# Funciones de Hashing

En términos generales una función de hashing es de la forma:

**f(s)=h**

Donde:

f, es la función de hashing.
s, es una tira de bytes de una cierta longitud, a s se le puede llamar también "clave" aunque no es cierto que s sea siempre la clave de algo.
h, es el valor devuelto por la función de hashing.

La longitud de s determina el "**espacio de claves**"
La longitud de h determina el "**espacio de direcciones**"

**Colisiones y Sinónimos.**

En la mayoría de los casos el espacio de direcciones es mucho mas chico que el espacio de claves por lo que dos o mas claves pueden ser asignadas a un mismo espacio de direcciones, dos claves que dan como resultado el mismo h se denominan sinónimos.

Sinónimos:

s1 y s2 son sinónimos sii: f(s1)=f(s2)

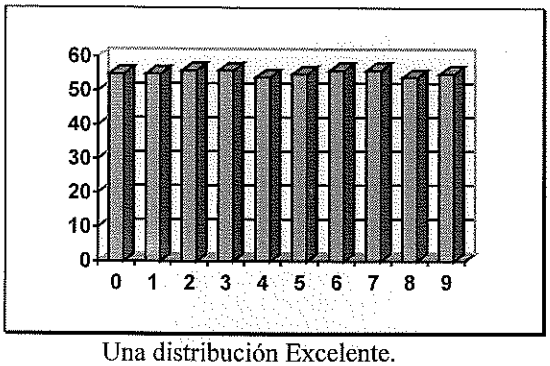
Cuando el resultado de f(s) da como resultado un resultado que ya había sido obtenido al hashear otro s' (sinónimo) se dice que se produce una colisión. Sinónimos y colisiones son términos fuertemente vinculados ya que los sinónimos producen colisiones.

Una buena función de hashing es aquella que para un cierto conjunto de datos minimiza la cantidad de colisiones que se producen.

**Distribución de la función de hashing.**

La distribución de la función se obtiene al observar los valores devueltos por la función para un cierto espacio muestral de datos. La distribución puede analizarse estadísticamente o incluso graficarse para evaluar el rendimiento de la función. Para que una función de hashing produzca la menor cantidad de colisiones posibles la distribución debe ser equilibrada En el caso ideal cada valor posible del espacio de direcciones tiene igual numero de ocurrencias.

Ejemplo.




Una distribución Excelente.

Si el espacio de direcciones es el 0..9 la función de arriba presenta una distribución muy buena.

Cuanto mas chico sea el espacio de direcciones con respecto al espacio de claves mayor numero de colisiones producirá la función de hashing, por lo que la forma en la cual se distribuirán dichas colisiones se convierte a veces en un factor critico.

Algunas funciones de hashing funcionan únicamente para claves de longitud fija, otras en cambio trabajan con claves de longitud variable una función de hashing para claves desde 2 bytes hasta 1Gb debe estar diseñada con mucho mayor cuidado que funciones que por ejemplo solo sirven para claves de 8 bytes. Las funciones que trabajan con claves de longitud variable son mucho mas versátiles que las que no lo hacen ya que permiten hashear todo tipo de información, desde claves de registros de poca longitud hasta archivos enteros.

Una función de hashing puede aplicarse en un gran numero de situaciones:

- Para obtener la dirección tentativa de un registro de un archivo directo (se hashea la clave)
- Para transformar un string en otro de menor longitud que represente al primero.
- Para ubicar datos en archivos o en memoria implementando una tabla de acceso directo.
- Para obtener un numero o string que represente a un archivo (se hashea el archivo), esto se usa en firmas digitales, autenticación de archivos etc.
- Para obtener una representación numérica de un dato que no lo es.
- etc.

## Funciones de hashing perfectas.

Una función de hashing perfecta es aquella que nunca produce colisiones. También se las denomina funciones 1 a 1. En general estas funciones son necesarias cuando se quiere obtener una representación numérica de una cierta información de forma tal que el numero represente en forma unívoca a la información hasheada.

Ejemplo:

En el juego del loto se tienen que elegir 6 (seis) números distintos entre 0 y 41. Existen en total 5.245.786 combinaciones posibles a jugar. Un problema es como convertir algo de la forma n1,n2,n3,n4,n5,n6 en un numero de 0 a 5.245.785. Por ejemplo ¿como convertir 00,01,02,03,04,15 en el numero 10?. Una solución trivial consistiría en concatenar todos los números y luego tomar la representación binaria de los números concatenados como resultado de la función. Esta solución falla, ya que si bien el numero generado es único (no habrá dos iguales) el espacio necesario es de 6*8=48 bits y 2^48 es mucho mayor a 5.245.785 que son todas las combinaciones posibles. En este caso y en muchos otros la solución pasa por construir una función de hashing 1 a 1.

La construcción de funciones de hashing perfectas puede ser a veces una tarea muy difícil, en los casos en los cuales no logra encontrarse la técnica adecuada para construir la función puede recurrirse a una tabla de doble entrada que represente la función, en estos casos se habla de hashing por tablas, ya que la función esta representada por una tabla con todos los valores de entrada posibles y los valores de salida que le corresponden. La cantidad de entradas de esta tabla será la longitud del espacio de direcciones.

Para nuestro caso de los números del loto no hace falta la tabla ya que la función puede construirse en forma relativamente sencilla, esto se deja como ejercicio para el lector.

Donald Knuth en "The Art of Computer Programming Vol3" menciona una funcion de hashing perfecta para 31 palabras del idioma inglés elegidas arbitrariamente, luego de varios calculos estrambóticos sobre los caracteres se llega a un resultado único para cada palabra. Knuth señala que funciones perfectas pueden encontrarse para conjuntos de claves relativamente pequeños sin utilizar tablas en aproximadamente un día de trabajo. Para conjuntos mas y mas grandes la tarea se hace imposible ya que en general existen demasiadas funciones posibles a efectuar sobre las claves y son muy pocas aquellas que dan resultados unicos para cada clave. En general se establece que una de cada 10 millones de funciones es perfecta.

(*) Un método nuevo sumamente interesante para construcción de funciones de hashing perfectas se describe en forma completa en el apunte sobre sistemas de recuperación total de textos.

## Funciones de hashing no-perfectas.

Las funciones de hashing no-perfectas son aquellas que pueden producir colisiones, este tipo de funciones son las mas usadas en la mayoría de las aplicaciones en las que tiene que usarse una función de hashing.

Las técnicas usadas para construir funciones de hashing son diversas, muchas funciones de hashing procesan bloques de una cierta longitud de bits y devuelven un valor de otra longitud de bits. Este valor debe en muchos casos adaptarse al espacio de direcciones con el cual se trabaje, en general para este paso se usa la función mod.

| |
|---|
| Esquema de hashing usando una función genérica y mod.<br><br>h=f(s) mod k. |

## La función MOD.

La función a mod b devuelve el resto de la división entera entre a y b. Por ejemplo 8 mod 3 devuelve 2 ya que el resto de 8/3 es 2. La expresión a mod b devuelve siempre un numero comprendido entre 0 y b-1, por lo que si el espacio de direcciones es k, basta con aplicarle a un numero x la función x mod k para obtener un valor que pertenezca al espacio de direcciones.

La función MOD se usa en casi todas las aplicaciones que usen una función de hashing, o bien como parte de la función de hashing o bien para mapear el resultado de la función al espacio de direcciones correspondiente. El único dato a considerar al usar la función MOD es el numero sobre el cual se efectúa la división.

¿Dado un x cual es el mejor y para hacer x mod y?

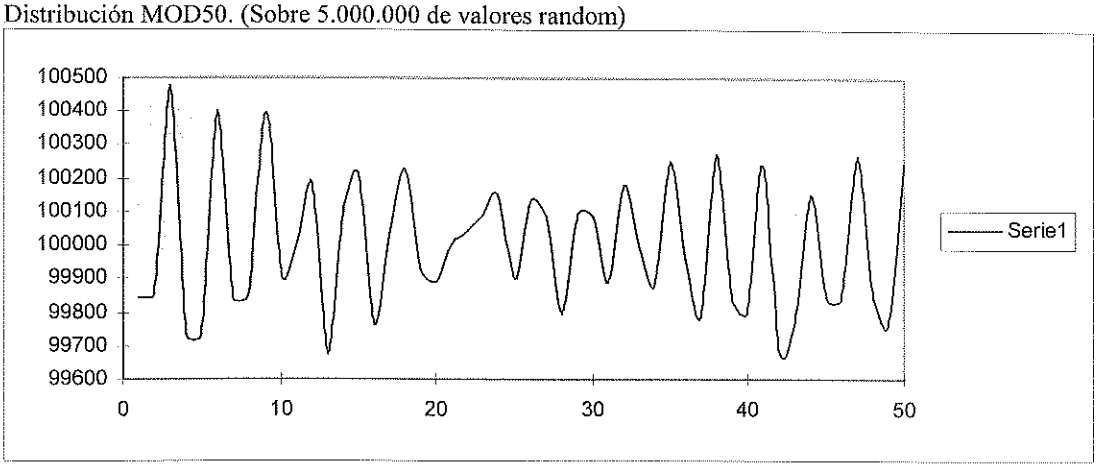
Numerosos estudios han llegado a la conclusión de que los restos de las divisiones enteras se distribuyen en forma mas uniforme si se utiliza como divisor un numero primo, por lo que en casi todos los casos se busca que el numero y sea primo.

La elección de y suele hacerse tomando el primer numero primo que se encuentre mayor al espacio de direcciones, en general esto implica una pequeña corrección sobre el espacio de direcciones a utilizar.

A continuación mostramos tests realizados para la función mod sobre un total de 5.000.000 de claves aleatorias de 16 bytes de longitud.

**Distribución de la función MOD.**

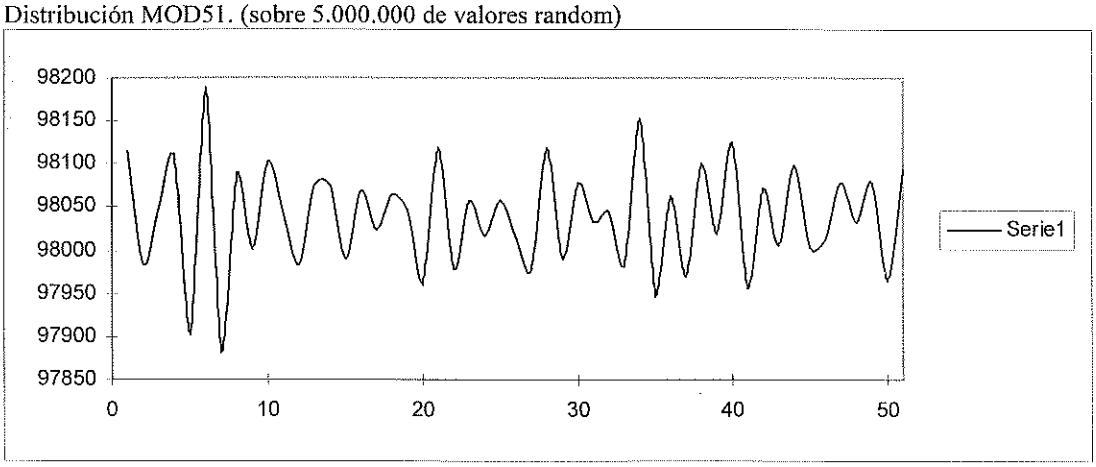
Distribución MOD50. (Sobre 5.000.000 de valores random)




Análisis:

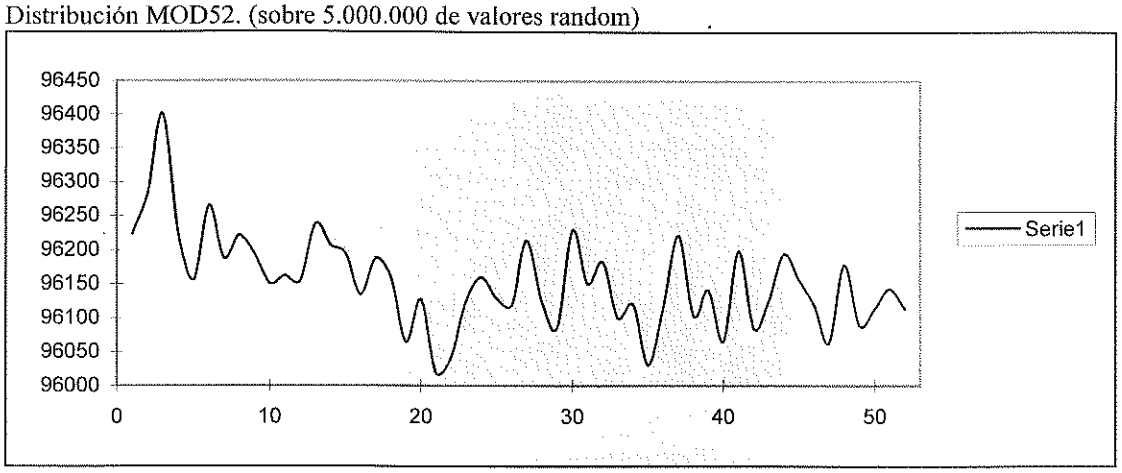
La distribución no es equilibrada, sobre todo en los primeros valores de la serie con picos muy altos en los valores 2,5 y 8. A partir del valor 34 la distribución es realmente muy mala y presenta un patrón que se repite con picos en los valores 34, 37,40, 43, 46 y 49 y disminuciones parejas en todos los valores medios.
El rango de distribución de la función es muy amplio desde 99200 hasta 100600, lo que equivale a una amplitud total de 1400.

Distribución MOD51. (sobre 5.000.000 de valores random)



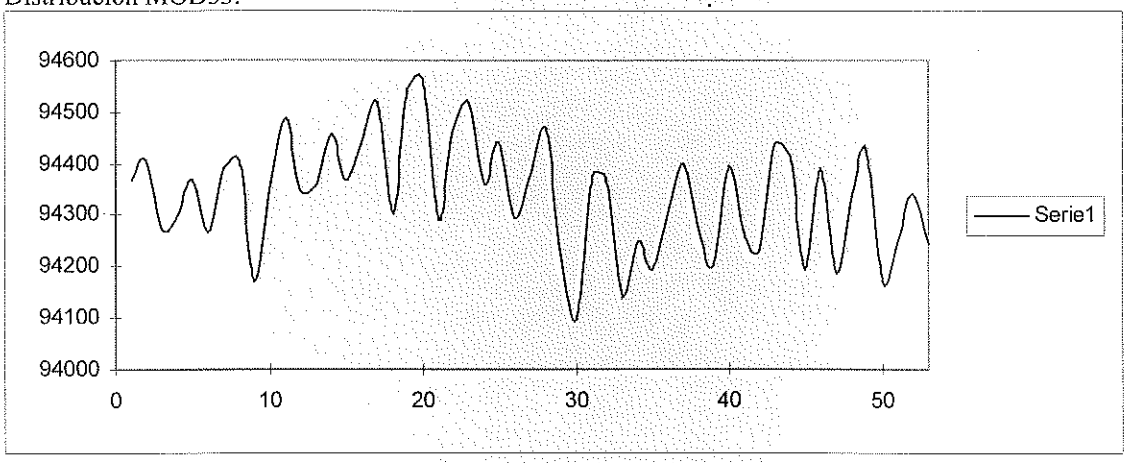

Análisis:
La distribución es mas equilibrada que en el caso anterior y ya no se presentan patrones repetitivos, hay algunos picos y caídas relativamente importantes en el comienzo de la serie pero el rango total de la misma es muy chico: la amplitud es de solamente 500.

Distribución MOD52. (sobre 5.000.000 de valores random)




Análisis:
La distribución es muy desequilibrada, con un pico especifico para el valor 2 y disminuciones pronunciadas en los valores 20 y 34 . Pese a tener una amplitud muy baja (450) la distribución muy desequilibrada lo hace inferior a la distribución MOD51.

Distribución MOD53.




Análisis:
La distribución es equilibrada y presenta una amplitud baja (600), la distribución de claves sin embargo presenta algunos patrones geométricos reconocibles con un mínimo muy malo en el valor 30.

Conclusión:

La peor distribución es la distribución MOD50, con una amplitud enorme de 1400 en comparación con otras distribuciones, las distribuciones MOD51,52 y 53 tienen amplitudes buenas, de las 3 MOD52 es la que presenta una distribución mas despareja. Entre MOD51 y MOD53 la elección se basa en la forma mas aleatoria que tiene la distribución MOD51 por lo que para el conjunto de claves que analizamos la función elegida seria esa.

De los 4 números probados 50 es el que mas divisores tiene, luego viene 52 y finalmente 51 y 53 que son números primos. En este test puede observarse como los números primos tienden a generar una distribución mas equilibrada y aleatoria que aquellos que no lo son para la función MOD. Entre los números que no son primos aquellos que tienen mas divisores son los que peores distribuciones generan, en particular aquellos que tienen muchos divisores chicos (2,3,4,5 etc). Por este motivo en las funciones de hashing que usan MOD se utiliza casi siempre un numero primo.

Las funciones que utilizan MOD se basan en una division para obtener el resultado de la función de hashing, existe otros esquema para distribuir claves denominado esquema multiplicativo.

$$h = \lfloor k * ((f(s) * A) \bmod 1) \rfloor$$

Esquema multiplicativo.

En donde f(s) es simplemente una funcion que convierte el string en un número y no tiene mayor incidencia sobre la distribución de la función, el único detalle a cuidar es que use todos los dígitos del string al hacer la conversión.

K es el espacio de direcciones y mod 1 implica tomar la parte decimal del resultado.
El número "A" es una constante multiplicativa.

La función mod, recordemos, dependía de encontrar un buen número sobre el cual hacer la división, los esquemas multiplicativos dependen de encontrar una buena constante "A". Numerosos tests y estudios se hicieron sobre este esquema llegandose a la asombrosa conclusion de que la mejor constante era o bien el número aureo (Φ) o bien un número similar al número aureo.

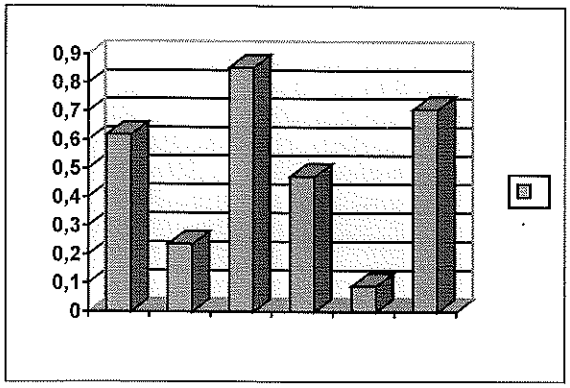
El número aureo se obtiene de:

$$\phi = \frac{\sqrt{5}-1}{2}$$

Φ = 0,6180339887499...

Este número tiene propiedades matemáticas, físicas y hasta psicológicas que muchas veces resultan inexplicables, una vez que alguien lo encuentra por primera vez ya no puede separarse de el.

La propiedad que tiene este numero es que si graficamos la función (Φ*n) mod 1. Para n=1, 2 , 3, 4, 5, 6, 7, ... obtenemos algo de la forma:




Distribucion de la constante Φ

En el grafico solo se ilustran los 6 primeros valores, pero como puede observarse la funcion distribuye los valores en el intervalo 0-1 de forma tal que para dos n consecutivos la distancia entre f(n) y f(n+1) siempre es grande. Aun mejor la distribucion es extremadamente pareja y aleatoria.

El esquema multiplicativo utilizando como constante multiplicativa a un número que distribuya bien es equivalente en cuanto a rendimiento a la utilización de la función MOD por un numero primo, el esquema multiplicativo tiene como ventaja que la operación de multiplicación es menos costosa que la división.

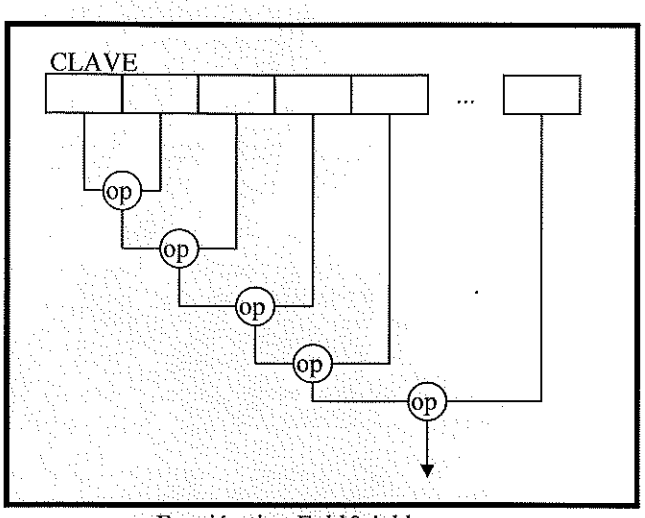
| *Constantes multiplicativas recomendadas:* |
|---|
| *0.6180339887* |
| *0.6161616161* |
| *0.6125423371* |

## Funciones de hashing genéricas.

A continuación analizamos algunas funciones de hashing genéricas para claves de longitud variable, junto con cada función se incluye un análisis de la distribución de la misma para un conjunto de datos formado por 2.500.000 claves de 64 bytes de longitud sobre un espacio de direcciones de longitud 3701.

### Fold & Add.

Las funciones tipo fold&add trabajan tomando porciones de la clave y operando sobre ellas, en general sumando de a partes o bien multiplicando o realizando otro tipo de operaciones. En este caso analizamos seis funciones tipo fold&add similares.

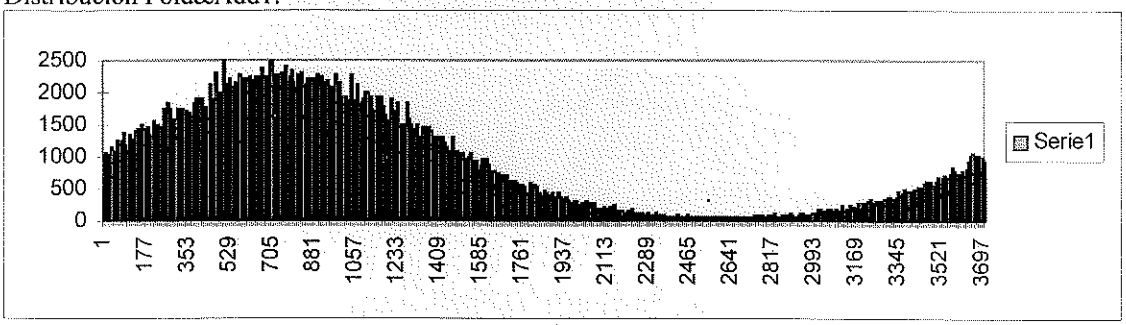



Función tipo Fold&Add

### Fold&Add1.

Esta función particiona la clave en grupos de 2 bytes. Si la longitud de la clave no es par completa con un byte en cero. La operación que efectúa entre 2 bytes es la suma. Por ultimo al resultado final de la función se le hace un mod de acuerdo al espacio de direcciones.

Distribución Fold&Add1:

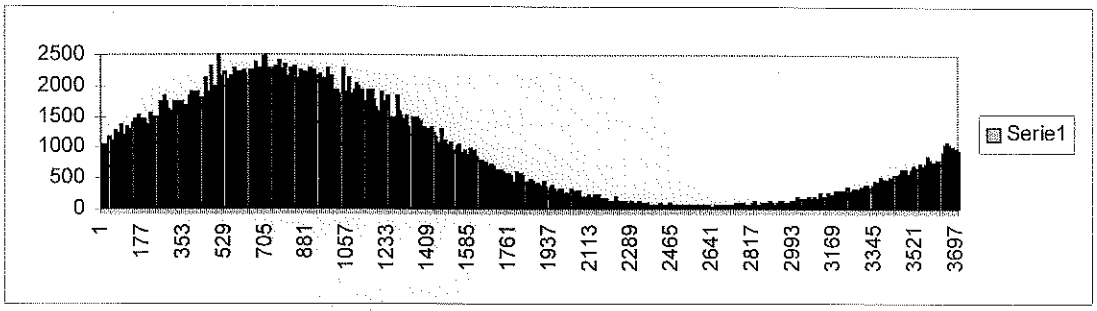


Análisis:
Esta es una distribución pésima, hay valores que la función nunca toma y hay otros en los cuales se concentran la mayoría de las claves. La forma geométrica senoidal de la distribución s un factor negativo extra ya que permite suponer que es fácil deducir los puntos en los cuales se producirán mayor cantidad de colisiones.

**Fold&Add2.**

Esta función es muy similar anterior variando únicamente la operación que se realiza para los 2 bytes que se van tomando de la clave, en este caso los bytes se suman y al resultado se le efectúa un mod por el espacio de direcciones. Al resultado final de esta función no hace falta aplicarle un mod.

Distribución Fold&Add2.

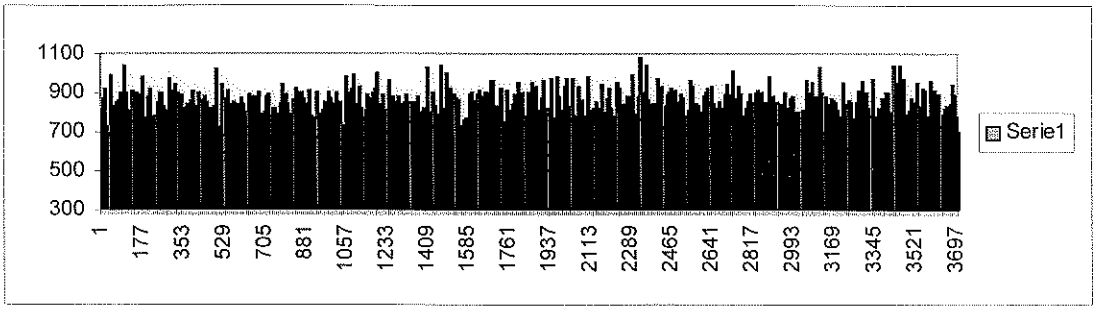



Análisis:
Esta es otra distribución lamentable con características similares a las anteriores, el motivo por el cual estas funciones presentan este tipo de distribución puede encontrarse en la decisión de procesar la clave de a 1 byte (se opera entre 2 bytes). En general es mejor tomar bloques de mayor cantidad de bytes de la clave.

**Fold&Add3.**

Esta función toma la clave de a dos bytes y opera con grupos de 4 bytes, si la longitud de la clave no es múltiplo de 4 completa con ceros. Al resultado de la función se le aplica un mod de acuerdo al espacio de direcciones.

Distribución Fold&Add3.

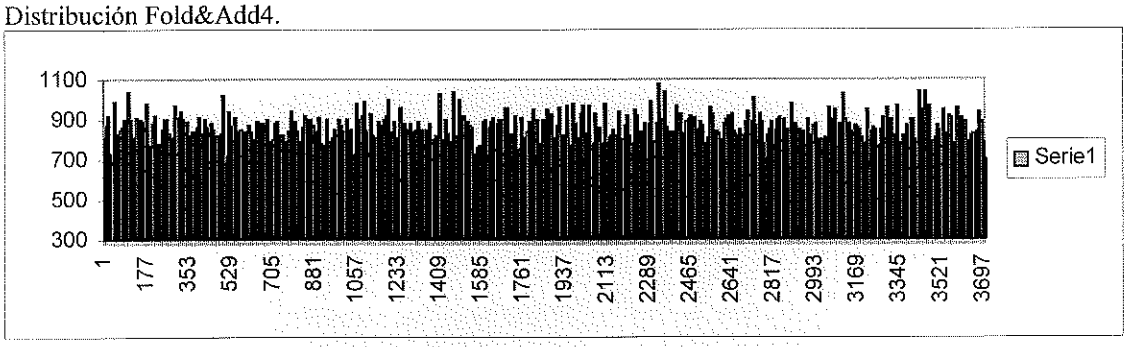


Análisis:
Esta es una distribución buena, con una amplitud chica (diferencia entre el máximo valor para las ocurrencias del espacio de direcciones). La amplitud es de 400 y la distribución es pareja y aleatoria.

**Fold&Add4.**

Esta es análoga a Fold&Add3 con la diferencia de que en la operación al resultado de la suma de los dos bloques de 2 bytes se le aplica un mod de acuerdo al espacio de direcciones, al resultado final de esta función no hace falta aplicarle nuevamente la función mod.

Distribución Fold&Add4.

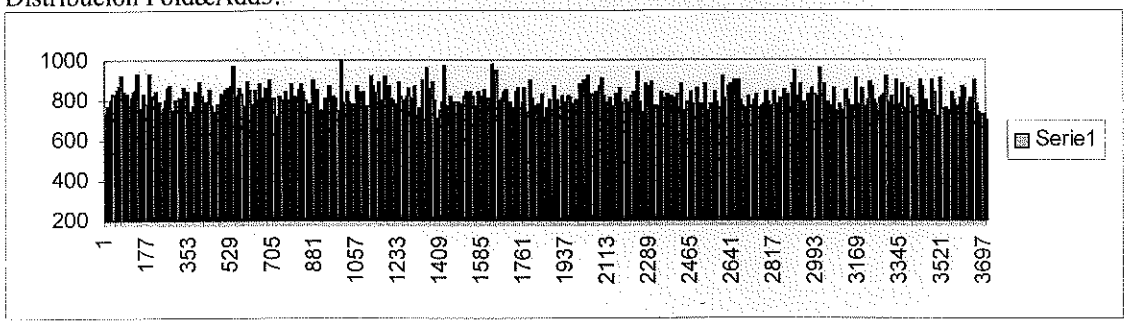



Análisis:
Fold&Add4 presenta una distribución casi idéntica a la de Fold&Add3

**Fold&Add5.**

Esta función trabaja operando sobre dos grupos de 4 bytes sumándolos, si la longitud de la clave no es múltiplo de 8 completa con ceros. Al resultado de la función se le hace un mod de acuerdo al espacio de direcciones.

Distribución Fold&Add5.

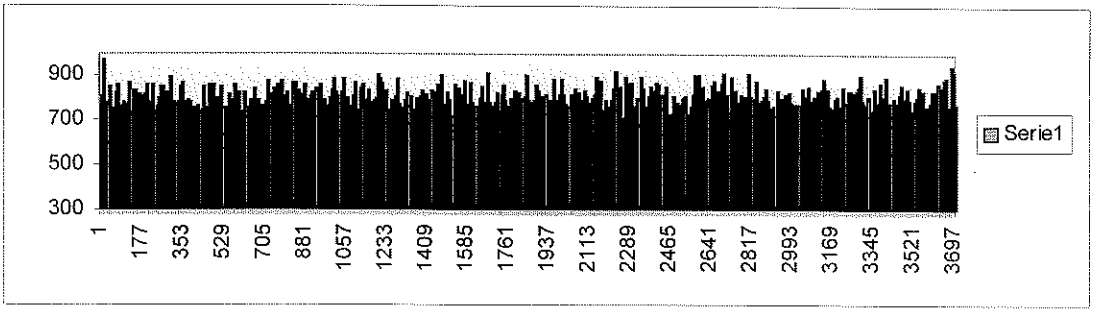



Análisis:
Esta es una distribución muy buena, muy equilibrada, aleatoria y con una amplitud incluso menor a la de Fold&Add3. Quizás la mejora en la distribución este dada por el hecho de tomar una mayor cantidad de bytes de la clave para cada operación que se hace.

**Fold&Add6.**

Es similar a Fold&Add5 pero al resultado de cada operación se le hace un mod de acuerdo al espacio de direcciones, de esta forma al resultado final de la función no hace falta aplicarle la función mod.

Distribución Fold&Add6.




Análisis:
Otra distribución muy buena, con un patrón aleatorio y amplitud muy escasa.

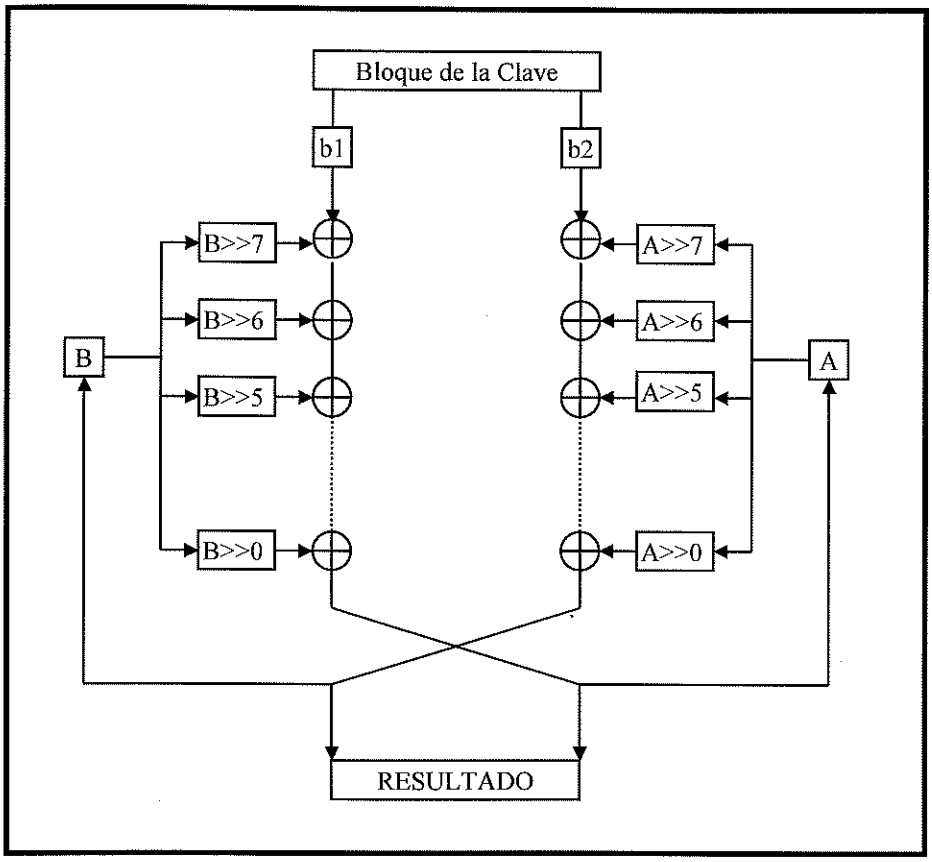
Conclusión:
Las funciones tipo Fold&Add son sencillas de implementar y se usan con frecuencia como función de hashing, en general distribuyen bastante bien siempre y cuando se tomen sub-bloques de la clave de por lo menos 4 bytes (aunque en el ejemplo con 2 era suficiente). La operación mas común para las funciones fold&add es la suma, la inclusión de un mod luego de cada operación no parece ser determinante en cuanto a la distribución de la función.

**Krhash.**

Krhash es una función de hashing diseñada en 1995, la función trabaja con sub-bloques de la clave de 2 bytes (si la longitud de la clave no es par se agrega un byte 0xA3) y devuelve un valor de 16 bits (2 bytes), se aplica a claves de al menos 2 bytes y hasta cualquier longitud.

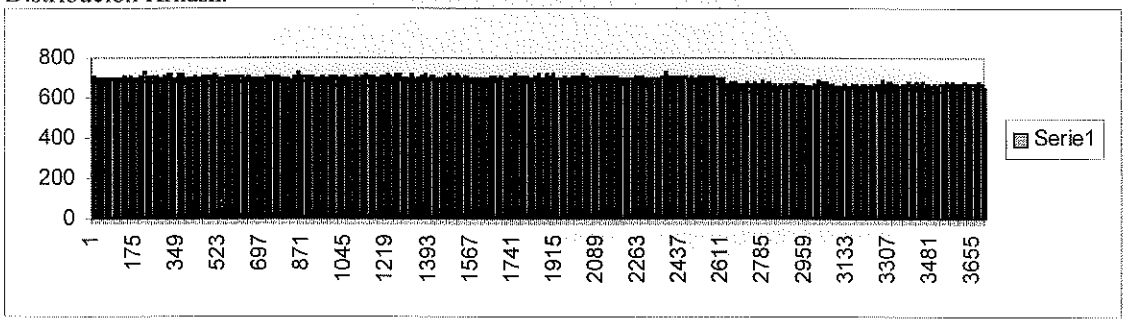
Krhash se basa en operaciones tipo XOR de cada byte de la clave con una mascara deslizante constante para el primer bloque de 2 bytes y que luego varía. El proceso es el siguiente.

En primer lugar se inicializan dos variables A=0xBFA7 y B=0x0123. El sub-bloque de la clave de 2 bytes se divide en dos variables b1 y b2 de 1 byte cada una. Luego se procesa b1 de la siguiente manera. Primero se hace un XOR de b1 contra el primer bit de B, al resultado del dicho XOR se le hace un XOR contra los dos primeros bits de B y así sucesivamente hasta hacer un XOR contra los últimos 8 bits de B. Lo mismo se hace con b2 y la constante A. Luego a A se le asigna el valor de b1 y a B se le asigna el valor de b2. La función devuelve la concatenación de A y B. Al resultado final de la función se le aplica un mod.




Krhash, descripción de la función.

Distribución Krhash.



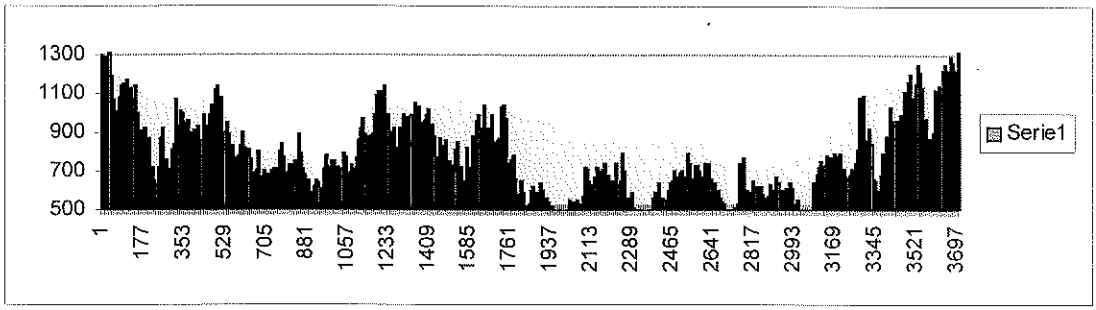

Análisis:
Esta es una distribución definitivamente excelente, la amplitud es bajisima (menos de 100) y la distribución es equilibrada, aunque presentando una disminución pareja a partir del valor 2611, la amplitud tan baja hace que este escalón sea muy poco significativo en comparación con las variaciones que presentaban otras funciones.

**Modulus.**

La función modulus es otra función genérica con una funcionalidad idéntica a la de Krhash, procesa los datos en grupos de 2 bytes (si la longitud de la clave no es par completa con ceros) y devuelve un valor de 16 bits al cual luego hay que aplicarle la función mod para llevarlo al espacio de direcciones que corresponda.

La función comienza dividiendo el bloque de 2 bytes de la clave en dos partes b1 y b2 cada una de 8 bits. Luego copia el valor mayor entre b1 y b2 en a1 y el valor menor en a2, a continuación hace un xor entre b1 y k1, y entre b2 y k2. k1 y k2 son dos variables estáticas inicialmente en cero. Luego hace a1 mod 53 y a2 mod 67, luego eleva a1 y a2 al cuadrado y toma de los resultados de dichas operación los 8 bits del medio. Por ultimo hace el xor de los resultados con k1 y k2 respectivamente asigna dichos valores a k1 y k2 y devuelve la concatenación de k1 y k2.

Distribución Modulus.




Análisis:
Pese a las operaciones realizadas la función modulus en general no presenta una distribución muy uniforme, la amplitud es muy elevada y hay concentraciones de valores en los extremos, el patrón de distribución mas allá de la concentración en las puntas es aleatorio pero aun con este aliciente la distribución es inferior.

**Funciones trigonometricas.**

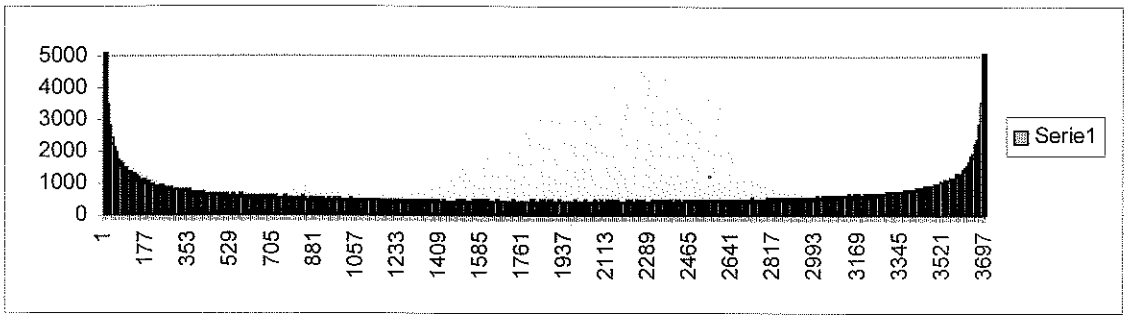
En algunas ocasiones se utiliza, en forma equivocada, para construir funciones de hashing el resultado devuelto por una o mas funciones trigonometricas, como por ejemplo el seno o el coseno. A dicho resultado comprendido entre -1 y 1 se lo lleva al espacio de direcciones mediante una transformación simple, de esta forma se evita la utilización de la función mod. La desventaja reside en que las funciones trigonometricas no distribuyen sus valores en forma pareja sino que presentan concentraciones visibles en algunas partes.

Ejemplo:

Si tenemos un espacio de direcciones de 0..3700 podemos hacer.

f(clave) = (cos(clave)*1850)+1850.

Distribución Coseno.



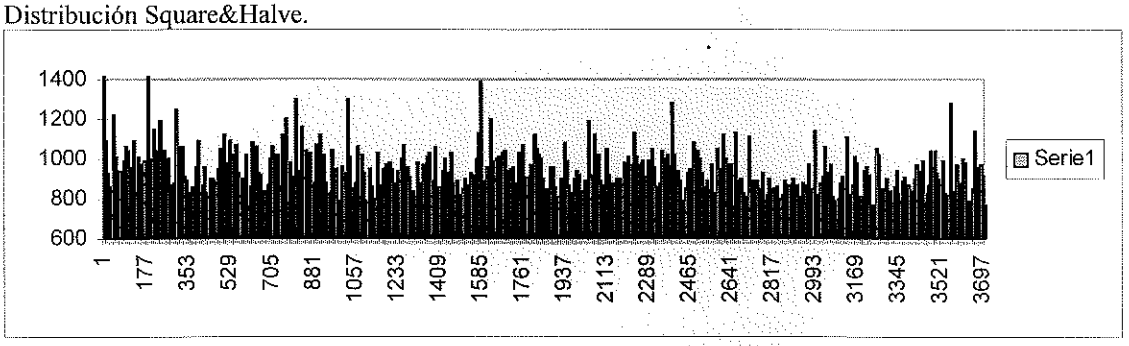

Análisis:
Como puede observarse la distribución es inadmisible para cualquier tipo de función de hashing, la distribución es predecible, con una concentración altísima en los extremos y bajisima en los valores medios. Esto puede deducirse fácilmente si consideramos el gráfico de la función coseno ya que la función no es lineal y si bien devuelve valores comprendidos en el -1..1 tiene muchos mas puntos cercanos al -1 o al 1 que a los valores medios. Por ello la concentración altísima de la función de hashing en los extremos.

El uso de funciones trigonometricas dentro de una función de hashing es desaconsejable.

**Square&Halve.**

Esta función trabaja considerando a toda la clave como si fuera un único gran numero, luego elevan dicho numero al cuadrado y sobre el resultado toman algunos de los dígitos del medio, de esta forma el resultado obtenido es dependiente de la clave pero a su vez es también altamente aleatorio. Sobre el valor devuelto por la función se aplica un mod.

Distribución Square&Halve.




Análisis:
La distribución es buena, tiene una amplitud elevada debido a varios picos que se producen pero la distribución en si es aleatoria, este tipo de distribución es común en las funciones de este tipo, varios picos aislados y el resto distribuido en forma random. Si no se requiere un especial cuidado en el numero de colisiones que se produce en algunos valores la función es buena.

La función Square&Halve tiene una distribución bastante equilibrada aunque, resulta un tanto difícil de implementar y además es mucho mas lenta que otras funciones que no involucran operaciones aritméticas complejas.

## Funciones de Hashing Unidireccionales.

Una función de hashing unidireccional H(M) aplicada a un mensaje M de longitud arbitraria devuelve un valor de longitud fija h comúnmente llamado hash-value.

Muchas funciones pueden producir un resultado de longitud fija a partir de un mensaje, pero las funciones de hashing unidireccionales tienen, además, que cumplir con requisitos adicionales.

- Dado M debe ser sencillo calcular h.
- Dado h es muy difícil encontrar un M tal que H(M)=h.
- Dado M es muy difícil encontrar otro mensaje M ' tal que H(M)=H(M ') -Sinónimos-

El objetivo fundamental de las funciones de hashing unidireccionales es proporcionar una identificación unívoca del mensaje con un valor de longitud sensiblemente inferior a la del mensaje mismo. Las funciones de hashing unidireccionales se aplican en la autenticación de documentos, firmas digitales, códigos de verificación y en cualquier otra circunstancia en la que se requiera un 'concentrado' del mensaje que permita identificarlo en forma segura.

Además, las funciones de hashing unidireccionales buenas, funcionan muy bien como funciones de hashing tradicionales presentando una distribución sumamente equilibrada. La diferencia entre una función de hashing tradicional o simple y una unidireccional es que las segundas deben cumplir con requisitos de seguridad que las primeras no necesariamente cumplen.

El esquema utilizado para firmar en forma digital un documento es el siguiente: a partir del mensaje M se extrae un concentrado h usando la función de hashing y luego se encripta h con la clave privada del usuario que firma, el receptor desencripta con la clave publica de quien dice haber firmado y luego controla que el concentrado del mensaje sea el mismo que el concentrado desencriptado. (Ambos usuarios disponen de la función de hashing que extrae el concentrado). Con este procedimiento se verifica tanto la firma como la integridad del mensaje, ya que un cambio en el mismo implicaría un cambio en el concentrado.

Para que este procedimiento sea seguro es necesario que la función de hashing utilizada cumpla de la mejor forma posible con las propiedades enunciadas anteriormente, veamos los problemas que podrían presentarse con una función de hashing poco segura.

**Caso 1**: Si dado un h cualquiera es posible hallar un M tal que H(M)=h.
En este caso se podría capturar un documento cualquiera firmado por MrX y utilizar el concentrado h para luego encontrar otro mensaje M que de como resultado el mismo concentrado y hacer creer que MrX ha firmado ambos documentos.

**Caso 2**: Si dado un M es posible encontrar otro mensaje M ' tal que H(M)=H(M ')
En este caso con un cierto mensaje se podría encontrar un mensaje sinónimo y si un usuario malintencionado obtuviese la firma del primer mensaje entonces también obtendría la firma del sinónimo.

En algunas aplicaciones criticas no solo es necesaria una función unidireccional sino también una función que resistente a colisiones, una función de hashing unidireccional es además resistente a colisiones sii:

- Es difícil encontrar dos mensajes cualesquiera M y M ' tales que H(M) = H(M ')

Si esta propiedad no se cumple podríamos tener otro caso de fraude:

**Caso 3**: Si es posible encontrar encontrar dos mensajes M y M ' tales que H(M)=H(M ')
Aquí tenemos un problema muy similar al caso 2: cualquier persona malintencionada podría preparar dos documentos (contratos por ejemplo) similares que den como resultado el mismo h, uno neutro y otro que favorezca ampliamente a una de las partes, dicha persona le daría a firmar a MrX el documento neutro para luego utilizar la firma de MrX para el otro documento (como ambos originan el mismo h el concentrado encriptado será el mismo). MrX no tendría forma de demostrar cual fue el documento firmado por el.

La longitud de h es un factor muy importante de este tipo de funciones de hashing ya que de dicha longitud depende el éxito o no de los ataques por fuerza bruta (probar todas las posibles combinaciones), en este tipo de funciones hay que tener en cuenta el denominado birthday-atack que consiste en encontrar sinónimos. Si dada la función es posible encontrar sinónimos entonces la función como vimos en el caso 3 no es resistente a colisiones y en la mayoría de los casos no será lo suficientemente segura. Encontrar sinónimos por fuerza bruta es mucho mas sencillo de lo que parece.

El nombre birthday-attack surge de la paradoja de los cumpleaños que dice lo siguiente:

¿Cuantas personas tiene que haber en una habitación para que haya mas de 50% de chances de que una de ellas cumpla años el mismo día y el mismo mes que una cierta persona 'X'?

Repuesta: 253.

¿Cuantas personas tiene que haber en una habitación para que haya mas de 50% de chances de que dos de ellas cumplan años el mismo día y el mismo mes?

Respuesta: 23. (¿sorprendidos?)

De la respuesta a estas preguntas puede verse cuanto mas fácil es buscar sinónimos que tratar de acertar un resultado aislado.

Las funciones de hashing que devuelven valores de 64 bits de longitud son demasiado débiles como para sobrevivir a un birthday-attack,es por esto que la mayoría de las funciones de hashing unidireccionales devuelven valores de 128 bits. Con 128 bits de salida hay que probar aproximadamente 2^64 documentos distintos para encontrar dos sinónimos, esto es bastante seguro, si se requiere de seguridad super-extrema pueden usarse funciones que devuelvan mas bits, con 160 bits de salida se necesitarían 2^80 pruebas al azar para encontrar dos sinónimos.

A partir de una función de hashing que devuelve n bits es posible obtener mas bits de salida mediante un proceso de expansión, el proceso que se sugiere es el siguiente:

1) Generar h a partir del mensaje M.
2) Concatenar el mensaje con h obteniendo M '.
3) Generar h' a partir de M'
4) Concatenar h y h' y devolver dicho valor.

Los pasos 1 a 3 se pueden repetir tantas veces como sea necesario para obtener un valor h de la longitud que se desee. Nunca se pudo demostrar si esta técnica es segura o insegura pero varios especialistas han manifestado sus dudas acerca del procedimiento.

**Construcción de funciones de hashing unidireccionales.**

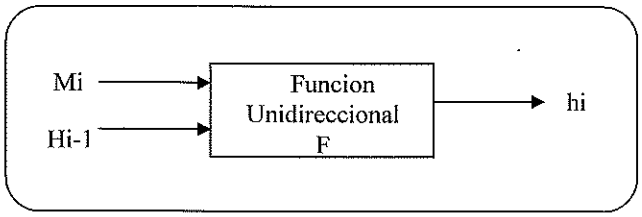
En general la mayoría de las funciones de hashing unidireccionales trabajan dividiendo al mensaje M en bloques de una cierta longitud, a partir de M0 se obtiene el valor h0 de alguna manera y luego en cada paso se hace.

hi = f(Mi,hi-1)

El ultimo h cuando ya no quedan mas bloques del mensaje por procesar es el valor que devuelve la función.

En algunas funciones al mensaje original se le agrega una representación binaria de su longitud, de esta forma se pretende evitar que dos mensajes de distinta longitud sean sinónimos, esta técnica se denomina MD-strengthening.

Funciones de Hashing Unidireccionales:




A continuación se analizan algunas de las funciones de hashing unidireccionales mas utilizadas.

## N-HASH.

N-Hash es un algoritmo inventado por un compañía de comunicaciones japonesa. Devuelve valores de 128 bits. Para funcionar utiliza bloques de 128 bits y una función de pseudo-aleatorizacion compleja. El hash de cada bloque es función del bloque y del valor de hash anterior.

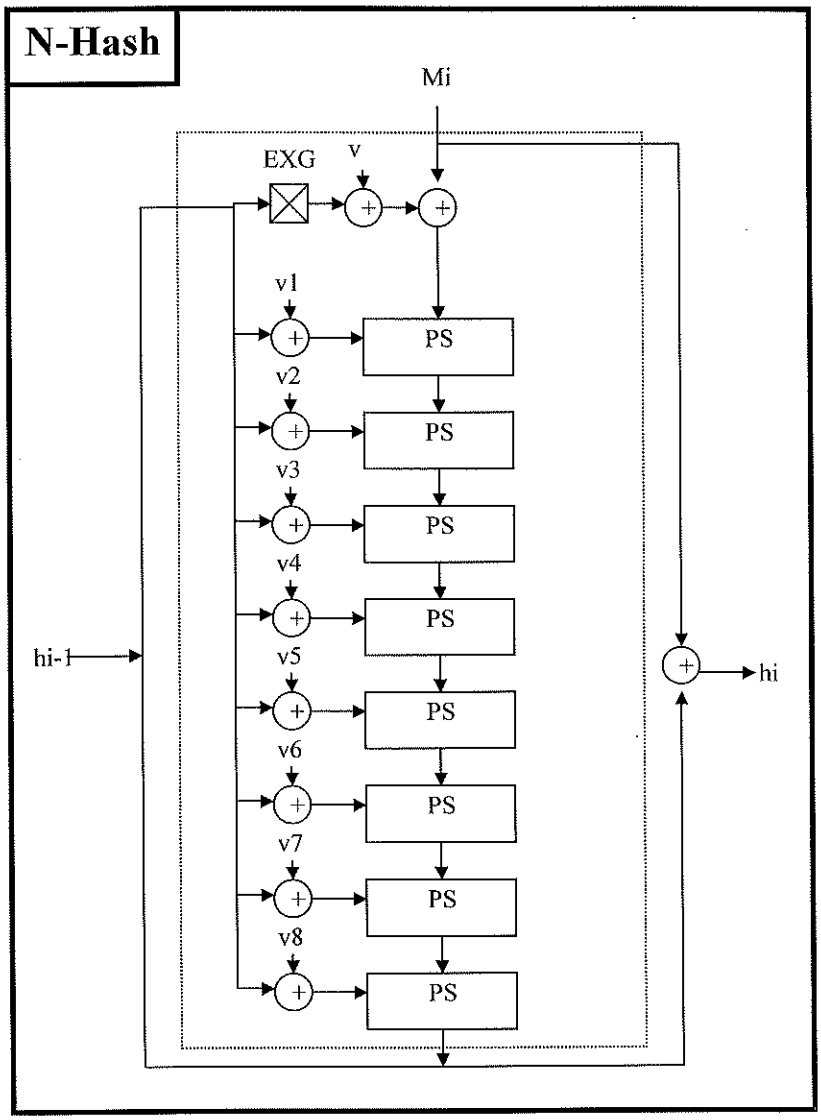
H0=I donde I es un valor random inicial.
Hi = g(Mi,Hi-1) ⊕ Mi ⊕ Hi-1

El resultado final es el H resultante del ultimo bloque. El valor random I puede ser cualquier valor determinado por el usuario o por el algoritmo, incluso todos ceros. Su única función es rellenar los bits faltantes.

La función g es compleja, el esquema muestra el funcionamiento general del algoritmo.

Al comienzo se swapean los 64 bits derechos con los 64 bits izquierdos de los 128 bits del hash del bloque previo Hi-1. El resultado es XOReado con una mascara que repite unos y ceros (1010101...) y luego el resultado es XOReado con el bloque actual Mi. Este valor pasa luego por N (8 en el ejemplo) etapas de procesamiento. El otro input de cada etapa de procesamiento es el valor de hash anterior Hi-1 XOReado con una de 8 constantes binarias.

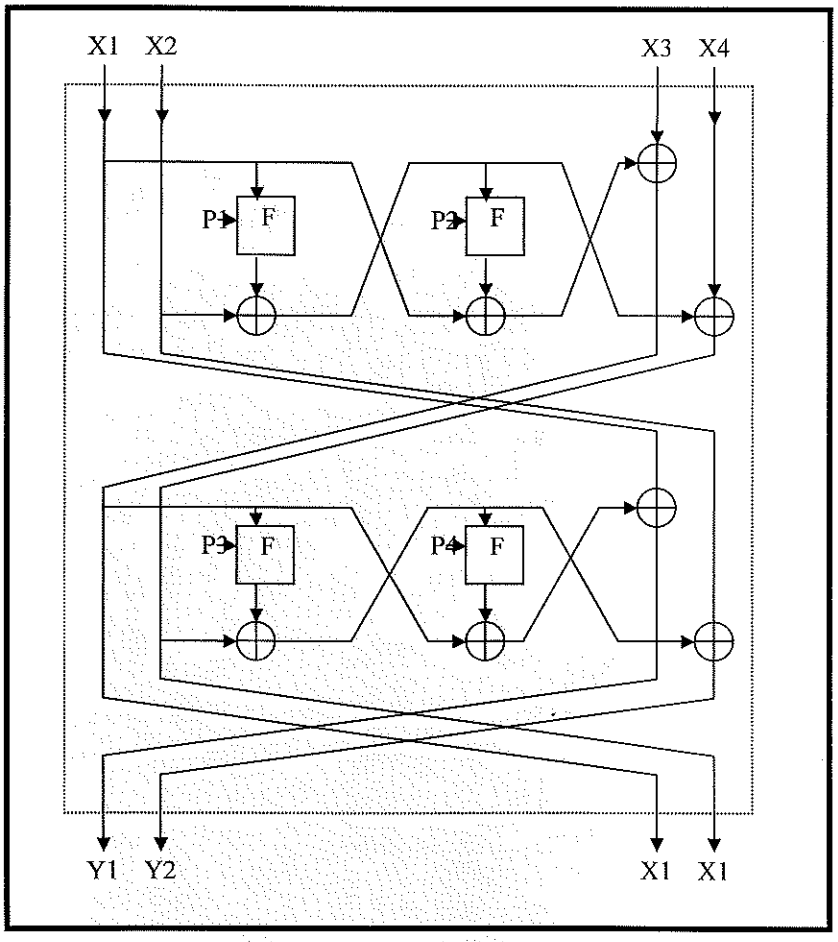
Esquema N-Hash. (En línea de puntos la función g)




EXG=Swappear las dos mitades de 64 bits.
⊕ = XOR.
v = 10101010... (128 bits)

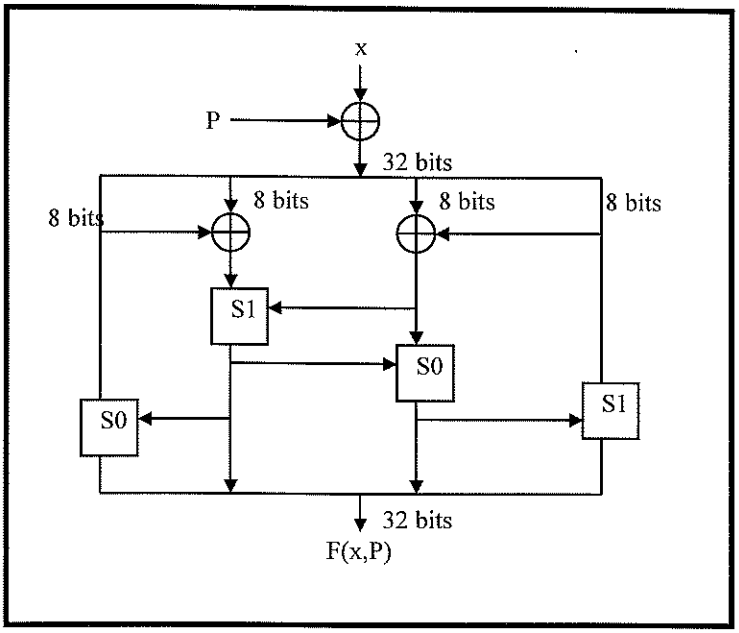
PS = Etapa de procesamiento.
vj=Q||Aj1||Q||Aj2||Q||Aj3||Q||Aj4
Q=000000... (24 bits)
Ajk=4*(j-1)+k (8 bits)
Hi = g(Mi,Hi-1) ⊕ Mi ⊕ Hi-1

Cada etapa de procesamiento PS de N-Hash toma dos inputs, denominados X y P, la obtención de X y P se ve en el esquema anterior. Para cada etapa PS X y P se descomponen en 4 sub-bloques de 32 bits cada uno denominados X1..X4 y P1..P4. El esquema de procesamiento es:




Etapa PS del algoritmo N-Hash. El Output es Y=PS(X,P)

La función "f" utilizada dentro de cada etapa de procesamiento es a su vez bastante compleja, a continuación el esquema de la función "f".




Función F

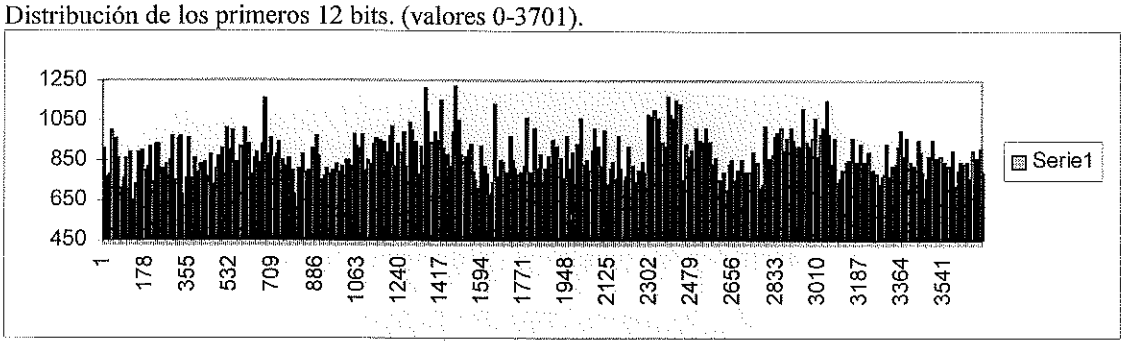
Por ultimo restan las especificaciones de las funciones S0 y S1:

S0(a,b) = rotate left two bits ( (a + b) mod 256 )
S1(a,b) = rotate left two bits ( ( a + b + 1 ) mod 256 )

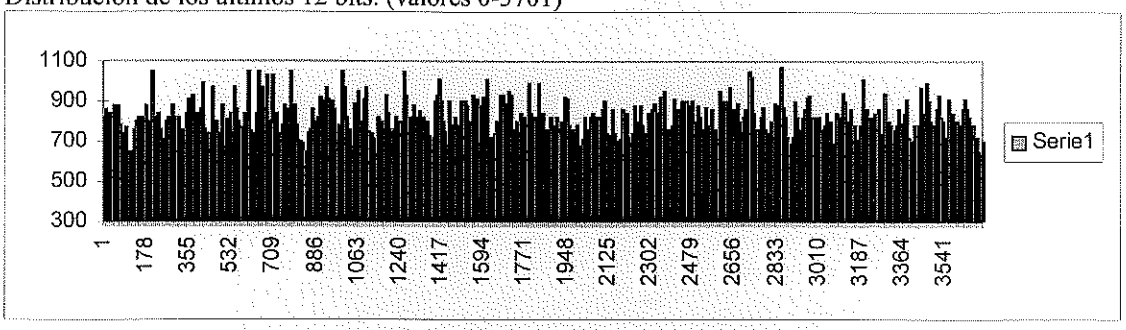
Los autores de N-Hash recomiendan usarlo con al menos 8 pasadas. El algoritmo no es muy seguro y ha sufrido algunos ataques exitosos. Bert deb Boer, Biham y Shamir han conducido investigaciones exitosas que permiten obtener sinónimos de N-Hash. En general el algoritmo no es muy utilizado como función unidireccional, aunque si puede usarse efectivamente como función de hashing tradicional ya que presenta una distribución muy pareja.

Distribución N-Hash.

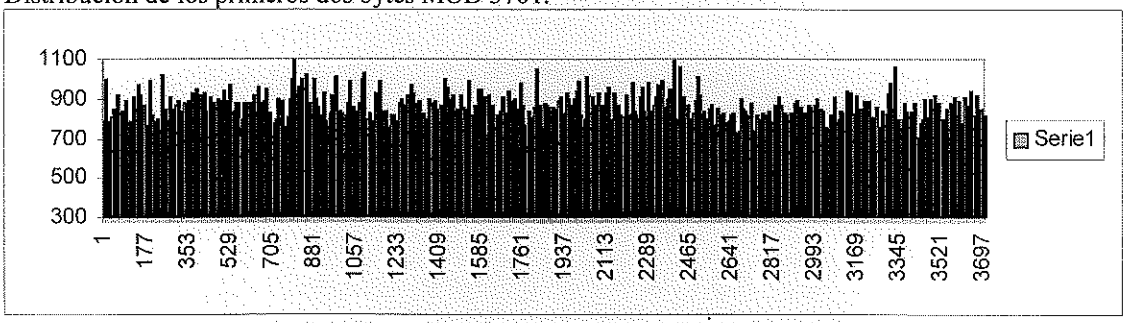
Distribución de los primeros 12 bits. (valores 0-3701).



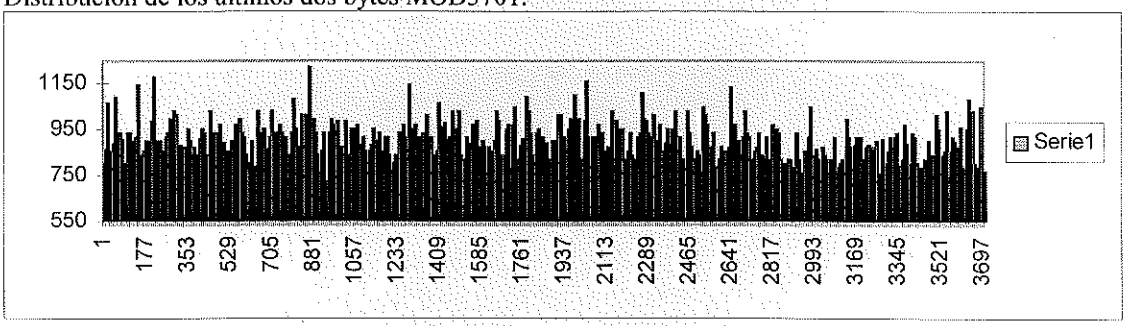

Distribución de los últimos 12 bits. (valores 0-3701)




Distribución de los primeros dos bytes MOD 3701.




Distribución de los últimos dos bytes MOD3701.



## MD5.

MD5 es una función desarrollada por Ron Rivest. MD significa "Message Digest" (algo así como digerir un mensaje). El algoritmo devuelve un valor de 128 bits. MD5 es una versión mejorada del algoritmo original de Rivest: M4. Los objetivos de MD5 son:

Seguridad. Es computacionalmente imposible encontrar dos mensajes que hashean al mismo valor. No hay ataques mas eficientes contra la función que los de fuerza bruta.

Seguridad directa. MD5 no esta basado en ninguna suposición, como por ejemplo la dificultad de factorear números.

Velocidad. MD5 puede ser implementado en forma sumamente eficiente, utiliza únicamente manipulación de bits sobre operandos de 32 bits.

Simplicidad. MD5 es un algoritmo relativamente sencillo que no utiliza estructuras de datos complejas ni gran volumen de código.

Descripción del algoritmo.

MD5 procesa el mensaje en bloques de 512 bits. Cada bloque de 512 bits se subdivide además en 16 sub-bloques de 32 bits cada uno. La salida del algoritmo son 4 bloques de 32 bits cada uno, obteniéndose 128 bits como resultado.

En primer lugar el mensaje se expande de forma tal que su longitud sea exactamente 64 bits mas chica que el próximo múltiplo de 512. Esto se hace agregando un bit "1" y luego tantos ceros como sea necesario. Luego se guarda en 64 bits la representación binaria de la longitud original del mensaje. Esto sirve para convertir la longitud del mensaje en un múltiplo exacto de 512 bits a la vez que se asegura que mensajes diferentes no van a convertirse en el mismo mensaje luego de expandirlos.

Luego se inicializan 4 variables de 32 bits.

```
A=0x01234567
B=0x89ABCDEF
C=0xFEDCBA98
D=0x76543210
```

Estas variables se denominan variables de encadenamiento (chaining-variables).

Luego comienza el ciclo principal del algoritmo, el cual continua hasta que no queden mas bloques de 512 bits por procesar. Las cuatro variables se copian en otras cuatro variables auxiliares denominadas a,b,c y d. El ciclo principal tiene 4 vueltas todas muy similares. Cada vuelta usa una operación diferente 16 veces. Cada operación es una función no-lineal de tres de las variables a,b,c y d. El resultado de la función se suma a la cuarta variable, a un sub-bloque del mensaje y a una constante, luego el resultado se rota a la izquierda una cantidad variable de bits y se suma a,b,c o d. El resultado final se asigna a una de las variables a,b,c o d.

Hay cuatro funciones no-lineales:

```
F(X,Y,Z) = (X^Y)v((-X)^Z)

G(X,Y,Z) = (X^Z)v(Y^(-Z))

H(X,Y,Z) = X⊕Y⊕Z

I(X,Y,Z) = Y⊕(Xv(-Z))
```

Estas funciones están diseñadas de forma tal que si los bits de X,Y,Z son independientes los bits del resultado de la función también lo serán. La función F es el condicional de bits (If X then Y else Z). La función H es el operador de paridad.

Siendo Mj el j-esimo sub-bloque de 32 bits del bloque de 512 bits a procesar (de 0 a 15) y siendo <<< la operación de rotar hacia la izquierda una cierta cantidad de bits las cuatro operaciones son:

```
FF(a,b,c,d,Mj,s,ti) => a = b+((a+F(b,c,d)+Mj+ti)<<<s)
GG(a,b,c,d,Mj,s,ti) => a = b+((a+G(b,c,d)+Mj+ti)<<<s)
HH(a,b,c,d,Mj,s,ti) => a = b+((a+H(b,c,d)+Mj+ti)<<<s)
II(a,b,c,d,Mj,s,ti) => a = b+((a+I(b,c,d)+Mj+ti)<<<s)
```

Las cuatro vueltas del algoritmo (64 pasos) son:

Vuelta 1:

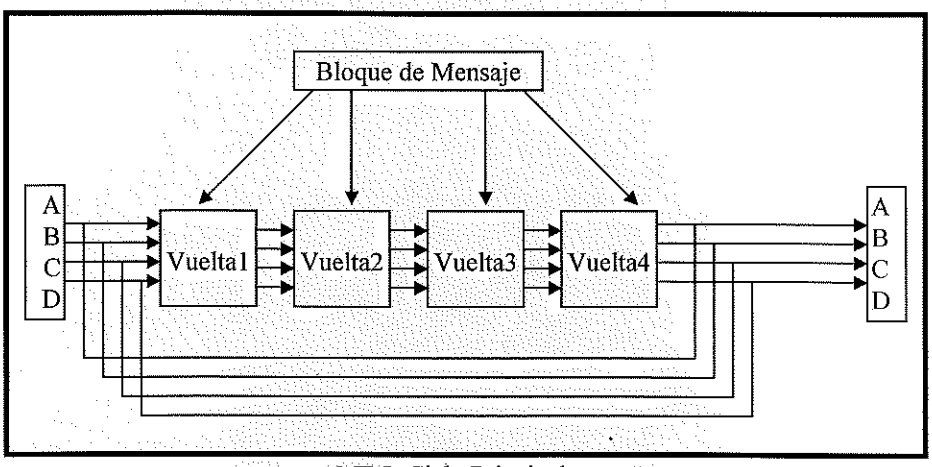
```
FF(a,b,c,d, M0, 7,0xd76aa478)
FF(d,a,b,c, M1,12,0xe8c7b756)
FF(c,d,a,b, M2,17,0x242070db)
FF(b,c,d,a, M3,22,0xc1bdceee)
FF(a,b,c,d, M4, 7,0xf57c0faf)
FF(d,a,b,c, M5,12,0x4787c62a)
FF(c,d,a,b, M6,17,0xa8394613)
FF(b,c,d,a, M7,22,0xfd469501)
FF(a,b,c,d, M8, 7,0x698098d8)
FF(d,a,b,c, M9,12,0x8b44f7af)
FF(c,d,a,b,M10,17,0xffff5bb1)
FF(b,c,d,a,M11,22,0x895cd7be)
FF(a,b,c,d,M12, 7,0x6b901122)
FF(d,a,b,c,M13,12,0xfd987193)
FF(c,d,a,b,M14,17,0xa679438e)
FF(b,c,d,a,M15,22,0x49b40821)
```

```
Vuelta 2:

GG(a,b,c,d, M1, 5,0xf61e2562)
GG(d,a,b,c, M6, 9,0xc040b340)
GG(c,d,a,b,M11,14,0x265e5a51)
GG(b,c,d,a, M0,20,0xe9b6c7aa)
GG(a,b,c,d, M5, 5,0xd62f105d)
GG(d,a,b,c,M10, 9,0x02441453)
GG(c,d,a,b,M15,14,0xd8a1e681)
GG(b,c,d,a, M4,20,0xe7d3fbc8)
GG(a,b,c,d, M9, 5,0x21e1cde6)
GG(d,a,b,c,M14, 9,0xc33707d6)
GG(c,d,a,b, M3,14,0xf4d50d87)
GG(b,c,d,a, M8,20,0x455a14ed)
GG(a,b,c,d,M13, 5,0xa9e3e905)
GG(d,a,b,c, M2, 9,0xfcefa3f8)
GG(c,d,a,b, M7,14,0x676f02d9)
GG(b,c,d,a,M12,20,0x8d2a4c8a)

Vuelta 3:

HH(a,b,c,d, M5, 4,0xfffa3942)
HH(d,a,b,c, M8,11,0x8771f681)
HH(c,d,a,b,M11,16,0x6d9d6122)
HH(b,c,d,a,M14,23,0xfde5380c)
HH(a,b,c,d, M1, 4,0xa4beea44)
HH(d,a,b,c, M4,11,0x4bdecfa9)
HH(c,d,a,b, M7,16,0xf6bb4b60)
HH(b,c,d,a,M10,23,0xbebfbc70)
HH(a,b,c,d,M13, 4,0x289b7ec6)
HH(d,a,b,c, M0,11,0xeaa127fa)
HH(c,d,a,b, M3,16,0xd4ef3085)
HH(b,c,d,a, M6,23,0x04881d05)
HH(a,b,c,d, M9, 4,0xd9d4d039)
HH(d,a,b,c,M12,11,0xe6db99e5)
HH(c,d,a,b,M15,16,0x1fa27cf8)
HH(b,c,d,a, M2,23,0xc4ac5665)
```

```
Vuelta 4:

II(a,b,c,d, M0, 6,0xf4292244)
II(d,a,b,c, M7,10,0x432aff97)
II(c,d,a,b,M14,15,0xab9423a7)
II(b,c,d,a, M5,21,0xfc93a039
II(a,b,c,d,M12, 6,0x655b59c3)
II(d,a,b,c, M3,10,0x8f0ccc92)
II(c,d,a,b,M10,15,0xffeff47d)
II(b,c,d,a, M1,21,0x85845dd1)
II(a,b,c,d, M8, 6,0x6fa87e4f)
II(d,a,b,c,M15,10,0xfe2ce6e0)
II(c,d,a,b, M6,15,0xa3014314)
II(b,c,d,a,M13,21,0x4e0811a1)
II(a,b,c,d, M4, 6,0xf7537e82)
II(d,a,b,c,M11,10,0xbd3af235)
II(c,d,a,b, M2,15,0x2ad7d2bb)
II(b,c,d,a, M9,21,0xeb86d391)
```

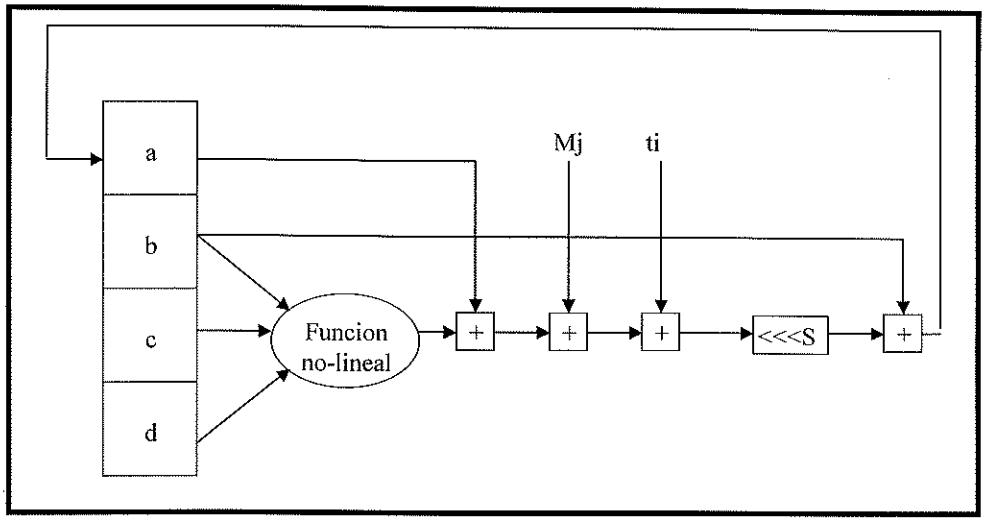
Las constantes ti fueron elegidas como la parte entera de 2^32*abs(sin(i)) con i en radianes.

Luego de estos 64 pasos a,b,c y d se suman a A,B,C y D y el algoritmo pasa al próximo bloque de datos, cuando no hay mas bloques el resultado final es la concatenación de A,B,C y D.




MD5: Ciclo Principal.

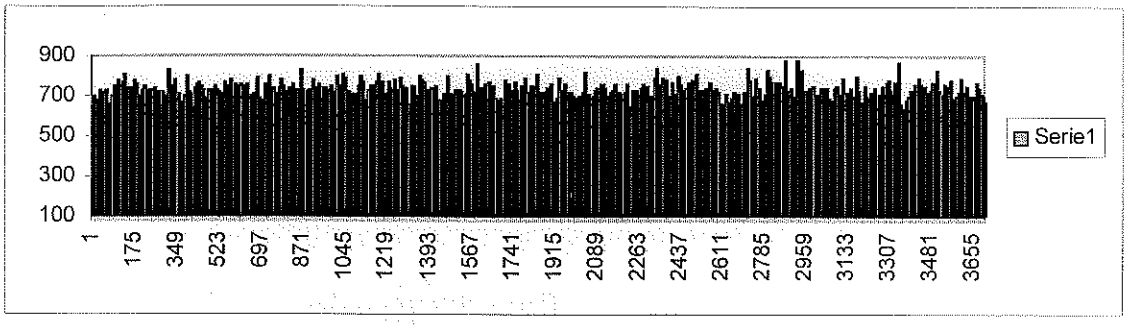
MD5 es un algoritmo bastantes seguro, los ataques mas importantes contra el MD5 han sido registrados por Berson y Den Boer-Bosselaers. Berson logro un análisis muy bueno contra una vuelta individual de MD5 pero sus resultados no tienen ningún éxito contra las 4 vueltas del algoritmo. Den Boer y Bosselaers lograron algunas colisiones usando parte del algoritmo lo cual no implica que puedan encontrar colisiones sobre el algoritmo entero. En general la opinión de los expertos sobre MD5 es que aparenta tener alguna debilidad pero que esta aun no puede ser encontrada. Por el momento MD5 es un algoritmo muy seguro y muy utilizado por diversas aplicaciones aunque si de seguridad militar se trata MD5 puede despertar algunos temores.



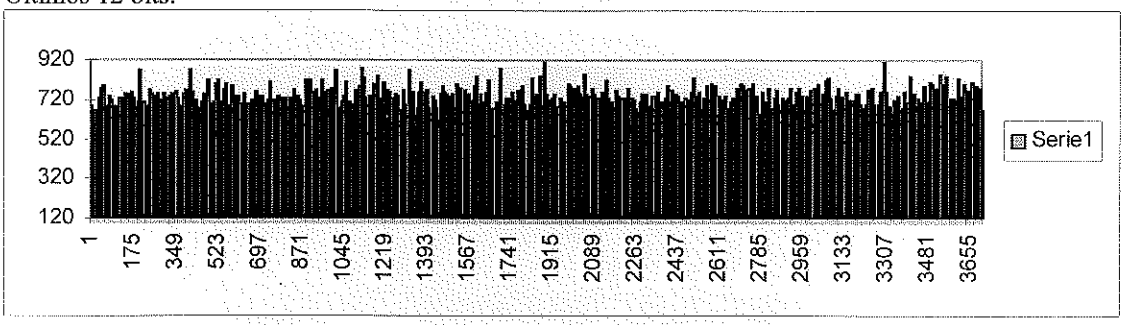

Esquema de uno de los 64 pasos de MD5.

Distribución MD5.

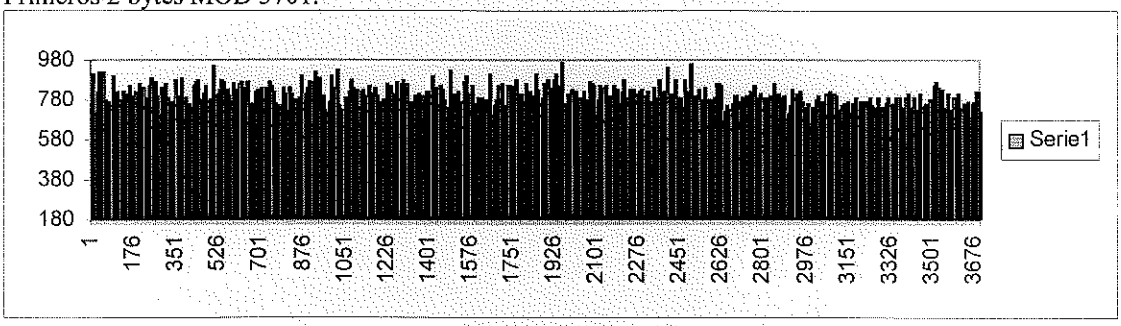
Primeros 12 bits.



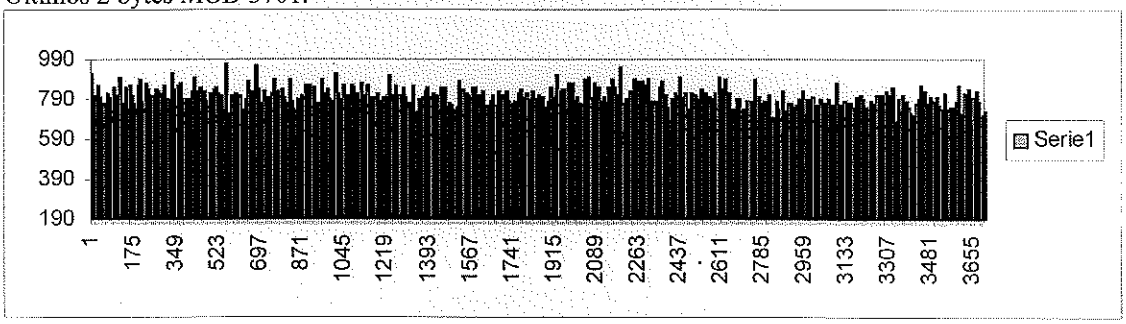

Ultimos 12 bits.




Primeros 2 bytes MOD 3701.




Ultimos 2 bytes MOD 3701.



**MD2.**

MD2 es otra función de hashing unidireccional que devuelve un valor de 128 bits diseñada por Ron Rivest. La seguridad de MD2 depende de una permutación aleatoria de bytes, la permutación es fija y depende de lo dígitos de π. s0,s1,...,S255 es la permutación.

Para hashear un mensaje M.

1) Se completa el mensaje con i bytes de valor i de forma tal que la longitud total del mensaje en bytes sea múltiplo de 16.
2) Se agrega un checksum de 16 bytes al mensaje.
3) Se inicializa un bloque de 48 bytes x0..x47. Los primeros 16 bytes de x se ponen en cero, los siguientes 16 bytes de x son los primeros 16 bytes del mensaje, y el tercer grupo de 16 bytes de x es el xor entre los primeros 16 bytes de x y los segundos 16 bytes de x.
4) Luego se aplica esta función:

```
t=0
for j=0 to 17
      for k=0 to 47
            t=Xk XOR St
            Xk=t
      t=(t+j) mod 256
```

5) Los segundos 16 bytes de X se setean con los segundos 16 bytes del mensaje y el tercer grupo de 16 bytes de X se asigna al XOR de los primeros 16 bytes de X con los segundos 16 bytes de X.

Los pasos 4 y 5 se repiten hasta que no queden mas bloques de 16 bytes por procesar en el mensaje.

El resultado son los primeros 16 bytes de X.

No se ha encontrado ninguna debilidad en el algoritmo MD2, sin embargo este algoritmo es mas lento que otras funciones de hashing unidireccionales.

## SHA. (Secure Hash Algorithm)

La agencia de seguridad nacional de Estados Unidos (NSA) diseño este algoritmo para ser utilizado en el standard de firmas digitales (DSS).

La descripción del algoritmo según la NSA es la siguiente:

"A Federal Information Processing standard (FIPS) for Secure Hash Standard (SHS) is being proposed. This proposed standard specified a Secure Hash Algorithm (SHA) for use with de Digital Signature Standard (DSS)... Additionally for applications not requiring a digital signature yje SHA is to be used whenever a secure hash algorithm is required for Federal applications."

SHA esta basado en MD4 y MD5 aunque con variantes, la principal diferencia reside en que SHA devuelve un valor de 160 bits mientras que MD5 devuelve valores de 128 bits.

Descripción del algoritmo.

En primer lugar el mensaje es expandido de la misma forma en que se hacia en MD5, se agrega un bit en uno y luego tantos bits en 0 como sean necesarios para que a la longitud le falten 64 bits para ser un múltiplo de 512, luego se agrega una representación binaria de 64 bits de la longitud original del mensaje con lo cual el mensaje puede dividirse en un numero entero de bloques de 512 bits de longitud.

Luego se inicializan cinco variables de 32 bits. (MD5 usaba 4)

```
A=0x67452301
B=0xefcdab89
C=0x98badcfe
D=0x10325476
E=0xc3d2e1f0
```

Luego comienza el ciclo principal del algoritmo, las cinco variables se copian en a,b,c,d y e respectivamente. El ciclo principal tiene 4 vueltas de 20 operaciones cada una. En cada operación se efectúa una función no-lineal sobre tres variables, luego un shift y una suma en forma similar al algoritmo MD5.

Las funciones de SHA son:

```
ft(X,Y,Z)=(X^Y)v((-X)^Z) para t=0..19
ft(X,Y,Z)=X⊕Y⊕Z para t=20..39
ft(X,Y,Z)=(X^Y)v(X^Z)v(Y^Z) para t =40..59
ft(X,Y,Z)=X⊕Y⊕Z para t=60..79
```

El algoritmo usa cuatro constantes

```
Kt=0x5a827999 para t=0..19
Kt=0x6ed9eba1 para t=20..39
Kt=0x8f1bbcdc para t=40..59
Kt=0xca62c1d6 para t=60..79
```

Cada bloque del mensaje se transforma de 16 bloques de 32 bits (M0..M15) en 80 bloques de 32 bits (W0..W79) usando el siguiente algoritmo:

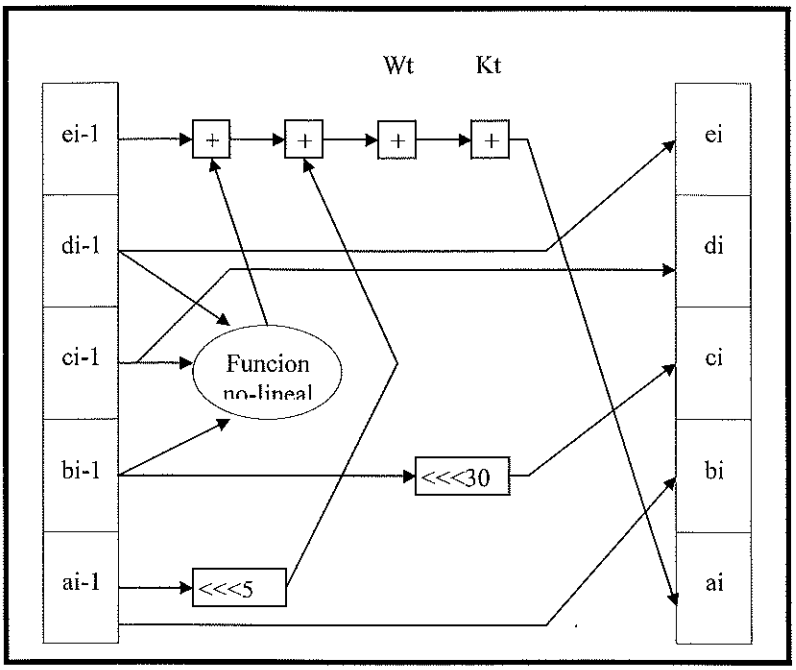
Wt=Mt para t=0..15
Wt=(Wt-3⊕Wt-8⊕Wt-14⊕Wt-16)<<<1 para t=16..79

Como nota interesante podemos mencionar que el algoritmo SHA original no tenia la rotación hacia la izquierda. El cambio según la NSA permite corregir un defecto técnico que hacia al algoritmo menos seguro. El motivo por el cual la rotación influye en la seguridad del algoritmo no fue revelado.

SHA realiza 80 pasos (4 vueltas de 20 operaciones cada una). El funcionamiento del algoritmo es entonces:

```
FOR t=0 to 79
      TEMP=(a<<<5)+ft(b,c,d)+e+Wt+Kt
      e=d
      d=c
      c=b<<<30
      b=a
      a=temp
```

Luego de esto a,b,c,d y e se suman a A,B,C,D y E respectivamente y el algoritmo pasa a procesar el próximo bloque. El resultado final es la concatenación de A,B,C,D,E.




Esquema de uno de los 80 pasos de SHA.

Comparación entre MD5 y SHA.

- Ambos tienen 4 vueltas, sin embargo en SHA la cuarta vuelta usa la misma función f que se usa en la segunda vuelta.
- En MD5 cada paso tiene una constante aditiva única, en SHA las constantes se reusan para cada grupo de 20 pasos.
- La función G en la segunda vuelta de SHA es ((X^Y)v(X^Z)v(Y^Z)) mientras que MD5 usa ((X^Y)v(Y^-(Z))) que es una función menos simétrica. La versión usada en SHA es la que e usaba en MD4.
- En SHA se agrega una quinta variable a la cual se suma el resultado de la función f, en lugar de sumarlo a una de las variables ya usadas en la función, este pequeño cambio en SHA anula el ataque de Boer-Bosselaers usado contra el MD5.
- En MD5 las rotaciones usadas en cada vuelta son distintos, en SHA usa una constante fija para las rotaciones siendo dicha constante relativamente prima con el tamaño de los bloques.
- SHA devuelve valores de 160 bits por lo que es mucho mas resistente a ataques por fuerza bruta que MD5.

No existen ataques exitosos contra SHA hasta la fecha.

**SNEFRU.**

SNEFRU de Ralph Merkle es una función de hashing que devuelve valores de 128 o 256 bits. SNEFRU esta basada en una función de pseudoaleatorizacion, la seguridad de SNEFRU reside en dicha función, se realizaron varios ataques exitosos contra SNEFRU hasta la fecha por lo que su autor recomienda usar la función en al menos 8 pasadas con lo cual el algoritmo se vuelve mucho mas lento que MD5 o SHA.

**RIPE-MD.**

RIPE-MD es un desarrollo de la comunidad europea para el proyecto RIPE. El algoritmo es una variante de MD4 que produce valores de 128 bits. Las rotaciones y el orden de los bloques del mensaje son modificados. Además dos instancias del algoritmo que difieren únicamente en las constantes utilizadas corren en paralelo luego de cada bloque la salida de ambos algoritmos se suma a las variables de encadenamiento. Esta característica vuelve al algoritmo altamente resistente a los ataques mas comunes.

**HAVAL.**

HAVAL es una modificación de MD5, procesa al mensaje en bloques de 1024 bits, tiene 8 variables de encadenamiento. Usa una cantidad de vueltas variable entre 3 y 5, cada vuelta tiene 16 pasos y produce valores de 128, 160, 192, 224, o 256 bits.
HAVAL reemplaza las funciones no-lineales simples de MD5 con funciones altamente no-lineales de 7 variables. Cada vuelta usa una única función pero en cada paso una permutación diferente se aplica a los valores de input. Cada paso usa a su vez una constante aditiva diferente el algoritmo usa también dos rotaciones.

El núcleo de HAVAL es:

```
TEMP=(f(j,A,B,C,D,E,F,G)<<<7)+(H<<<11)+M[i][r(j)]+K(j)
H=G
G=F
F=E
E=D
D=C
C=B
B=A
A=TEMP
```

La cantidad de vueltas variable y la longitud de salida variable hacen que existan 15 versiones de este algoritmo. El ataque de Den Boer y Bosselaers no se puede aplicar a HAVAL debido a la rotación de H.

**Funciones de hashing unidireccionales usando algoritmos de encriptación.**

Es posible utilizar algoritmos de encriptación para construir funciones de hashing unidireccionales, la idea es que si el algoritmo de encriptación es seguro la función de hashing también lo será.

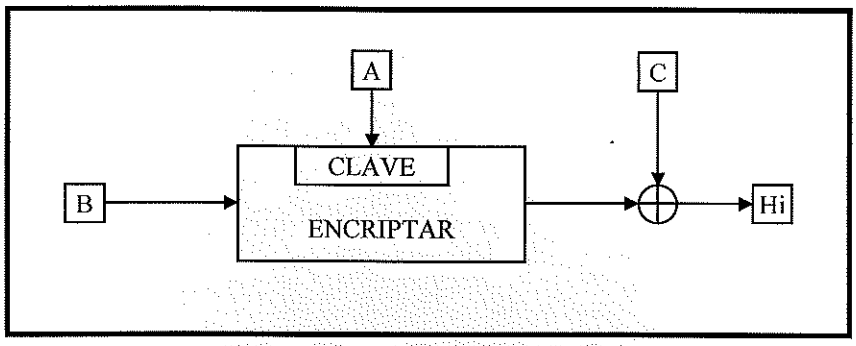
Una de las aproximaciones consiste en usar el bloque del mensaje como clave, y el valor de hash previo como input del algoritmo de encriptación. Repitiendo este proceso hasta que se usa el ultimo bloque como clave. La salida final es el resultado de la función de hashing.

Esquemas en los cuales la longitud de salida es igual al tamaño del bloque:

El esquema general es:

H0 = Ih (Donde Ih es un valor inicial aleatorio)
Hi = Ea(B) ⊕ C

En donde A,B y C pueden ser:
Mi, Hi-1, (Mi⊕Hi-1) o una constante.




Función de hashing genérica usando encriptación.

A,B y C pueden entonces tomar uno de 4 valores posibles por lo que existen en total 64 posibles funciones de hashing para cada algoritmo de encriptación. Suponiendo que el algoritmo de encriptación es seguro lo que queda es estudiar la seguridad de la función de hashing según que valores toman A,B y C. Bart Preneel estudio todas las posibles combinaciones arribando a resultados interesantes: 15 son triviales porque el resultado no depende de uno de los inputs. Mientras que otras 37 combinaciones son inseguras por razones mas complejas. Restan por lo tanto 12 combinaciones seguras.

| *Funciones de Hashing Seguras Usando Encriptación.* | | |
|---|---|---|
| A | B | C |
| Hi-1 | Mi | Mi |
| Hi-1 | Mi⊕Hi-1 | Mi⊕Hi-1 |
| Hi-1 | Mi | Hi-1⊕Mi |
| Hi-1 | Mi⊕Hi-1 | Mi |
| Mi | Hi-1 | Hi-1 |
| Mi | Mi⊕Hi-1 | Mi⊕Hi-1 |
| Mi | Hi-1 | Mi⊕Hi-1 |
| Mi | Mi⊕Hi-1 | Hi-1 |
| Mi⊕Hi-1 | Mi | Mi |
| Mi⊕Hi-1 | Hi-1 | Hi-1 |
| Mi⊕Hi-1 | Mi | Hi-1 |
| Mi⊕Hi-1 | Hi-1 | Mi |

Las primeras 4 combinaciones son seguras contra todo tipo de ataques mientras que las ultimas 8 son seguras excepto contra ataques muy específicos que no comprometen en absoluto la seguridad del algoritmo.

El tercer esquema fue propuesto como un standard ISO. El quinto esquema propuesto por Carl Meyer se denomina en general esquema Davies-Meyer. La décima combinación es usada como modelo para la función LOKI.

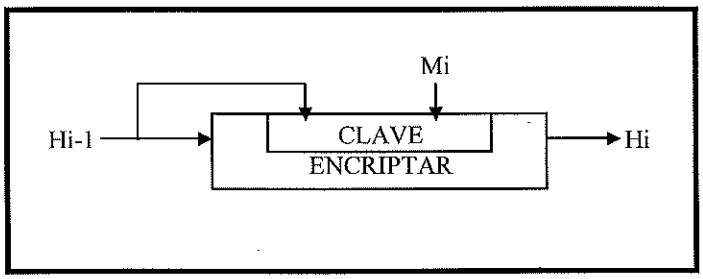
En todas las combinaciones la longitud del bloque es igual a la longitud de salida de la función de hashing sin embargo, la longitud de la clave puede variar. En las combinaciones 1,2,3,4,9 y 11 la longitud de la clave es igual a la longitud del bloque. En las demás combinaciones la longitud de la clave puede variar.

Otros esquemas propuestos en distintas publicaciones o algoritmos se han demostrado inseguros.

**Esquema Davies-Meyer modificado.**

Lai y Massey modificaron el esquema Davies-Meyer para poder usar como algoritmo de encriptación IDEA. IDEA trabaja con claves de 128 bits y bloques de 64 bits. El esquema es:

H0=Ih (IH es un valor random inicial)
Hi=Ehi-1|Mi(Hi-1)




Esquema Davies-Meyer modificado (devuelve 64 bits)

**Esquemas en los cuales la longitud del bloque es distinta a la longitud de salida.**

**Esquema Preneel-Bosselaers-Govarts-Vandewalle.**

Esta es una función de hashing que produce un valor del doble de longitud que el bloque usado por el algoritmo de encriptación. Un algoritmo de 64 bits produce un valor de 128 bits.

G0=Ig (Ig es random)
Ho=Ih (Ih es random)
$Gi=E_{li \text{ xor } Hi-1}$ (Ri xor Gi-1) xor Ri xor Gi-1 xor Hi-1
$Hi=E_{li \text{ xor } Ri}$ (Hi-1 xor Gi-1) xor Li xor Gi-1 xor Hi-1

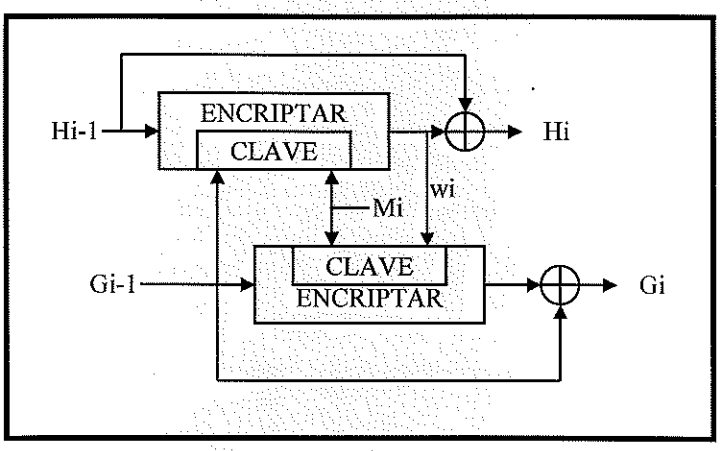
El resultado es la concatenación de Gi con Hi cuando ya no quedan mas bloques por procesar. Este algoritmo no es seguro y hay varios métodos especializados para encontrar sinónimos.

**Tandem Davies-Meyer.**

En este esquema dos esquemas Davies-Meyer modificados funcionan en tandem para obtener un valor de 128 bits usando bloques de 64 bits y claves de 128 bits.

G0=Ig (Ig random)
H0=Ih (Ih random)
Wi=Egi-1|Mi(Hi-1)
Gi=Gi-1⊕Emi|wi(Gi-1)
Hi=Wi⊕Hi-1

Los valores de Gi y Hi se concatenan generando un valor de 128 bits.




Tandem Davies-Meyer.

Hasta el momento no hay ataques efectivos contra este esquema a excepción de los de fuerza bruta, encontrar sinónimos para este tipo de función implica probar al menos 2^64 combinaciones.

Conclusiones:

Muchas funciones de hashing unidireccionales han sido construidas hasta la fecha, la forma en la que cada una de ellas distribuye las claves es dependiente de la función, en cuanto a seguridad las funciones mas probadas son SHA y MD5, de ambas SHA devuelve valores de longitud superior a MD5 por lo que es considerada mas segura, aunque en general tanto MD5 como SHA son muy utilizadas y se supone hasta el momento que ambas funciones son sumamente seguras.

**Elección de una función de hashing.**

Muchas consideraciones deben hacerse al elegir una función de hashing, en primer lugar y si es posible hay que hacer un análisis detallado de los datos que se deben hashear y por que motivos, cuales resultados son buenos y cuales malos y consideraciones sobre como debería trabajar la función para que sea eficiente, ¿se requiere una distribución muy exigente o se necesita que la función sea muy rápida? ¿es necesario que la función sea unidireccional o puede ser una función de hashing tradicional? ¿puede variar el conjunto de datos o es fijo?. En muchos casos cuando no es posible analizar el conjunto de datos debe elegirse una o varias funciones de hashing genéricas para finalmente de entre todas las candidatas seleccionar la que mejor rendimiento tuvo para el conjunto de datos con el cual se trabaje. Varias de las funciones presentadas en este apunte pueden usarse con buenos resultados en forma genérica, de las funciones no-unidireccionales las funciones tipo fold&add y otras como krhash tienden a distribuir en forma pareja casi cualquier conjunto de claves, si se requiere seguridad extra puede usarse una función unidireccional de las mas testeadas como por ejemplo MD5 o SHA, este tipo de funciones es bueno si la longitud de las claves a hashear supera los 1024 bytes, en caso contrario una función tradicional es mas rápida y puede llegar a ser igual de eficiente. Lo importante es no olvidar que el éxito o fracaso de la función depende de dos cosas: de la función y del conjunto de datos para el cual se use.