Vishash : une fonction de hachage qui génère des images facilement reconnaissables et comparables

Lorsque l'on télécharge, transfère, ou tout simplement manipule des fichiers, il est parfois nécessaire de pouvoir vérifier l'intégrité de ces fichiers. Usuellement, on utilise pour cela des fonctions spéciales qui permettent de calculer **l'empreinte** du fichier. Cette empreinte consiste souvent en une chaîne de caractères relativement courte. Ainsi pour comparer deux fichiers, on calcule l'empreinte des deux fichiers, et si elles sont différentes, alors les fichiers sont différents. Une bonne fonction d'empreinte assure aussi que si les empreintes sont identiques, alors les fichiers le sont probablement aussi. Cependant, l'espace des possibilités rend impossible d'assurer cette propriété sûrement. La propriété qu'on demande alors est qu'il soit difficile pour un fichier donné de trouver une fichier différent ayant la même empreinte. De telles fonctions de hachage peuvent également être utilisées pour des applications cryptographiques. Dans ce cas, il est également intéressant que pour un hash donné, il soit impossible de retrouver l'entrée originale.

Vishash est une fonction de hachage de fichier, c'est-à-dire une fonction permettant de calculer l'empreinte de n'importe quel fichier. Mais contrairement aux fonctions classiques qui calculent des chaînes de bits (ou de caractères), Vishash calcule une image. Cette particularité possède l'avantage de générer une empreinte visuelle facilement reconnaissable et comparable, contrairement à une chaîne de de 64 caractères alphanumériques comme renvoyé par l'algorithme SHA3.

Ces avantages seraient encore plus intéressants s'il s'avérait que Vishash possédait les propriétés d'une fonction de hash cryptographique comme décrit plus haut.

Avant de mesurer les propriétés de l'algorithme, présentons en détail son fonctionnement

Fonctionnement général

L'algorithme Vishash prend en argument une chaîne quelconque d'octets, ainsi que des paramètres associés aux dimensions de l'image à générer (`width`, `height`), et une constante `K` permettant de contrôler la taille des motifs sur l'image finale.

Voici les deux étapes principales de l'algorithme :

1. Générer une image pseudo-aléatoire à partir des données du fichier
2. Traiter cette image jusqu'à obtenir l'image finale

Partie 1 : Génération de l'image pseudo-aléatoire

L'objectif de cette partie est d'obtenir à partir de données quelquonques et de taille quelconque, une suite de pixels (donc de 3 octets) pseudo-aléatoire nous permettant de remplir l'image originale.

Afin de se protéger contre des fichiers contenant toujours la même donnée, très peu d'informations, ou des informations mal distribuées dans l'espace des possibles, nous devons traiter les données afin de les rendre pseudo-aléatoires.

Pour cela, au lieu de directement remplir l'image avec les données du fichier, on va combiner chaque octet du fichier original avec un nombre aléatoire en effectuant un xor (OU exclusif) binaire.

Les nombres aléatoires sont données par une suite pseudo-aléatoire initialisée sur une seed, elle-même calculée à partir du fichier original. Les détails de la suite pseudo-aléatoire seront donnés plus tard.

Voici comment on calcule la couleur d'un pixel à partir d'un entier aléatoire sur 4 octets et d'un octet du fichier :

```
IntPixel combine(uint8_t byte, uint32_t random_number) {
    IntPixel pix;
    pix.r = byte ^ random_number;
    pix.g = byte ^ (random_number >> 8);
    pix.b = byte ^ (random_number >> 16);
    return pix;
}
```

Voici la fonction remplissant l'image avec ces pixels :

```
IntImage load_image(const char* filename, int width, int height) {
    IntImage img = create_empty_int_image(width, height);
    int image_size = img.height * img.width;

    long int file_size;
    uint8_t* file_data = load_file_data(filename, &file_size);

    if (file_data == NULL) {
        return (IntImage){0};
    }

    init_random_number(file_data, file_size);

    // on s'assure que l'image sera remplie et qu'on parcourt tout le
fichier
    int nb_iterations = max(image_size, file_size);

    for (int i = 0 ; i < nb_iterations ; i++) {
        img.colors[i%image_size] = combine(file_data[i%file_size],
next_random_number());
    }

    free(file_data);
    return img;
}
```

Comme vous le voyez, on parcourt toujours systématiquement le fichier en entier, et s'il y a plus de pixels dans l'image que d'octets dans le fichier, on parcourt de nouveau le fichier.

La suite pseudo-aléatoire

La suite pseudo-aléatoire utilisée est très simple, et chaque nombre est calculé comme suit :

```
uint32_t next_random_number(void) {
    random_number = (random_number * A + C) % M;
    return random_number;
}
```

avec les constantes suivantes :

```
#define A 16807
#define C 42
#define M (((uint32_t)1 << 31) - 1)
```

Comme dit précédemment, cette suite pseudo-aléatoire est calculée à partir d'une seed générée grâce aux données du fichier d'entrée. Voici l'algorithme qui calcule le nombre aléatoire initial :

```
void init_random_number(uint8_t* file_data, long int size) {
    random_number = 3;
    for (long int i = 0 ; i < size ; i++) {
        random_number = (random_number * A + file_data[i]) % M;
    }
}
```

Cet algorithme utilise les mêmes constantes que celles données plus haut.

Partie 2 : Traitement de l'image pseudo-aléatoire jusqu'à image finale

L'algorithme de traitement de l'image peut être considéré comme un algorithme de diffusion : il fonctionne par étapes de débruitage successives.

Voici la fonction réalisant l'algorithme (en C simplifié) :

```
void make_iterations(FloatImage* img, int taille, int K, int n) {
    for (int i = 1 ; i < n ; i++) {
        double sigma = (double)(i*taille)/(double)K;
        size = kernel_size(sigma);
        img = gaussian_blur(img, size, sigma);

        img *= 2 + 5*(n-i)/n;
        remove_zero(img);
    }
}
```

L'algorithme consiste à itérativement flouter l'image, puis multiplier chaque couleur par une certaine constante. Au fur et à mesure des itérations, le flou effectué est de plus en plus agressif, et la constante multiplicative diminue.

Les formules ont été déterminées ad-hoc, dans le but de garantir que la constante K permette de contrôler la taille des motifs *relativement à l'image*.