# Cahier des charges :

Il faut remplacer le nom pris par les devices d’un plugin qui sont simplement une incrémentation en fonction du moment où on les appelle (ex : Device declared [DeviceId = (1,1)] ).

À la place il faut générer un nom qui sera toujours le même pour un device, qui soit généré en fonction de caractéristique propre au device (ex : le nom) et qui soit homogène qu’importe le device.

Pour cela la technique dite du hachage a été choisie.

# Le hachage en général

Une fonction de hachage permet à une donnée de taille arbitraire d’en créer une donnée de taille fixe. Comme on le voit sur la figure (Figure 1) qu’importe la longueur de l’entrée, la fonction de hachage la transforme en une sortie d’information de même taille, c’est-à-dire homogène.

:

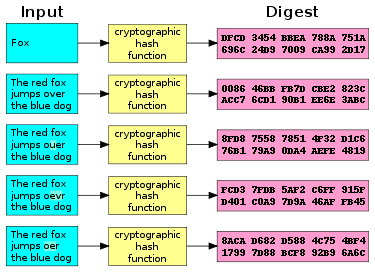


Figure 1 : Hachage

## Quelques termes importants

Les collisions : C’est la probabilité que deux données d’entrée différente, une fois hacher, donnent la même donnée de sortie. Ce qui est à éviter.

L’effet d’avalanche : C’est l’effet qui permet que le changement même d’un bit sur l’information d’entrée du hachage va grandement modifier l’information de hachage de sortie. C’est un paramètre important dans une fonction de hachage (surtout pour celle de cryptographie).

Ex : Signal => 2329670123

SignaL => 1586212811

Ici on peut observer un fort effet d’avalanche puisque les informations de sortie sont très différentes.

## L’utilisation

La fonction de hachage a principalement deux utilisations :

Soit on l’utilise pour hacher et crypter des informations, par exemple un login et un mot de passe sur un site web. Le site web va stocker le hachage de ces deux informations et quand on voudra se connecter à ce site web, le site va comparer le hachage du login et du mot de passe que l’on vient de rentrer avec tous ceux contenus dans sa base de données. Dans ce cas-là il est important qu’il soit impossible de retrouver les informations d’entrées grâce à l’information de sortie du hachage.

Soit on utilise cette fonction pour faire un tableau de hachage. Cela consiste à hacher une information « importante » (exemple : Mr. Martin) d’un tableau, de la moduler jusqu’à une petite valeur est de la mettre dans la case du tableau correspondante, pour Mr.Martin ça pourrait être la case 27 du tableau par exemple.

Cette méthode permet de retrouver rapidement une ligne dans un tableau quand on a comme donnée seulement cette information « importante ». On refait le hachage et la modulation sur cette donnée et on retrouve directement la ligne du tableau correspondant avec toutes les informations qui nous intéresse, ça évite de scruter toutes les cases de notre tableau.

Dans notre cas on va plus s’orienter vers une fonction de hachage non sécurisé (pour les tableaux de hachages) pour plusieurs raisons :

* Elles sont plus simples et donc plus rapides à exécuter.
* Nous n’avons pas besoin, dans notre cas, d’une fonction de hachage qui rend impossible le fait de retrouver la donnée d’entrée grâce à la donnée de sortie (hachage cryptage)
* Il est possible d’en trouver dans le domaine public alors que celle de hachage cryptage non.
* Elles ont quand même une très faible chance de collision

## Les différents hachages pouvant nous intéresser

Fonctions de hachage non cryptographiques

|  |  |  |
| --- | --- | --- |
| **Nom** | **Taille** | **Type** |
| Pearson hashing | 8 bits | xor/table |
| Paul Hsieh's SuperFastHash[[1]](https://fr.wikipedia.org/wiki/Liste_de_fonctions_de_hachage#cite_note-1) | 32 bits |  |
| Buzhash | variable | xor/table |
| Fowler–Noll–Vo hash function (FNV Hash) | 32, 64, 128, 256, 512, or 1024 bits | xor/product or product/xor |
| Jenkins hash function | 32 or 64 bits | xor/addition |
| Java hashCode() | 32 bits |  |
| Bernstein hash *djb2*[[2]](https://fr.wikipedia.org/wiki/Liste_de_fonctions_de_hachage#cite_note-2) | 32 bits |  |
| PJW hash / Elf Hash | 32, 64 bits | hash |
| MurmurHash | 32, 64, or 128 bits | product/rotation |
| SpookyHash | 32, 64 or 128 bits | Cf Jenkins hash function |
| CityHash | 64, 128, or 256 bits |  |
| numeric hash (nhash)[[3]](https://fr.wikipedia.org/wiki/Liste_de_fonctions_de_hachage#cite_note-3) | variable | Division/Modulo |
| xxHash[[4]](https://fr.wikipedia.org/wiki/Liste_de_fonctions_de_hachage#cite_note-4) | 32, 64 bits |  |

Les fonctions de hachage qui sont dans le domaine publique sont : FNV, Murmur Hash et Spooky Hash.

D’un point de vue simplicité le hachage FNV est le meilleur et donc le plus rapide.

# Le hachage Fowler–Noll–Vo hash function

Algorithme:

**hash** = ***offset\_basis***

for each *octet\_of\_data* to be hashed

**hash** = **hash** xor *octet\_of\_data*

**hash** = **hash** \* ***FNV\_prime***

return **hash**

L’offset de départ du hachage est fixe et est donnée par ce site web (<http://www.isthe.com/chongo/tech/comp/fnv/>) ce qui donne 14695981039346656037.

Le FNV\_prime est la clé de hachage, elle est très importante et c’est ça taille qui permet de calculer la probabilité de collision.

Nous allons partir pour une clé de hachage de 64 bits ce qui donne ce nombre pour le FNV prime, 1099511628211.

Code en C++ :

string str = "Signal";

const unsigned int fnv\_prime = 0x1099511628211;

unsigned int hash = 14695981039346656037;

unsigned int i = 0;

unsigned int len = str.length();

for (i = 0; i < len; i++)

{

hash ^= (str[i]);

hash \*= fnv\_prime;

}

cout << hash << endl;

system("pause");

return 0;

## Chance de collision de la fonction de hachage FNV

On part du postulat qu’une fonction de hachage va générer une suite de chiffre de telle manière que l’on soit près du hasard. Cela veut dire que ce qui va influencer sur les chances de collision est la clé de hachage, dans notre cas elle est de 64 bits. En se référant à la Figure 2 et en supposant qu’il n’y aura pas plus de 6000 devices et donc pas plus de 6000 hachages, cela fait une probabilité de collision de moins de 1 sur un trillion (1/(109x109 )).

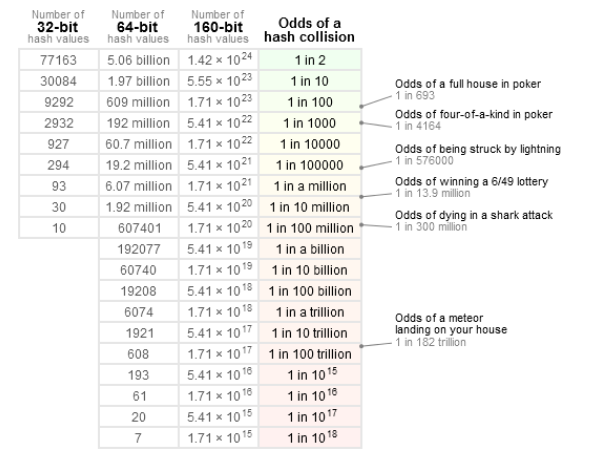


Figure 2: Probabilité de collision