Filtros de Bloom y Cryptohashing

Omar López Rubio, Marc Ángel Ortiz, David Williams

Abril 2017

1 Introducción

Un filtro de Bloom es una estructura de datos probabilística que nos permite determinar si un elemento pertenece o no a un conjunto. Uno de los inconvenientes que tiene es que, por culpa de las colisiones en la función de hash, podríamos encontrarnos con falsos positivos. El objetivo de este trabajo ha sido hacer una análisis de los falsos positivos obtenidos utilizando diferentes métodos de representación para el conjunto de claves, y observar así si tenemos alguna diferencia significativa.

2 Implementación

2.1 Filtro de Bloom

Hemos basado nuestra implementación del filtro en un diseño que encontramos en GitHub (1) pero lo adaptamos de manera que también se generara el filtro dado su tamaño. En la implementación original solo se pedía como parámetro el error que se quería obtener; esto es, una vez elegido el número de claves y su longitud, se calculaban unos valores m y k dado el error teórico que se quería conseguir, utilizando la fórmula $k = \frac{m}{n} \ln 2$. En la modificación que hemos hecho, podemos observar el error teórico pero dando como parámetro el tamaño del filtro, cosa que nos pareció útil para la experimentación.

2.2 Funciones de hash

Como función principal queríamos utilizar MurmurHash3, pero debido a problemas de compatibilidad con nuestras máquines tuvimos que utilizar MurmurHash2, una versión que sólo genera hashes de 32 o 64 bits. Esta función es conocida por ser bastante rápida y ofrecer buena resistencia a colisiones, como explica el creador en su página (2). Una vez conocidas las k funciones de hash que necesitamos, utilizamos la técnica de double hashing para ir creando las nuevas.

Por falta de tiempo no hemos podido experimentar con CityHash, una familia de funciones no criptográficas mantenidas por ingenieros de Google, que presume de mejorar la performance de MurmurHash3 (3).

2.3 Generador de Strings

Para automatizar la generación de claves para los experimentos, hemos añadido también un pequeño programa que crea aleatoriamente n strings de una longitud determinada.

2.4 Funciones Criptográficas

Como funciones criptográficas hemos utilizado SHA256 tal y como decía el enunciado, de la cuál no se conocen aún colisiones (4) y también MD5 (5), que no es tan robusta en este aspecto (6) pero por construcción tiene un coste de cálculo menor. Esta decisión nos permitirá ver si el tiempo invertido en hashear las claves con SHA está justificado en la reducción de falsos positivos.

2.5 Programa principal y tests

En el programa principal preguntamos al usuario el número de claves que desea y fijamos su longitud. Estas claves son creadas utilizando el generador de strings. Después de esto, pedimos el tamaño de filtro que desea o el error teórico, llamando a las funciones del filtro necesarias para crearlo. Finalmente, preguntamos si no se quiere utilizar encriptación o si se quiere usar SHA256 o MD5, y una vez lo sabemos, vamos introduciendo las claves encriptadas o no, dependiendo del caso. Una vez inicializado el filtro con nuestras claves, utilizamos los 10 ficheros de test como input para calcular los falsos positivos, de manera que tenemos una tabla de hash en la que comprobamos si el elemento realmente pertenece al filtro o no, siendo en este caso incrementado el contador de falsos positivos. Con llamadas a la función clock() de la librería estándar time.h calculamos el coste de aplicar las diferentes funciones criptográficas e insertarlas en el filtro, y también el tiempo necesario para hacer todas las querys.

Cabe mencionar que, en los ficheros de test, hemos utilizamos diferentes criterios; por ejemplo, en algunos test utilizamos carácteres alfanuméricos aleatorios, en otros hemos utilizado un subconjunto de contraseñas del fichero rockyou.txt (7) el cuál contiene una selección de contraseñas frecuentes. Esto nos pareció interesante al empezar el trabajo ya que nos planteamos que, debido a los patrones que utilizamos regularmente en el lenguaje, nos encontráramos con una eficacia superior al utilizar métodos criptográficos ya que el hasheo de texto, llamémosle "normal", siguiera también unos patrones similares, y tomamos esto como hipótesis.

3 Experimentos

Para cada valor de la gráfica hemos hecho 5 ejecuciones diferentes, quedándonos con la mediana. Primero mostraremos los resultados para el conjunto de claves generadas pseudoaleatoriamente, para un error teórico de 0.5 utilizando nuestros tests y $k = 2$.

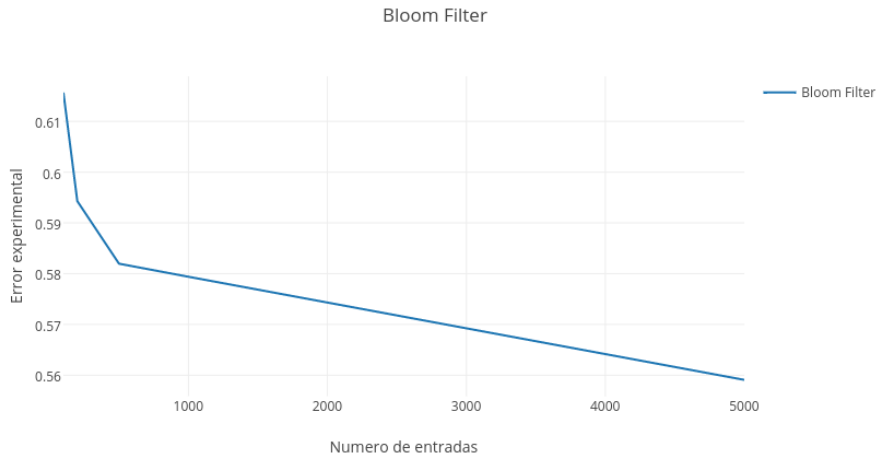

Figure 1: Error experimental contra el número de entradas testeadas pseudoaleatorias

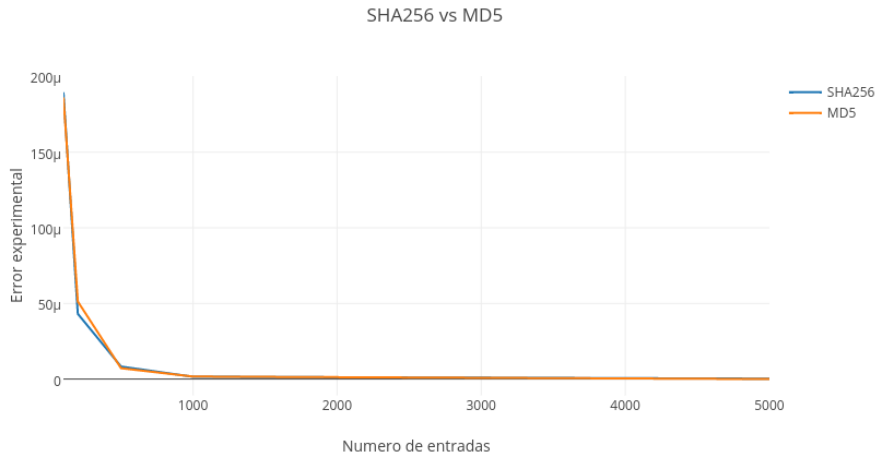

Figure 2: Error experimental aplicando SHA256/MD5 contra el número de entradas pseudoaleatorias testeadas

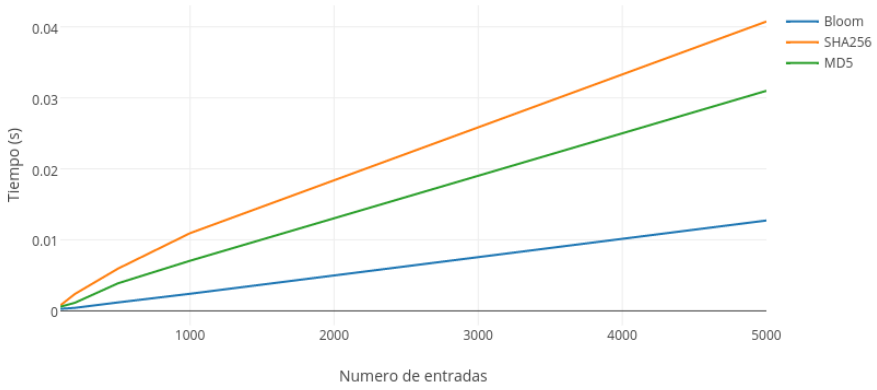

Figure 3: Tiempo para añadir las claves pseudaleatorias al filtro contra el número de entradas

En la Figura 1 podemos apreciar que el valor experimental va tendiendo hacia el teórico a medida que el tamaño de los valores almacenados en el filtro va creciendo.

En la Figura 2 vemos que los falsos positivos han decrecido drásticamente, lo que parece prometedor, y vemos como ambos se van aproximando al 0. No encontramos ninguna diferencia significativa entre SHA256 y MD5 respecto a los falsos positivos; ambas presentan una mejoría bastante grande.

Vemos también en la Figura 3 como tardamos más en aplicar SHA256 que MD5. Como ya hemos comentado anteriormente, el cálculo es ligeramente más complejo.

Pasamos ahora a utilizar 5000 contraseñas aleatorias del fichero *test*8.*txt* para ver si encontramos alguna diferencia remarcable. El error teórico sigue siendo de 0.5 y $k = 2$.

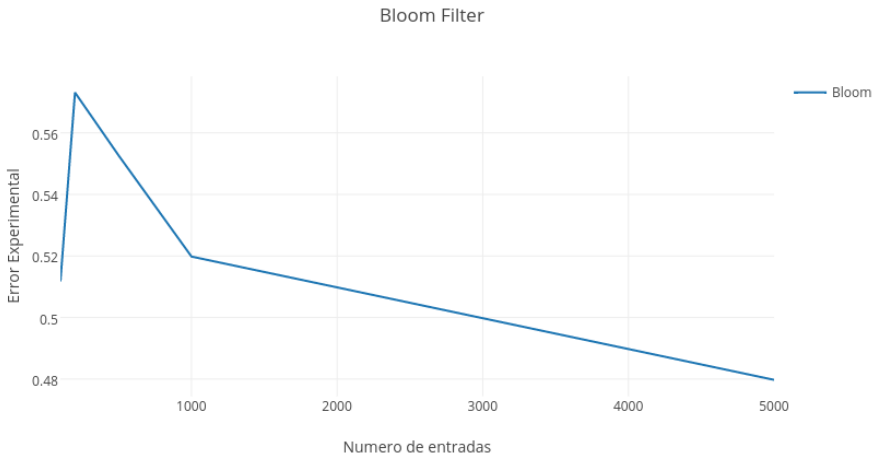

Figure 4: Error experimental contra el número de entradas testeadas para *test*8.*txt*

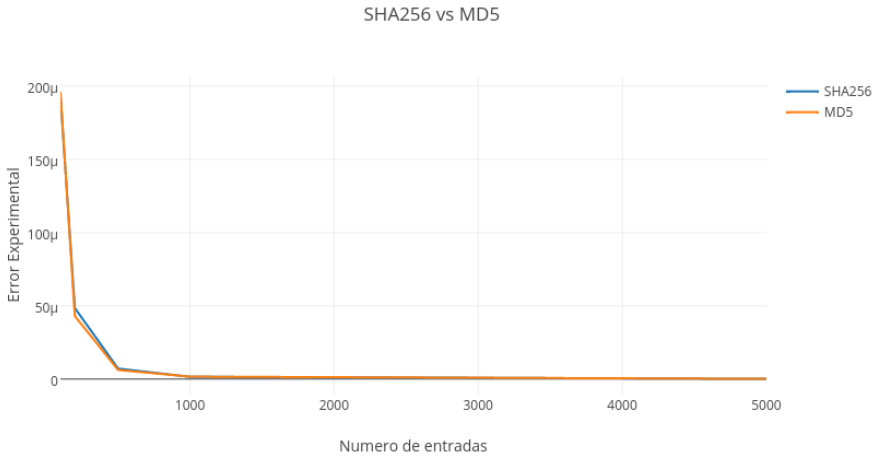

Figure 5: Error experimental aplicando SHA256/MD5 contra el número de entradas testeadas para *test*8.*txt*

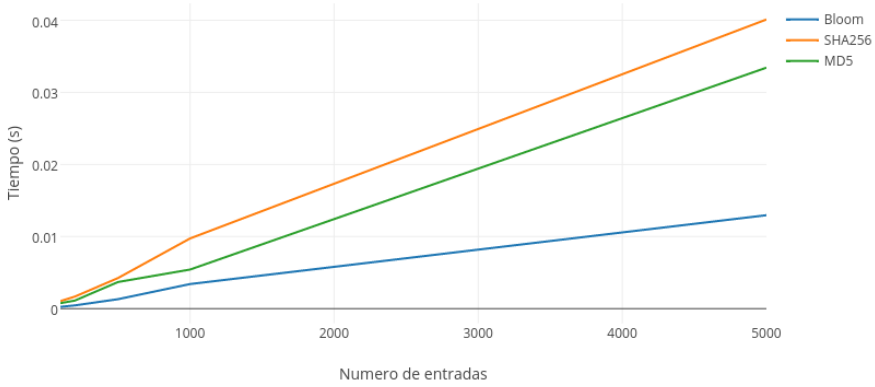

Figure 6: Tiempo para añadir las claves al filtro contra el número de entradas para *test*8.*txt*

En la Figura 4 vemos como se sigue respetando bastante el error teórico aunque hayamos cambiado el conjunto de claves. Los tiempos siguen siendo bastante parecidos, como se aprecia en la Figura 5, y no vemos ningún valor claramente remarcable. Finalmente, en la Figura 6 se ve que, lógicamente, el coste de la encriptación también se respeta.

4 Conclusiones

Nos falló la intuición de que, al usar claves pseudaleatorias o claves "normales", veríamos algo remarcable en los experimentos. Parece ser que, debido al efecto avalancha (8) de las funciones usadas, es indiferente utilizar un conjunto de claves o otro; pero el uso de SHA256/MD5 nos ha dado una mejora significativa en los experimentos, donde claramente ha ganado MD5 por su sencillez y velocidad. Entonces, dependerá de para qué queramos utilizar el filtro, ya que el encriptar las claves con MD5 nos reducirá los falsos positivos, pero tardaremos algo más en calcularlo.

Referencias

[1] `https://github.com/jvirkki/libbloom`

[2] `https://sites.google.com/site/murmurhash/`

[3] `https://opensource.googleblog.com/2011/04/introducing-cityhash.html`

[4] `http://stackoverflow.com/questions/4014090/is-it-safe-to-ignore-the-possibility-of-sha-collisions-in-practice`

[5] `http://www.zedwood.com/article/cpp-md5-function`

[6] `http://www.mscs.dal.ca/ selinger/md5collision/`

[7] `https://wiki.skullsecurity.org/Passwords`

[8] `https://en.wikipedia.org/wiki/Avalanche`$_{e}ffect$