueur de message sous les 8 derniers octets
*/
   context->Message_Block[56] = context->Length_High >> 24;
   context->Message_Block[57] = context->Length_High >> 16;
   context->Message_Block[58] = context->Length_High >> 8;
   context->Message_Block[59] = context->Length_High;
   context->Message_Block[60] = context->Length_Low >> 24;
   context->Message_Block[61] = context->Length_Low >> 16;
   context->Message_Block[62] = context->Length_Low >> 8;
   context->Message_Block[63] = context->Length_Low;

   SHA1ProcessMessageBlock(context);
}
```

7.3 Pilote d'essais

Le code qui suit est un pilote d'essai de programme principal pour vérifier le code en sha1.c.

```
/*
*  sha1test.c
*
* Description :
* Le présent fichier va vérifier le code SHA-1 qui effectue les trois essais documentés dans FIPS PUB 180-1 plus un qui
* invoque SHA1Input avec un multiple exact de 512 bits, plus quelques vérifications d'essai d'erreurs.
*
* Problèmes de portabilité :  Aucun.
*
*/

#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include "sha1.h"

/*
* Définition des schémas d'essai
*/
#define TEST1   "abc"
#define TEST2a  "abcdbcdecdefdefgefghfghighijhi"

#define TEST2b  "jkijkljklmklmnlmnomnopnopq"
#define TEST2   TEST2a TEST2b
#define TEST3   "a"
#define TEST4a  "01234567012345670123456701234567"
#define TEST4b  "01234567012345670123456701234567"              /* un multiple exact de 512 bits */
#define TEST4   TEST4a TEST4b
char *testarray[4] =
{
   TEST1,
   TEST2,
   TEST3,
   TEST4
};
long int repeatcount[4] = { 1, 1, 1000000, 10 };
char *resultarray[4] =
{
   "A9 99 3E 36 47 06 81 6A BA 3E 25 71 78 50 C2 6C 9C D0 D8 9D",
   "84 98 3E 44 1C 3B D2 6E BA AE 4A A1 F9 51 29 E5 E5 46 70 F1",
   "34 AA 97 3C D4 C4 DA A4 F6 1E EB 2B DB AD 27 31 65 34 01 6F",
   "DE A3 56 A2 CD DD 90 C7 A7 EC ED C5 EB B5 63 93 4F 46 04 52"
};

int main()
{
   SHA1Context sha;
   int i, j, err;
   uint8_t Message_Digest[20];

/*
* Effectue les essais SHA-1
*/

   for(j = 0; j < 4; ++j)
   {
      printf( "\nTest %d: %d, '%s'\n",
            j+1,
            repeatcount[j],
            testarray[j]);

      err = SHA1Reset(&sha);
      if (err)
```

```
        {
            fprintf(stderr, "SHA1Reset Error %d.\n", err );
            break;                                  /* sortie pour la boucle j */
        }

        for(i = 0; i < repeatcount[j]; ++i)
        {

            err = SHA1Input(&sha,
                (const unsigned char *) testarray[j],
                strlen(testarray[j]));
            if (err)
            {
                fprintf(stderr, "SHA1Input Error %d.\n", err );
                break;                                          /* sortie pour la boucle i */
            }
        }

        err = SHA1Result(&sha, Message_Digest);
        if (err)
        {
            fprintf(stderr,
            "SHA1Result Error %d, could not compute message digest.\n",
            err );
        }
        else
        {
            printf("\t");
            for(i = 0; i < 20 ; ++i)
            {
                printf("%02X ", Message_Digest[i]);
            }
            printf("\n");
        }
        printf("Should match:\n");
        printf("\t%s\n", resultarray[j]);
    }

/* Vérifie des retours d'erreur */
    err = SHA1Input(&sha,(const unsigned char *) testarray[1], 1);
    printf ("\nError %d. Should be %d.\n", err, shaStateError );
    err = SHA1Reset(0);
    printf ("\nError %d. Should be %d.\n", err, shaNull );
    return 0;
}
```

8. Considérations pour la sécurité

Le présent document est destiné à fournir à la communauté de l'Internet un accès pratique et gratuit à la norme fédérale de traitement de l'information des États-Unis d'Amérique de la fonction de hachage sécurisé SHA-1 [FIPS 180-1]. L'auteur n'a pas l'intention de faire d'assertion personnelle sur la sécurité de cette fonction de hachage pour une utilisation particulière.

Références

[FIPS 180-1] "Secure Hash Standard", United States of America, National Institute of Science and Technology, Federal Information Processing Standard (FIPS) 180-1, avril 1993.

[MD4] "The MD4 Message Digest Algorithm," Advances in Cryptology - CRYPTO '90 Proceedings, Springer-Verlag, 1991, pp. 303-311.

[RFC1320] R. Rivest, "Algorithme de résumé de message MD4", avril 1992. *(Historique, Information)*

[RFC1321] R. Rivest, "Algorithme de résumé de message MD5", avril 1992. *(Information)*

[RFC1750] D. Eastlake, 3rd et autres, "Recommandations d'aléa pour la sécurité", décembre 1994. *(Info., remplacée par la RFC 4086)*

Adresses des auteurs

Donald E. Eastlake, 3rd
Motorola
155 Beaver Street
Milford, MA 01757 USA
téléphone : +1 508-634-2066 (domicile)
+1 508-261-5434 (bureau)
mél : Donald.Eastlake@motorola.com

Paul E. Jones
Cisco Systems, Inc.
7025 Kit Creek Road
Research Triangle Park, NC 27709 USA
téléphone : +1 919 392 6948
mél : paulej@packetizer.com

Déclaration de droits de reproduction

Remerciement

Le financement de la fonction d'édition des RFC est actuellement fourni par l'Internet Society.