Combining Chunk Boundary and Chunk Signature Calculations for Deduplication

W. Litwin, D. D. E. Long, *Fellow, IEEE*, Thomas Schwarz, *Senior Member, IEEE*

***Abstract*— Many modern, large-scale storage solutions offer deduplication, which can achieve impressive compression rates for many loads, especially for backups. When accepting new data for storage, deduplication checks whether parts of the data is already stored. If this is the case, then the system does not store that part of the new data but replaces it with a reference to the location where the data already resides. A typical deduplication system breaks data into chunks, hashes each chunk, and uses an index to see whether the chunk has already been stored. Variable chunk systems offer better compression, but process data byte-for-byte twice, first to calculate the chunk boundaries and then to calculate the hash. This limits the ingress bandwidth of a system. We propose a method to reuse the chunk boundary calculations in order to strengthen the collision resistance of the hash, allowing us to use a faster hashing method with fewer bytes or a much larger (256 times by adding two bytes) storage system with the same high assurance against chunk collision and resulting data loss.**

***Keywords*— Deduplication, Algebraic Signatures.**

I. Introducción

LA DEDUPLICACIÓN es una estrategia cada vez más popular para comprimir datos en un sistema de almacenamiento para la identificación y eliminación de datos duplicados. La deduplicación en línea haciendo copias de seguridad puede tener proporciones impresionantes de compresión (Zhu, Li, y Patterson [1] reportaron proporciones de hasta 20:1, y Mandagere, Zhou, Smith, y Uttamchandi [2] hasta 30:1). Es idónea para sistemas con peta-bytes.

La identificación en línea de datos duplicados en un sistema que ya almacena peta-bytes de datos es difícil. La deduplicación basada en archivos solamente identifica archivos duplicados, pero pierde oportunidades de deduplicación cuando un archivo es almacenado solamente con cambios pequeños. Además, muchos sistemas no almacenan archivos sino flujos, por ejemplo cuando se toma una copia de seguridad de un sistema completo. En vez de procesar archivos enteros, se corta el archivo o flujo en *trozos* (chunks). La utilización de trozos de tamaño fijo pierde oportunidades para deduplicar cuando un nuevo archivo es una versión de un archivo ya almacenado con pequeños cambios
como inserciones o eliminaciones. La definición de trozos por contenidos (*Content-based chunking* [5]) utiliza información local para definir las fronteras entre los trozos en un modo independiente de alteraciones en trozos vecinos. En general, los cambios locales en un archivo largo no cambian la mayoría de los trozos. El sistema puede reconocer estos trozos y evitar almacenarles otra vez. El cálculo de los límites de los trozos se realiza procesando un archivo de entrada byte por byte. Después de calcular los límites de los trozos, el sistema de deduplicación calcula un hash de los trozos y luego busca (y almacena) esos valores en un índice de todos los hashes de trozos ya vistos. Si se puede encontrar el mismo valor del hash, se concluye que el trozo ya está almacenado y no se almacena otra vez. Para obtener una tasa aceptable para el ingreso de datos, la deduplicación usualmente no verifica la identidad de trozos por una comparación byte por byte. Si hay una colisión de hashes – dos trozos diferentes tienen el mismo valor del hash – entonces el sistema ha perdido los datos más actuales. La tasa de pérdida de datos causada por estas colisiones debe ser menor a la tasa de perdida por otras razones, como falla de un disco, para que la deduplicación sea aceptable. Podemos controlar la probabilidad de colisiones de hash utilizando una función buena de hash, por ejemplo substituyendo a MD5 (128b) por SHA-1 (160b) o utilizando SHA-256 con 256 bits.

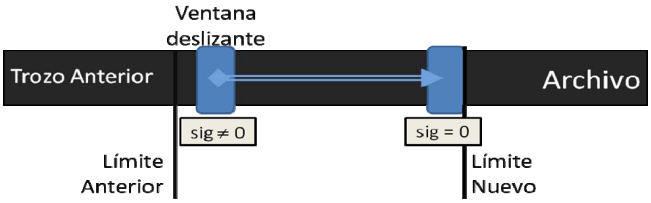

Figura 1. Método de ventana deslizante

Para la deduplicación, hay dos cuellos de botella, el del I/O por la búsqueda en el índice y el mantenimiento del mismo, y el cálculo de límites y hashs de trozos. Por eso, una función de hash más rápida resulta más atractiva.

Los límites de trozos deben ser independientes de alteraciones al flujo o archivo a gran distancia del trozo actual. Se ha adoptado el método de ventana deslizante [5] (Fig. 1). Este método calcula una función de los bytes en una ventana pequeña y define un límite entre trozos si el valor de esa función es igual a cero. Nuestra contribución es la reutilización de ese cálculo para obtener bytes adicionales por el hash y de este modo, aumentar la resistencia a colisiones sin gastos adicionales de desempeño. También es posible utilizar una función de hash con menos bytes pero más rápida. Nuestro trabajo puede ser incorporado con métodos actuales

W. Litwin, Centre d'Etude y Recherche en Informatique Appliqué, Université Paris Dauphine, Paris, France. Witold.Litwin@dauphine.fr

D. Long, Computer Science Department, University of California at Santa Cruz, Santa Cruz, CA, USA. darrell@cs.ucsc.edu

T. Schwarz, Dto. Informática y Ciencias de la Computación, Av. 8 de Octubre, Universidad Católica del Uruguay, Montevideo, Uruguay. tschwarz@ucu.edu.uy

con una pequeña excepción que discutimos a continuación.

El método genérico de determinación de trozos es probabilístico y puede resultar en trozos muy largos o muy pequeños. Eshgi y Tang [7] utilizan un método adicional para evitar esos trozos mal dimensionados. Para prevenir trozos demasiados chicos, se ignoran los límites posibles encontrados a poca distancia del comienzo de un trozo. Para evitar trozos demasiados grandes, se utiliza un método alternativo para calcular los límites. Si el trozo actual es demasiado grande, se utiliza un lugar donde el segundo método determina un límite. Para incorporar nuestra contribución con estos métodos, perdimos el ahorro de no evaluar una ventana cerca al comienzo del trozo actual, pero ese es el único punto negativo.

La seguridad del sistema depende de la seguridad de la función de hash utilizada. Un adversario con acceso al sistema de almacenamiento puede comenzar un ataque de una colisión deseada [8] encontrando una colisión con un trozo para ser insertado en el futuro. Por ejemplo, si el sistema de almacenamiento guarda imágenes del sistema de archivos y el adversario ya conoce una actualización, los contenidos del sistema cambiarán en un modo determinado. El adversario puede encontrar un trozo en el sistema actualizado colisionando con un trozo de su construcción con el mismo hash e introducir este trozo en el sistema de almacenamiento. Cuando el sistema de la víctima es actualizado y la copia de seguridad se hace en el sistema de almacenamiento, el sistema silenciosamente cambia el trozo por el trozo del adversario. Después de reiniciar el sistema de la víctima con la copia de seguridad, el adversario habrá introducido exitosamente malware. Se puede defender de ese ataque de varios modos [8], pero lo más fácil sería substituyendo a un hash fijo y bien conocido como MD5 por un hash con clave utilizado como código de autenticación de mensaje (HMAC). En este artículo, nos preocupamos de la prevención de colisiones causadas estadísticamente por el gran número de trozos en el sistema.

En lo que sigue, explicamos la matemática del cálculo de límites de trozos orientándonos en el desempeño (Sección II), después presentamos nuestro método combinado (Sección III), y finalmente evaluamos la calidad de los bytes agregados al hash de trozos y la mejora en la resistencia a colisiones (Sección IV).

II. Cálculo de Límites de Trozos

Se determinan los límites de los trozos por el cálculo de una función de una ventana pequeña (ej. 4 – 10 bytes) y colocando límites si la función tiene un valor en un conjunto específico. En general, se utiliza un *rolling hash* donde el valor de la función de la ventana movida por un byte a la derecha es calculado del valor de la ventana original, del byte a la izquierda que acaba de salir de la ventana, y el byte a la derecha que acaba de entrar en la ventana. Hay varias posibilidades de definir un rolling hash. En este artículo, usamos una variante de *firmas algebraicas* [9] pero también presentamos los Rabin Fingerprints [10] para mostrar que nuestra elección incurre la misma carga computacional.

A. Rabin Fingerprint

El método de Rabin Fingerprints [10, 11] asocia un polinomio a una cadena de bits $P = (p_0, p_1, ..., p_{m-1})$ por

$$p_P(t) = p_0 t^{m-1} + p_1 t^{m-2} + ... + p_{m-2} t + p_{m-1}$$

y después utiliza un polinomio fijo e irreducible $g(t)$ sobre F_2, es decir, un polinomio con coeficientes 0 o 1, para definir el Rabin fingerprint

$$R(P) = p_P \pmod g$$

Como es usual, identificamos un polinomio $p_P(t)$ con la cadena de bits P donde, si es necesario, llenamos con ceros para que P entre en bytes o palabras. El número de los bits en un Rabin fingerprint es dado por el grado de $g(t)$. Por ejemplo, un fingerprint de cuatro bytes necesita un g de grado 32.

Asumimos a continuación una ventana deslizante de w bytes y una cadena de bytes $P = (p_0, p_1, ...)$. Primero, convertimos P en una cadena de bits. También identificamos el byte p_i con el polinomio

$$p_i(t) = p_{i,0} t^7 + p_{i,1} t^6 + ... + p_{i,6} t + p_{i,7}$$

donde $p_i = [p_{i,0}, p_{i,1}, ..., p_{i,6}, p_{i,7}]$ es la descomposición del byte p_i en ocho bits. Ahora calculamos mod $g(t)$:

$$\begin{aligned} & R(p_{r+1}, p_{r+2}, ..., p_{r+w}) \cdot t^{-8} \\ &= p_{r+1}(t) t^{8(w-2)} + p_{r+2}(t) t^{8(w-3)} + ... + p_{r+w}(t) t^{-8} \\ &= p_r(t) t^{8(w-1)} + p_r(t) t^{8(w-1)} + p_{r+1}(t) t^{8(w-2)} + ... + p_{r+w}(t) t^{-8} \\ &= p_r(t) t^{8(w-1)} + R(p_r, p_{r+1}, p_{r+2}, ..., p_{r+w-1}) + p_{r+w}(t) t^{-8} \end{aligned}$$

La segunda ecuación utiliza que la adición de los coeficientes es la operación de disyunción exclusiva (XOR). Utilizaremos el símbolo "+" para denotar esa operación ya que es la adición en F_2. Como consecuencia, $p_r(t) t^{8(w-1)} + p_r(t) t^{8(w-1)} = 0$. Nuestro cálculo brinda la regla de transformación para ventanas deslizantes:

$$\begin{aligned} & R(p_{r+1}, p_{r+2}, ..., p_{r+w}) \\ &= p_r(t) t^{8w} + R(p_r, p_{r+1}, p_{r+2}, ..., p_{r+w-1}) + p_{r+w}(t) \end{aligned}$$

Esa regla de transformación se utiliza para calcular la firma de la ventana deslizante. La implementación más fácil utiliza dos tablas para la multiplicación con t^8 y t^{8w}. El método eficiente de Broder [11] calcula el Rabin fingerprint de un objeto procesando varios símbolos al mismo tiempo, pero claramente, no podemos utilizar ese método en nuestro contexto.

Un modo alternativo del cálculo de un Rabin fingerprint de una ventana deslizante mantiene el fingerprint $R(p_0, ..., p_m)$ del trozo visto actualmente. Se calcula el valor del trozo hasta el byte n por

$$R(p_0, p_1, ..., p_n) = R(p_0, p_1, ..., p_{n-1}) \cdot t^8 + p_n(t)$$

El valor para la ventana deslizante es entonces:

$$R(p_r, ..., p_{r+w-1}) = R(p_0, ..., p_{r+w-1}) \cdot t^{8w} + R(p_0, ..., p_{r-1})$$

donde otra vez calculamos mod $g(t)$. En ambos casos, la implementación directa utiliza dos búsquedas en tablas y dos operaciones XOR.

Las tablas son grandes. Si el grado de g es k, hay 2^k valores posibles para el fingerprint y en consecuencia 2^k entradas de

tamaño 2^k bits. Aún para valores pequeños de k, la tabla no cabe en las caches L1 o L2 de un procesador. Para k = 16, el tamaño de la tabla es 131kB. Para ser eficiente, el uso de una sola tabla limitaría el tamaño del fingerprint a dos bytes.

Por suerte, se puede utilizar la linealidad de la multiplicación. Para generalizar, asumimos que queremos implementar la multiplicación mod $g(t)$, un polinomio fijo de grado k. Sea

$$\Re_k = \{a_k t^k + a_{k-1} t^{k-1} + ... a_2 t + a_1\}$$

el conjunto de todos los polinomios con coeficientes binarios hasta grado k–1. Para implementar la multiplicación con un polinomio $f(t) \in \Re_k$ cortamos cada polinomio $a(t) \in \Re_k$ en

$$A_0(t) + A_1(t) t^8 + A_2(t) t^{16} + ...$$

con coeficientes $A_0(t) = a_8 t^7 + ... + a_2 t + a_1$, $A_1(t) = a_{16} t^7 + ... + a_{10} t + a_9$ representando el resultado de cortar la cadena de k bits en bytes. Si es necesario, se llena con ceros al final. Por la ley distributiva de la multiplicación

$$\begin{aligned} f(t)a(t) &= f(t)\left(A_0(t) + A_1(t) t^8 + A_2(t) t^{16} + ...\right) \\ &= f(t) A_0(t) + f(t) t^8 A_1(t) + f(t) t^{16} A_2(t) + ... \end{aligned}$$

Por eso, necesitamos una tabla (con 2^8 entradas de k bits cada una) para implementar la multiplicación (mod $g(t)$) de $f(t)$ y un polinomio en $\Re_8$, y otra tabla para implementar multiplicación con $f(t)t^8$ etc. En conjunto, se necesitan k/8 tablas de tamaño $2^8 k$ bits, con ahorros considerables en espacio, pero también costando k/8–1 operaciones XOR adicionales.

B. Firmas Algebraicas

Firmas algebraicas (Algebraic Signatures) son funciones hash con propiedades algebraicas [9]. Como los hashes criptográficamente seguros (ej. SHA-1, SHA-2, MD5 [12,13]) identifican objetos grandes con una probabilidad muy pequeña de colisiones. En contraste a estos hashes, tienen propiedades algebraicas que dejan calcular la firma de un objeto compuesto de las firmas de sus componentes. Un adversario puede utilizar esa propiedad si una firma algebraica fuera utilizada en lugar de un hash criptográfico, pero aquí lo utilizamos en un modo que no impacta la seguridad. Las firmas algebraicas utilizan la aritmética de campos de Galois que veremos a continuación.

Como los campos más conocidos de los números racionales, reales, o complejos, los campos de Galois tienen adición y multiplicación con las mismas reglas aritméticas. El número de elementos en un campo de Galois es siempre una potencia de un primo, y para cada número, existe solamente un campo. Utilizamos solamente campos de Galois donde el número es una potencia de 2. Escribimos $F(2^f)$ para el único campo de Galois con 2^f elementos. Como la arquitectura de sistemas de almacenamiento está basado en bytes, solamente utilizamos f=8 de aquí en más. Con esa selección, se identifica cada elemento del campo con un byte. Nuestros resultados serán validos si se utiliza un valor diferente de f. Implementamos las operaciones en $F(2^f)$ utilizando la representación de los elementos del campo como cadenas de f bits. La adición es dado por la disyunción exclusiva y el elemento cero es la cadena con dígitos cero solamente.

Para nuestra finalidad, los *elementos primitivos* en campos de Galois son importantes. Un elemento α es primitivo si las potencias α^i, i=0, 1, ... recorren todos los elementos que no son cero del campo. El algebra demuestra que los campos de Galois no solamente tienen elementos primitivos, sino también que son numerosos. Si fijamos un elemento primitivo α, entonces todo otro elemento β, $\beta \neq 0$, es una potencia α^i de α, donde i es un número único entre 0 y 2^f –1. Llamamos i al logaritmo de β y escribimos $i = \log_\alpha(\beta)$ y $\beta = \text{antilog}_\alpha(i)$. Podemos multiplicar dos elementos $\beta, \gamma, \beta \neq 0 \neq \gamma$ por la formula:

$$\beta \cdot \gamma = \text{antilog}_\alpha(\log_\alpha(\beta) + \log_\alpha(\gamma))$$

donde la adición es mod 2^f–1. La implementación de la multiplicación en campos de Galois puede utilizar varios métodos, inclusive la sugerida por la formula anterior, pero la mejor depende de la arquitectura del procesador, tamaño y velocidad de acceso de los caches, y el tamaño del campo [14].

La definición de firmas algebraicas en [9] no es necesariamente la optima para usar como rolling hash. Por eso, definimos una firma algebraica alternativa, la s-sig, que tiene la misma estructura y las mismas propiedades. Es la firma algebraica original de la cadena al revés.

La s-sig consiste de varios componentes, donde cada uno es un byte. Se define utilizando un elemento β del campo Galois. Si $P = (p_0, p_1, ..., p_{n-1})$ es una cadena de bytes, entonces un componente de la s-sig se define como

$$s(\beta, P) := \sum_{\nu=0}^{n-1} p_\nu \beta^{n-\nu-1}$$

La s-sig completa es un vector de dimensión m, donde cada coeficiente es un componente de una s-sig definido por las potencias consecutivas de un elemento primitivo α.

$$s_{\alpha,m}(P) := \left(s(\alpha, P), s(\alpha^2, P), ..., s(\alpha^m, P)\right)$$

Si movemos una ventana por un símbolo a la derecha, actualizamos el componente de la s-sig por:

$$\begin{aligned} &s(\beta, (p_{r+1}, p_{r+2}, ..., p_{r+w})) = \\ &p_{r+w} + s(\beta, (p_r, p_{r+1}, p_{r+2}, ..., p_{r+w-1})) + \beta^w p_r \end{aligned}$$

Como queremos utilizar los s-sig como hash de un trozo completo, combinamos el cálculo de los componentes. Si procesamos un carácter adicional del trozo, actualizamos todos los componentes de la s-sig por

$$s(\beta, (p_0, p_1, ..., p_n)) = \beta \cdot s(\beta, (p_0, p_1, ..., p_{n-1})) + p_n$$

y calculamos la firma de la ventana por

$$\begin{aligned} &s(\beta, (p_{r+1}, p_{r+2}, ..., p_{r+w})) = \\ &s(\beta, (p_0, p_1, p_2, ..., p_{r+w-1})) + \beta^w s(\beta, (p_0, p_1, p_2, ..., p_{r-1})) \end{aligned}$$

Ponemos un límite del trozo si la s-sig de la ventana es cero. Eso se hace comparando dos s-sig compuestos, comenzando al límite izquierdo del trozo actual y terminando justamente antes o al final de la ventana. La condición es

$$s(\beta, (p_0, p_1, ..., p_{r+w-1})) = \beta^w \cdot s(\beta, (p_0, p_1, ..., p_{r-1}))$$

Cada vez que procesamos un nuevo símbolo, actualizamos

la *s*-sig del trozo visto hasta ese momento. Para cada componente de la *s*-sig, el procesamiento cuesta una adición

```
i = 0
for j in range(4):
  oc[j] = (0, 0)
while True:
  c = getNextCharacter(file)
  p1 = mult(a,p1) ^ c
  p2 = mult(a2,p2) ^ c
  (q1, q2) = oc[i]
  if mult(a4, p1), mult(a8, p2) & mask
      == (p1, p2) & mask:
       trigger_chunk_boundary()
oc[i] = (p1, p2)
i = i+1 mod 4
```

Figura 2. Seudo código para el cálculo de limite de trozo

(disyunción exclusiva) y una multiplicación en el campo Galois, implementado por la búsqueda en una tabla. Después probamos si hay un límite del trozo que también cuesta una adición (disyunción exclusiva) y una multiplicación por componente.

En comparación de los gastos del cálculo con *s*-sig y con Rabin fingerprints, obtenemos el mismo número de accesos en tablas y operaciones de disyunción exclusivas. Operativamente, ningún método tiene ventaja. Preferimos los *s*-sig porque podemos probar su idoneidad como funciones de hash. El argumento principal de este artículo (el cálculo de límites de trozos puede ser reutilizado para obtener bytes adicionales de los hash de los trozos sin incurrir en gastos extra) queda válido por el uso de firmas algebraicas o Rabin fingerprints.

III. Implementación

Utilizamos una *s*-sig con dos bytes y una ventana pequeña de cuatro bytes. Internamente, mantenemos un buffer circular para almacenar los últimos cuatro *s*-sig del trozo visto hasta este momento. Dado que las *s*-sig son buenas funciones hash (como vamos a probar en la sección siguiente), utilizamos una ventana pequeña, pero nuestro método es extensible naturalmente a ventanas más grandes. Los dos bytes de la *s*-sig implican que ponemos un límite en promedio después de 2^{16} bytes o en otras palabras, que el tamaño promedio de un trozo es 64kB. Ese valor es mayor a lo usual, pero podemos modificar la condición para cortar más frecuentemente.

Las propiedades de la *s*-sig (Sección IV) garantizan que cortamos para un límite después de cuatro ceros seguidos. Sin gastos adicionales, se puede examinar si tenemos una serie más amplia de ceros. No es necesario almacenar estos ceros explícitamente y la compresión con ceros solamente almacena el número de ceros encontrados.

Se provee un seudo-código de nuestra implementación en Fig. 2. Elegimos un elemento primitivo fijo α y generamos tablas de multiplicación para α, α^2, α^4, y α^8. Esos elementos son las variables a, a2, a4, y a8 en el código. Almacenamos $s(\alpha, .)$ y $s(\alpha^2, .)$ en las variables p1 y p2. Si procesamos un nuevo símbolo c del archivo, actualizamos p1 y p2 por $p1 = \alpha \cdot p1 \oplus c$ y $p2 = \alpha^2 \cdot p2 \oplus c$. Almacenamos la tupla (p1,p2) en el buffer circular.

TABLA I: Rendimiento de cálculo de MD5 y de límites de trozos.

Tarea	THROUGHPUT
MD5	1.939 GB/sec
Límites de Trozos	4.267 GB/sec

Antes de sobre-escribir este valor, recuperamos la tupla (q1, q2) antes almacenada, que es la *s*-sig del trozo anterior a los cuatro símbolos. Esos valores se utilizan para verificar las condiciones de corte $q1 \cdot \alpha^4 = p1$ y $q2 \cdot \alpha^8 = p2$. Las multiplicaciones mult con varias constantes son implementadas consultando una tabla con 256 entradas.

Un corte es activado (y un límite de trozo impuesto) cada vez que dos valores de 16 bits son iguales. Como ambos valores se comportan como valores al azar, pasa en promedio cada 2^{16} símbolos. Si queremos cortar el tamaño promedio de los trozos, calculamos el AND lógico de ambos partes con una máscara con l bits puestos para un tamaño promedio de 2^l bytes. Presentamos el algoritmo completo para la terminación de un trozo en Fig. 2.

Hemos verificado la eficiencia de nuestro algoritmo por un experimento. Los resultados en la Tabla I muestran que en nuestra maquina de prueba (con procesadora Intel Duo a 2.3 GHz), el cálculo de límites de trozos, se ejecutó dos veces más rápido que un implementación de MD5 de RSA Security, Inc. (1991). SHA-1 era 20% más lento que MD5. El cálculo de una firma algebraica completa de 16 bytes era desafortunadamente 3.8 veces más lento y nuestra esperanza original de combinar el cálculo de límites y hashs de trozos en un mismo paso se destruyó.

IV. Seguridad Adicional contra Colisiones

Una colisión es una situación donde las firmas (hashs) de dos trozos distintos son iguales. El resultado de una colisión en un sistema de deduplicación es que el archivo de entrada no es almacenado correctamente destruyendo el archivo. Por eso una colisión siempre constituye perdida de datos. Primero calculamos la seguridad adicional obtenida por añadir bytes de un hash perfectamente plano. Un hash es plano, si tiene cada valor con la misma probabilidad. Después discutimos si las firmas algebraicas son planas.

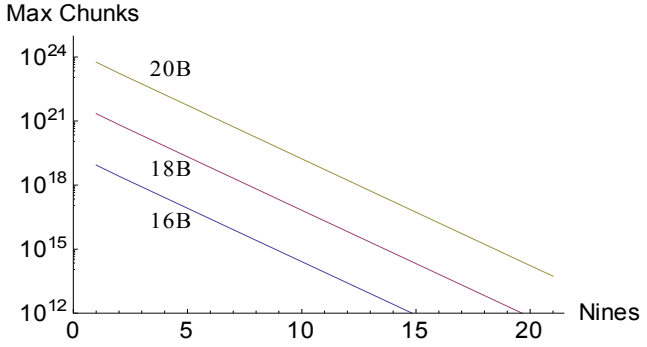

Figura 3. Número máximo de trozos para un hash perfectamente plano de 16, 18, y 20 bytes dependiente del número de nueves en la seguridad en contra de colisiones.

A. Límite de tamaño del sistema

El problema de calcular la probabilidad de una o más colisiones en un sistema con N trozos es conocido como la paradoja del cumpleaños. Existen buenas aproximaciones para la probabilidad de una colisión, pero estrictamente son validas solamente si todos los valores son igualmente probables. En ese caso, una aproximación razonable para la probabilidad de que exista una o más colisiones, si existen 2^m valores posibles de hashs es

$$p_{\text{col}}(m) = 1 - \exp(\frac{-N^2}{2^{m+1}})$$

Ese número aplica para hashs o firmas de m bits. Medimos la resistencia en contra de la existencia de una o más colisiones con la *seguridad*, definida como probabilidad de que no haya una colisión en el sistema. Damos esa seguridad en números de nueves (nines). Por ejemplo, un nivel de seguridad de seis nueves es 99,9999% y corresponde a un valor $\varepsilon = p_{\text{col}} =$ 0,000001. Para que deduplicación sea aceptable, p_{col} debe ser menos que las probabilidades de otras causas de pérdida de datos. Si n es el número de los nueves en la seguridad, queremos que la probabilidad de una (o más colisiones sea menos que $\varepsilon = 10^{-n}$). Solucionamos la desigualdad correspondiente y obtenemos

$$N \le \sqrt{-\ln(1\text{-}\varepsilon)2^{m+1}}$$

Como ε <<1, podemos utilizar la aproximación lineal de la expansión de Taylor para el logaritmo natural alrededor de 1 y obtenemos

$$N \le \sqrt{\varepsilon} \cdot 2^{(m+1)/2}$$

con un error de O(ε). Dado n, tenemos $\varepsilon = 10^{-n}$. Con el logaritmo de base 10 obtenemos

$$\log_{10}(N) \le -\frac{n}{2} + (m+1)\frac{\log_{10}(2)}{2}$$
$$\approx -\frac{n}{2} + 0.150515(m+1)$$

Como resultado, el número máximo de trozos que se puede admitir en un sistema de almacenamiento para que la probabilidad de una (o más) colisión sea menos de ε se incrementa por $10^{1.20412}$ para cada byte adicional. Dos bytes adicionales incrementan el tamaño máximo por un factor de 256 y tres por un factor de 4096. Presentamos los resultados numéricos en Fig. 3. El eje horizontal da el número de nueves de la seguridad en contra de colisiones y el eje vertical el número máximo de trozos. Si multiplicamos ese número con el tamaño promedio de los trozos, obtenemos la capacidad máxima del sistema de almacenamiento.

Por ejemplo, si la probabilidad de pérdida de datos es estimado a 10^{-9} por año (un valor muy bueno) y si la vida del sistema de almacenamiento es 100 años (un valor muy alto), entonces la seguridad en contra de colisiones debe ser mejor que 10^{-11}. Arbitrariamente elegimos un nivel de seguridad de 15 nueves. Si utilizamos un hash similar a MD5 con 16 bytes, el número máximo de trozos es $8{,}692\times10^{11}$. Si utilizamos nuestro método para añadir dos bytes al hash, ese límite cambia a $2{,}225\times10^{14}$. Si el promedio del tamaño de los trozos es 4KB, la capacidad máxima del sistema de almacenamiento cambia de cientos de petabytes a dieces de exabytes, llegando a los límites de un sistema actual de archivos.

B. Firmas algebraicas son planas

Nuestro cálculo dependía de que los hashes eran perfectamente planos. Eso es el caso exactamente para firmas algebraicas y aproximadamente para los fingerprints de Rabin, si consideramos que cualquier combinación de bytes es igualmente probable para un trozo. Un veredicto si los hashes criptográficos (como MD5, SHA-1, SHA-2) son planos es muy difícil de obtener aunque Bellone y Kohno [16] reportan que ciertos bytes de un SHA-1 son estadísticamente perfectamente planos.

El argumento de que las firmas algebraicas (o la s-sig) son planas para textos aleatorios es sencillo. Si todos los símbolos de 8 b en una cadena tienen la misma probabilidad y si no hay una dependencia entre símbolos, entonces la probabilidad de un cierto valor es el inverso del número de trozos que tienen ese valor como hash. Para que el hash de una cadena X tenga un cierto valor $\mathbf{c} = (c_1, c_2, \ldots, c_m)$ como su s-sig, se deben cumplir los m igualdades

$$s_{\alpha,m}(X) := \left(s(\alpha, X), s(\alpha^2, X), \ldots, s(\alpha^m, X)\right) = (c_1, c_2, \ldots, c_m)$$

que podemos escribir equivalentemente como

$$\mathbf{A} \cdot \mathbf{X} = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{pmatrix}$$

con la matriz

$$\mathbf{A} = \begin{pmatrix} \alpha^{n-1} & \alpha^{n-2} & \cdots & \alpha^1 & 1 \\ \alpha^{2(n-1)} & \alpha^{2(n-2)} & \cdots & \alpha^2 & 1 \\ \alpha^{3(n-1)} & \alpha^{3(n-2)} & \cdots & \alpha^3 & 1 \\ \vdots & \vdots & \ddots & & \vdots \\ \alpha^{m(n-1)} & \alpha^{m(n-2)} & \cdots & \alpha^m & 1 \end{pmatrix}$$

La matriz **A** es una matriz de tipo Vandermonde si ordenamos las columnas en orden reverso, y tiene por eso el rango máximo de m. La solución del sistema de igualdades anterior es un espacio de dimensión m y el número de soluciones es el mismo para cada selección **c** del valor de la firma. Por eso, si la entrada es aleatoria, cada valor de la firma algebraica compuesta tiene la misma probabilidad. La firma algebraica es completamente plana.

Las propiedades análogas para MD5, SHA-1 y SHA-2 son ciertas solo aproximadamente. No son ciertas para los Rabin fingerprints porque se realizan con un mod de un polinomio g. Si transformamos g en una cadena binaria y esa cadena en un número entero de precisión arbitraria, se puede decir que el Rabin fingerprint no puede ser mayor a g. Eso limita los valores que puede tener un Rabin fingerprint. El número de

valores que no puede tener es bastante pequeño y por eso se puede decir que los Rabin fingerprints son aproximadamente planos.

TABLA II: Número actual y esperado de colisiones con firmas de dos bytes en un diccionario (2.259MB) de palabras de poco tamaño.

Número de Palabras en Colisión	Observado	Esperado
0	2741	2664.6
1	8539	8533.6
2	13599	13664.5
3	14563	14587.0
4	11645	11678.9
5	7377	7480.4
6	4140	3992.7
7	1835	1826.7
8	721	731.25
9	269	260.2
10	69	83.3
11	30	24.3
12	8	6.5
≥ 13	0	2.1

TABLA III: Número actual y esperado de colisiones con firmas de tres bytes utilizando el mismo diccionario.

Número de Palabras en Colisión	Observado	Esperado
0	16 568 642	16 568 642.3
1	207 274	207272
2	1 293	1 296.47
3	7	5.40624
≥ 4	0	0.0169079

TABLA IV: Número actual y esperado de colisiones con firmas de dos bytes utilizando un diccionario de 733 KB..

Número de Palabras en Colisión	Observado	Esperado
0	19092	19011
1	23370	23527.7
2	14654	14558.7
3	5943	6005.84
4	1194	1858.18
5	455	499.928
6	89	94.8663
7	18	16.7721
8	1	2.5946
9	0	0.35678
10	0	0.0441544
≥ 11	0	0

Nuestros cálculos y argumentos asumen entradas aleatorias y sabemos que esa no es la carga normal de un sistema de almacenamiento. Utilizábamos texto en Ingles para probar si las firmas son planas. Para hacer la prueba más difícil, utilizamos una lista de palabras de poco tamaño. La primera lista tenía 209881 palabras y un tamaño total de 2.259 MB y estaba siendo utilizado en ataques de diccionarios en contra de contraseñas (para administradores verificando la seguridad de su sistema). En el primer experimento, calculamos la firma algebraica de dos bytes de todas las palabras con más de cinco letras (65536 palabras). La Tabla II da el número de palabras que tienen la misma firma. Por ejemplo, encontramos a 12 firmas que eran el valor para 8 palabras diferentes (penúltima línea). Similarmente, había 2741 combinaciones de dos bytes, que no eran firma de una o más palabras (primera línea de la Tabla II). Lo comparamos con el valor esperado para una firma completamente plana. Ese valor es dado por una distribución Poisson con un promedio de muestra de 3.202530. Esos valores forman la columna derecha en Tabla II. Una comparación visual muestra que las firmas algebraicas se comportaban mejor que lo esperado. Esa impresión es confirmada por el valor de χ^2, que calculamos a ser 0.21 utilizando 8+1 clases.

Una repetición del experimento utilizando el mismo conjunto de palabras, pero con una firma de tres bytes resultó en los valores presentados en la Tabla III. La coincidencia entre los valores observados y esperados es correcta, como es confirmado por el valor χ^2 de 0.479166.

Otra repetición del primer experimento con otra lista más pequeña de palabras (Tabla IV) no daba el mismo comportamiento estelar pero el valor de χ^2 de 4.79742 indica que estadísticamente no se puede distinguir entre la firma algebraica y un hash completamente plano. La cola de la distribución actual es más pequeña que la de la distribución esperada, indicando un desempeño mejor que el esperado de las firmas algebraicas.

Queremos insistir en la observación de que la existencia de colisiones es resultado del tamaño pequeño de las firmas algebraicas. Con 2 bytes, una firma solamente puede tener uno de 256^2 posibles valores y con más entradas, deben existir colisiones.

V. Conclusiones

Hemos presentado un método explotando trabajo ya hecho para la determinación de límites de trozos, para añadir dos (o tres) bytes al hash de un trozo. Los bytes adicionales son obtenidos sin incurrir en gastos adicionales pero mejoran la resistencia entre colisiones de hashs de trozos. Como resultado un sistema de almacenamiento con deduplicación puede crecer por un factor de 256 (o 4096 con tres bytes adicionales) dando la misma garantía contra colisiones que antes del cambio. Alternativamente, con los mismos bytes añadidos se puede utilizar un hash de trozos con menos bytes, pero más rápido de calcular.

Nuestro método es uno de pocos casos de "algo por nada" en la informática. La integración de los bytes adicionales es trivial.

VI. Trabajo Futuro

El trabajo presentado en este artículo surgió de un intento de combinar el cálculo de límites de trozos y de hashs de trozos

en un solo paso, utilizando firmas algebraicas. Desafortunadamente, nuestros experimentos mostraban que el cálculo de una firma algebraica de 16 o 20 bytes es bastante más lenta que el cálculo de MD5 y SHA-1. No es sorpresa, porque ambas funciones de hash fueron diseñadas explícitamente para procesar varios bytes en un solo paso sin el aislamiento costoso de un solo byte en una arquitectura actual de procesador. Las nuevas arquitecturas de chips (ejemplo los Westmere chips introducido por Intel en 2010) implementan nuevas instrucciones de maquina útiles para el cálculo de AES y posiblemente para el cálculo de firmas algebraicas. Para esas arquitecturas, el desempeño necesita ser investigado con más detalle.

REFERENCIAS

[1] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST), 2008, pp. 269–282.

[2] N. Mandagere, P. Zhou, M. Smith, and S. Uttamchandani, "Demystifying data deduplication," in Proceedings of the ACM/IFIP/USENIX Middleware'08 Conference Companion. ACM, 2008, pp. 12–17.

[3] D. Bhagwat, K. Eshghi, D. Long, and M. Lillibridge, "Extreme Binning: Scalable, Parallel Deduplication for Chunk-based File Backup," in Proceedings of the 17th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2009), 2009.

[4] G. Forman, K. Eshghi, and S. Chiocchetti, "Finding similar files in large document repositories," in Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining. ACM, 2005, p. 400.

[5] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in Proceedings of the eighteenth ACM symposium on Operating systems principles. ACM, 2001, pp. 174–187.

[6] S. Quinlan and S. Dorward, "Venti: a new approach to archival storage," in Proceedings of the FAST 2002 Conference on File and Storage Technologies, vol. 4, 2002.

[7] K. Eshghi and H. Tang, "A framework for analyzing and improving content-based chunking algorithms," Hewlett-Packard Labs Technical Report TR, vol. 30, 2005.

[8] M. W. Storer, K. Greenan, D. D. Long, and E. L. Miller, "Secure data deduplication," in Proceedings of the 4th ACM international workshop on Storage security and survivability, StorageSS '08. 2008, pp. 1–10.

[9] W. Litwin and T. Schwarz, "Algebraic signatures for scalable distributed data structures," in Proceedings. 20th International Conference on Data Engineering, 2004, pp. 412–423.

[10] M. Rabin, "Fingerprinting by random polynomials," Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[11] A. Broder, "Some applications of Rabins fingerprinting method," in Sequences II: Methods in Communications, Security, and Computer Science, Springer Verlag, 1993, pp. 143–152.

[12] F. I. P. S. NIST, "FIPS-180-1: Secure Hash Standard," 1995.

[13] R. Rivest, "RFC1321: The MD5 message-digest algorithm," RFC Editor United States, 1992.

[14] K. Greenan, E. Miller, and T. Schwarz, "Optimizing Galois Field arithmetic for diverse processor architectures and applications," in Proceedings of the 16th IEEE Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2008.

[15] Z. Schnabel, "The estimation of total fish population of a lake," American Mathematical Monthly, vol. 45, no. 6, pp. 348–352, 1938.

[16] M. Bellare and T. Kohno, "Hash function balance and its impact on birthday attacks," in Advances in Cryptology-Eurocrypt 2004. Springer, 2004, pp. 401–418.

Witold Litwin is Professeur Exceptionnel (the highest rank in France) at Université de Paris 9 (Dauphine) and Director of Centre d'Etudes et de Recherches en Informatique Appliquée (CERIA). He is a Fellow of the ACM (the only one in France), and a world authority on scalable data structures. He is the inventor of Linear Hashing. He is the author of more than 150 publications and has contributed to or edited 11 books.

Darrell D. E. Long, Fellow IEEE, Fellow AAAS is the Director of the Storage Systems Research Center. He is Professor of Computer Science and holds the Kumar Malavalli Endowed Chair. His current research interests in the storage systems area include high performance storage systems, archival storage systems and view-based file systems. His research also includes computer system reliability, video-on-demand, applied machine learning, mobile computing and computer security.

Thomas J. E. Schwarz, S.J., Senior Member, IEEE, is Adjunct Associate Professor of Computer Science at the University of California, Santa Cruz and Professor and Director of the Departamento de Informática y Ciencias de la Computación at the Universidad Católica del Uruguay. Previously he was on the faculty at Santa Clara University. His interests include distributed data structures, storage systems, and error detection and correction codes.