Funciones hash criptográficas

José Galaviz Casas

Facultad de Ciencias, UNAM

Índice general

1. Introducción

Desde muy temprano en los cursos que los profesionales de la computación reciben a nivel licenciatura, típicamente en los cursos de estructuras de datos, se les introduce el concepto de *función hash*, también llamada de dispersión. Este tipo de funciones se utilizan primeramente para determinar el lugar en el que se ha de almacenar un objeto arbitrario en un arreglo. Esto resulta de suma utilidad para lograr un acceso eficiente a los objetos almacenados de esta forma, ya que el tiempo de acceso a un arreglo es constante sin importar el número de elementos que el arreglo contenga. En el caso ideal que por cierto, como veremos, resulta imposible, desearíamos entonces que el número entregado por la función identifique unívocamente al objeto que recibió como entrada, para que no haya más de un objeto por cada lugar en el arreglo. Esta cualidad es la que hace interesantes a las funciones *hash* en el contexto criptográfico, ya que permitiría, en principio, identificar cada una de sus posibles entradas con lo que se podría verificar la autenticidad o la integridad de las mismas de manera eficiente.

Una situación cotidiana en la que la se puede apreciar la utilidad de las funciones *hash* es la siguiente. Imaginemos que creamos una cuenta de acceso en un sistema. Nos registramos, creamos nuestro nombre de usuario (*login*) y luego damos nuestra contraseña (*password*). De alguna manera segura, de la que no nos ocuparemos ahora, el texto de la contraseña llega hasta el programa que nos atiende y este debe almacenar algo que le permita identificar ese mismo texto la siguiente vez que le sea dado, para poder certificar que quien lo proporciona es un usuario auténtico del sistema. Claro que resultaría inseguro guardar el texto mismo de la contraseña, ya que entonces alguien con acceso privilegiado al sistema puede hacerse pasar por todos los usuarios, ya que puede copiar cualquier contraseña tal y como la tecleo cada usuario. Así que el sistema, en vez de almacenar la contraseña de cada usuario, guarda el resultado de evaluar una función *hash* sobre cada una de ellas. Así, cuando un usuario entra en sesión y teclea su contraseña, basta con evaluar la función *hash* sobre lo que tecleo esta vez y compararlo

con lo que el sistema tiene ya almacenado, si coinciden, si la función *hash* identificara unívocamente lo que recibe, podríamos estar seguros de que el usuario es quien dice ser. A lo largo de este documento veremos que se necesita para que esto ocurra, qué se puede lograr y hasta qué grado.

Aún en los cursos elementales, cuando se aborda el tema con un poco más de profundidad, se hace evidente que existe un problema insalvable, aunque se puede mitigar de hecho, bastante eficientemente. El problema en cuestión es que, al recibir una entrada arbitraria, el dominio de la función *hash* es enorme en la práctica; en términos teóricos, infinito, mientras que la salida de la función es un número entero en un rango bastante limitado en general. La consecuencia inmediata de esto es que habrá más de un objeto que sea mapeado al mismo número, a lo que se le denomina *colisión*. Las colisiones son inevitables, es seguro que ocurrirán, lo importante aquí es que, si pensamos en el contexto del manejo de contraseñas del párrafo previo, nadie pueda encontrar eficientemente ni la contraseña, ni otra cadena de texto que al evaluarse bajo la función *hash* entregue el mismo valor. Este es un concepto puramente computacional, como suele suceder en criptografía: sabemos que la función en general no puede ser inyectiva, que debe haber colisiones, pero lo importante es que sea *difícil* encontrarlas.

2. Características de una función hash

Dado que estaremos tratando con funciones que mapean datos en datos y estos serán representados en una computadora, podemos suponer que se tratan de cadenas de ceros y unos. Las funciones *hash* mapean cadenas de longitud arbitraria en cadenas de longitud fija n. Denotaremos estos conjuntos como $\mathbb{B}_\infty$ y $\mathbb{B}_n$ respectivamente.

Esencialmente cualquier función $H : \mathbb{B}_\infty \to \mathbb{B}_n$ puede ser considerada una *función hash* si posee las siguientes características:

- Determinismo. Las funciones *hash* son funciones en el sentido matemático del término, dado un elemento del dominio a, la función debe entregar siempre el mismo valor al evaluarse sobre a.
- Uniformidad. La función debe *dispersar* de la mejor manera posible su dominio en el contradominio. Más formalmente: todos los distintos valores en el contradominio de la función deben ser obtenidos con, aproximadamente, la misma probabilidad, evaluada sobre el dominio.
- Rango bien definido. Como el tamaño de las cadenas binarias entregadas como salida de la función es fijo (n), si estas son consideradas como números binarios, poseen un rango bien definido ($\{0, \ldots 2^n - 1\}$) sin importar cuál sea el valor de entrada que se da.

En ocasiones se le pide a la función tener algunas otras características, ser "continua", por ejemplo: mapear objetos de entrada similares en números similares de salida o ser aproximadamente "invertible", es decir, dado un valor en su contradominio, poder calcular eficientemente algunas características sobresalientes de los posibles objetos que se mapean en él. Estas características son del todo indeseables en las funciones *hash* usadas en criptografía.

Una función *hash* criptográfica H debe poseer en buena medida las siguientes características adicionales:

- Eficientemente calculable. Debe poder evaluarse sobre cualquier elemento de su dominio de manera eficiente. En términos computacionales, el orden de complejidad asintótico del tiempo que toma calcular la función debe ser lo más bajo posible. En las funciones usuales actualmente, depende linealmente del tamaño de la entrada.

- Altamente dependiente de la entrada. Debe depender de todos y cada uno de los elementos de los datos de entrada. Un pequeño cambio en estos, debe generar cambios notables en la salida. En términos de la teoría de la información de Shannon: debe poseer mucha difusión.

- Resistencia de pre-imagen. Dado un valor h en el contradominio de la función *hash* H, debe ser difícil encontrar un cierto elemento a en el dominio de H, tal que $H(a) = h$.

- Resistencia de segunda pre-imagen. Dado un elemento a_1 en el dominio de la función, debe ser difícil encontrar otro a_2 tal que $H(a_1) = H(a_2)$.

- Resistencia a colisión. Debe ser difícil encontrar dos elementos a_1 y a_2 en el dominio de H, tales que: $H(a_1) = H(a_2)$.

Cabe señalar que el ser eficientemente calculable y la alta sensibilidad a la entrada son cualidades deseables para las funciones *hash* aún fuera del contexto criptográfico cuando son usadas para hacer búsquedas eficientes en estructuras de datos.

La resistencia a colisión es una condición más fuerte que la de segunda-preimagen, es decir, si se tiene resistencia a colisión se tiene resistencia de segunda pre-imagen. Sin embargo la resistencia a colisión no implica la de pre-imagen. En términos prácticos, una función *hash* criptográfica en uso, deja de ser utilizada cuando se le han encontrado colisiones, porque eso significa que alguien ya ha podido hallar al menos una pareja de datos de entrada que generan el mismo valor de *hash* y posiblemente a partir de ello se pueden generar muchos más o bien se ha encontrado un medio para, al menos indirectamente, encontrar una segunda pre-imagen y con ello falsificar datos cuya integridad ha sido verificada con la función en cuestión.

De hecho la mera resistencia a colisiones puede resultar hasta insuficiente en cierto contexto: se necesita que sea prácticamente imposible generar entradas para la función, que produzcan resultados deseados y hay que tener en cuenta que la función *hash* está a nuestra entera disposición. Cualquiera puede y debe tener acceso al algoritmo que calcula la función *hash* y puede usarlo las veces que quiera con las entradas que quiera, así que realmente se

necesita que la salida de la función parezca completamente aleatoria. No debe dar pista alguna que permita deducir características de la salida a partir de las de la entrada. La función *hash* debe ser impredecible.

3. Probabilidad de colisión

¿Qué tan probable es encontrar una colisión en una función *hash*? ¿Cuantos elementos de su espacio de entrada se deberían probar para encontrar al menos una colisión? Para responder esto recurriremos a un símil bastante conocido, se le menciona con frecuencia en cursos básicos de probabilidad y suele llamarse *la paradoja del cumpleaños*. Partiendo del supuesto de que la distribución de las fechas de nacimiento de las personas es uniforme, consiste en analizar cuál es la probabilidad de que en un conjunto de personas, dos coincidan en fecha de cumpleaños. Cuando esto pasa, diremos que ha ocurrido una colisión. Nos interesa calcular la probabilidad de que en un conjunto de k personas, haya al menos una colisión en fecha de cumpleaños entre dos cualesquiera de ellas. Para calcular eso, calcularemos realmente la probabilidad del evento complementario, es decir, la probabilidad de que en un conjunto de k personas no exista ningún par con la misma fecha de cumpleaños[1].

1. Por supuesto si hay una sola persona no hay con quien colisionar en fecha de cumpleaños, así que la probabilidad de no colisión es 1. No importa cuál de los 365 días[2] sea su fecha de cumpleaños:

$$P_{\neg 1} = P(\text{no colisión entre 1 persona}) = \frac{365}{365} = 1$$

2. Si hay dos personas, para que no haya colisión debería ocurrir que, dada la fecha de una de ellas, la de la otra persona fuera cualquiera de las otras 364 posibles fechas:

$$P_{\neg 2} = P(\text{no colisión entre 2 personas}) = \frac{365}{365}\,\frac{364}{365} \approx 0.997$$

[1] La suposición de que la distribución de fechas de cumpleaños es uniforme es, de hecho, la que minimiza las colisiones. Suponer otra distribución las hace aún más probables [2].

[2] Supondremos, por simplicidad, que todos los años tienen 365 días.

3. Con tres personas:

$$P_{\neg 3} = P(\text{no colisión entre 3 personas}) = \frac{365}{365}\frac{364}{365}\frac{363}{365} \approx 0.991$$

4. Por analogía, cuando tenemos k personas:

$$\begin{aligned} P_{\neg k} &= \frac{365}{365}\frac{364}{365}\frac{363}{365}\cdots\frac{365-k+1}{365} \\ &= \left(1-\frac{1}{365}\right)\left(1-\frac{2}{365}\right)\left(1-\frac{3}{365}\right)\cdots\left(1-\frac{k-1}{365}\right) \end{aligned} \tag{3.1}$$

Ahora bien, la probabilidad de tener *al menos una* colisión cuando tenemos un conjunto de k personas, es entonces:

$$P_k = 1 - P_{\neg k} \tag{3.2}$$

En la tabla 3.1 se muestran los valores que se obtienen de la expresión 3.1 y los de el evento complementario, determinado por 3.2. Como se puede ver, para cuando reunimos a 23 personas, la probabilidad de que haya una colisión rebasa el 50 %. A pesar de tener 365 días disponibles, es muy probable que las colisiones ocurran aún en grupos de personas no tan numerosos.

Podemos hacer un análisis similar para el caso de una función *hash* H. ¿Cuántos mensajes $\{X_1, X_2, \ldots, X_k\}$ se necesitan para que haya una probabilidad sustancial de haya al menos dos mensajes X_p y X_q tales que $H(X_p) = H(X_q)$? Por analogía con nuestro razonamiento previo, si suponemos que el tamaño de bloque de salida de la función H es n y por tanto hay 2^n posibles resultados de aplicar H a una entrada cualquiera:

$$\begin{aligned} P_{\neg k} &= \left(1-\frac{1}{2^n}\right)\left(1-\frac{2}{2^n}\right)\cdots\left(1-\frac{k-1}{2^n}\right) \\ &= \prod_{i=1}^{k-1}\left(1-\frac{i}{2^n}\right) \end{aligned} \tag{3.3}$$

Usando la expresión A.2:

$$\begin{aligned} P_{\neg k} &\approx \prod_{i=1}^{k-1}\left(e^{-\frac{i}{2^n}}\right) \\ &= e^{-\frac{1}{2^n}}\cdot e^{-\frac{2}{2^n}}\cdots e^{-\frac{k-1}{2^n}} \\ &= e^{-\frac{1+2+3+\cdots+(k-1)}{2^n}} \end{aligned}$$

k	$P_{\neg k}$	P_k	k	$P_{\neg k}$	P_k
1	1.000	0.000	13	0.806	0.194
2	0.997	0.003	14	0.777	0.223
3	0.992	0.008	15	0.747	0.253
4	0.984	0.016	16	0.716	0.284
5	0.973	0.027	17	0.685	0.315
6	0.960	0.040	18	0.653	0.347
7	0.944	0.056	19	0.621	0.379
8	0.926	0.074	20	0.589	0.411
9	0.905	0.095	21	0.556	0.444
10	0.883	0.117	22	0.524	0.476
11	0.859	0.141	23	0.493	0.507
12	0.833	0.167	24	0.462	0.538

Tabla 3.1: Probabilidad de que no haya colisión y de que haya al menos una en un conjunto de k personas.

El numerador del exponente es la suma de los primeros $k-1$ números naturales:

$$1+2+3+\cdots+(k-1)=\frac{k(k-1)}{2}$$

En la expresión previa entonces:

$$P_{\neg k} \approx e^{-\frac{k(k-1)}{2^{n+1}}}$$

Así que la probabilidad de encontrar al menos una colisión en k mensajes:

$$\begin{aligned} P_k &= 1-P_{\neg k} \\ &\approx 1-e^{-\frac{k(k-1)}{2^{n+1}}} \end{aligned}$$

De donde:

$$\ln\left(1-P_k\right) \approx -\frac{k(k-1)}{2^{n+1}}$$

Entonces:

$$\ln\left(\frac{1}{1-P_k}\right) \approx \frac{k(k-1)}{2^{n+1}}$$

De donde:

$$k(k-1) \approx 2^{n+1}\ln\left(\frac{1}{1-P_k}\right)$$

Como en general k debe ser un número grande, $k(k-1)\approx k^2$, entonces:

$$k^2 \approx 2^{n+1}\ln\left(\frac{1}{1-P_k}\right)$$

O bien:

$$k \approx \sqrt{2^{n+1}\ln\left(\frac{1}{1-P_k}\right)}$$

Es decir:

$$k \approx 2^{\frac{n+1}{2}}\sqrt{\ln\left(\frac{1}{1-P_k}\right)} \tag{3.4}$$

es el número k de ensayos que deberíamos de hacer con la función *hash* para tener una probabilidad P_k de encontrar una colisión.

P_k	128	160	256	512
0.5	2^{65}	2^{81}	2^{129}	2^{257}
0.9	2^{67}	2^{82}	2^{130}	2^{258}

Tabla 3.2: Probabilidad de encontrar una colisión en funciones *hash* dependiendo del tamaño de la salida en bits de la función.

Ejemplo 3.1. Supongamos que tenemos una función *hash* de 80 bits de salida y que deseamos determinar el número k de pruebas que debemos hacer para tener una probabilidad del 0.5 de encontrar una colisión.

Substituyendo en la expresión 3.4 tenemos que:

$$k \approx 2^{\frac{81}{2}} \sqrt{\ln\left(\frac{1}{1-0.5}\right)} \approx 2^{40.2}$$

Es decir, habría que calcular alrededor de 2^{40} resultados de *hash* para tener un 50 % de posibilidades de encontrar una colisión.

$\triangleleft$

De la expresión 3.4 es claro que el número de resultados de la función *hash* que hay que calcular para tener una alta probabilidad de encontrar una colisión es del orden de la raíz cuadrada (dividir entre dos el exponente) del espacio de salida de la función. En la tabla 3.2 se muestra el número de ensayos necesario en función de la probabilidad de encontrar colisión y del tamaño de la salida de la función.

4. Funciones unidireccionales de compresión

Si lo que se requiere es tomar un conjunto de datos de tamaño arbitrario y obtener a partir de ello un bloque de datos de tamaño fijo, altamente dependiente de la entrada, y que sea muy difícil determinar lo que entró a partir de lo que sale, podríamos pensar en usar para ello un algoritmo de cifrado de bloque como los que solemos utilizar en criptografía simétrica. Hay que recordar que estos algoritmos reciben dos bloques de entrada, uno con los datos a cifrar y otro con la clave que se usará para hacerlo.

Digamos que tenemos un largo mensaje m de tamaño M al que queremos calcular su valor de *hash* que debe medir exactamente $N < M$ bits. Dividimos el mensaje en bloques de tamaño N: $\{m_1, m_2, \ldots, m_{\lceil M/N \rceil}\}$. Posiblemente el último bloque quede incompleto y entonces debemos pensar en rellenarlo para que mida también N bits, pero eso lo revisamos después. Cada uno de esos bloques lo introducimos en nuestro algoritmo de cifrado E que recibe, además de una cierta clave, un bloque de datos a cifrar de N bits y entrega N bits de datos cifrados a la salida. Al final hacemos un XOR de las salidas y obtenemos un solo bloque de tamaño N. Habría que aclarar que, dado que se requiere que el cálculo de *hash* sobre un conjunto de datos sea siempre el mismo, necesitamos garantizar que siempre se introduzca la misma clave, así que esta la deberemos fijar; en este caso le llamaremos *vector de inicialización*, lo que en la literatura del área suele denotarse como VI. De hecho podríamos pensar en dividir el mensaje en bloques del tamaño de la clave, no del bloque de entrada (que por cierto en muchas ocasiones son del mismo tamaño) e ir introduciendo los datos como "claves". En esta caso el vector de inicialización tendría que entrar, no en lugar de una clave, sino en el del texto a cifrar.

Esta construcción no suena mal, salvo el hecho de que no estamos logrando toda la difusión posible, cada uno de los bits del primer bloque (digamos el del extremo izquierdo) influye sobre todos, ciertamente, los bits del *primer*

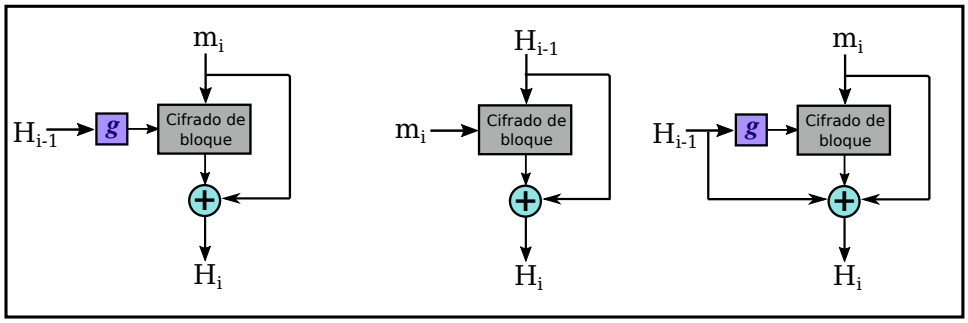

Figura 4.1: Construcción de funciones *hash* con base en un cifrado de bloque. A la izquierda la construcción de Matyas-Meyer-Oseas, en el centro la de Davies-Meyer y a la derecha la de Miyaguchi-Preneel.

bloque de salida y solamente esos. Un mejor esquema es mezclar las salidas del cifrado de cada bloque con la del previo. En la figura 4.1 se muestran tres diferentes construcciones que hacen uso de un cifrado de bloque. En los diagramas mostrados, la función g bien pudiera ser la identidad, si es que el tamaño de la clave que recibe el cifrado coincide con lo que se requiere, si no es así la función g deberá hacer algo para ajustar su entrada a lo requerido por el cifrado. En la construcción de Matyas-Meyer-Oseas el mensaje cuyo *hash* se requiere m es el texto a cifrar, H_0 sería el vector de inicialización. En la de Davies-Meyer el mensaje se divide en bloques del tamaño de la clave que usa el cifrado. La de Miyaguchi-Preneel pretende mayor dependencia al añadir a la de Matyas-Meyer-Oseas el *hash* del bloque previo en el XOR. Las ecuaciones que implementan estos esquemas de *hash* son las siguientes:

- Matyas-Meyer-Oseas: $H_i = m_i \oplus E_{g(H_{i-1})}(m_i)$.
- Davies-Meyer: $H_i = H_{i-1} \oplus E_{m_i}(H_{i-1})$.
- Miyaguchi-Preneel: $H_i = m_i \oplus H_{i-1} \oplus E_{g(H_{i-1})}(m_i)$.

En cada uno de los diseños mostrados, el cifrado de bloque usado recibe dos entradas y entrega una sola salida del tamaño de una de sus entradas. Es por ello que a la función implementada por el cifrado de bloque se le denomina *función unidireccional de compresión.*

5. Merkle-Damgård

Con la herramienta que ya poseemos, podemos pensar entonces en construir una función *hash* como un proceso iterativo en el que se van suministrando uno tras otro, los bloques del mensaje a la función de compresión y en cada iteración el resultado del *hash* de la iteración previa se introduce como una de las dos entradas de la función de compresión, la otra entrada es, claro, el siguiente bloque del mensaje. Al inicio, la entrada que estaría asociada al valor *hash* del bloque previo, claro, no existe; así que suministramos un valor constante arbitrario al que llamamos vector de inicialización o VI. Como mencionamos, el último bloque del mensaje puede no estar completo, es decir, puede tener un tamaño menor al del bloque aceptado por la función de compresión, así que se le añade un relleno R. En la figura 5.1 se muestra esta construcción.

Habíamos dejado de lado momentáneamente el hecho de que el mensaje del que se desea el *hash* tuviera una longitud que no fuera múltiplo del tamaño de bloque requerido por el cifrado. Dijimos ya que en estos casos se requiere hacer un relleno (*padding* en inglés) al final del mensaje para completar el bloque. En primera instancia uno podría pensar en rellenar con

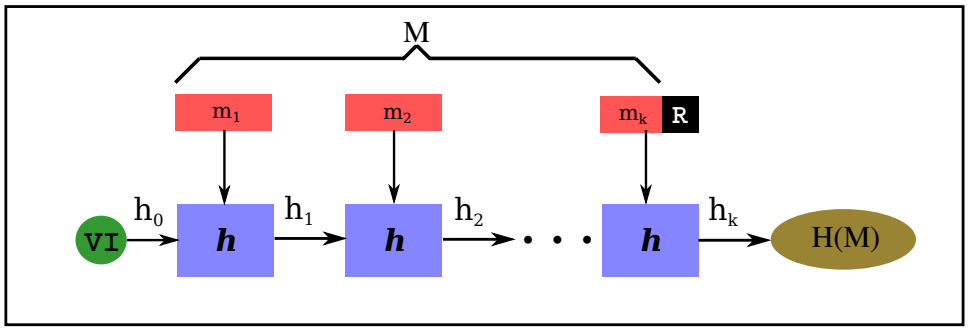

Figura 5.1: Construcción de Merkle-Damgård para una función *hash*.

una cadena de ceros, algo inocuo que no agrega dato alguno. Pero no puede ser así. Supóngase que se tiene un mensaje M_1 arbitrario, de un tamaño que no sea múltiplo del tamaño de bloque B, imaginemos de hecho que le faltan 6 bytes para tener un tamaño $T = k \cdot B$. Supóngase que se tiene además a M_2 un mensaje idéntico a M_1 salvo que es un poco más grande, tiene tres bytes adicionales a M_1, los tres iguales a cero. Para completar el mismo tamaño T habrá que añadir tres bytes menos. Si rellenamos con puros ceros, sin embargo, ambos mensajes M_1 y M_2 tendrán el mismo valor de *hash*, siendo que eran mensajes estrictamente diferentes.

El relleno más sencillo consiste entonces en añadir, al final del mensaje un bit con valor "1" y luego una cadena de ceros de la longitud necesaria para completar una longitud que sea múltiplo de B. En el peor de los casos el mensaje tenía una longitud igual a $B-1$, así que se añade un 1 al final, se completa el tamaño de bloque y, para distinguir este caso de aquel en el que el mensaje tenía un múltiplo de B de longitud y terminaba en 1, se deberá añadir un bloque completo de ceros. En este esquema el relleno es fijo y no depende para nada del mensaje. Por el momento supondremos que este es el esquema de relleno usado.

La construcción de Merkle-Damgård tiene la virtud de transferir la seguridad completa del *hash* H a la función de compresión h.

Teorema 1. *Sea $H : \mathbb{B}_\infty \to \mathbb{B}_n$ una función hash construida mediante el procedimiento iterativo de Merkle-Damgård y sea $h : \mathbb{B}_n \to \mathbb{B}_n$ su función de compresión. Si h es resistente a colisión entonces lo es también H*

Demostración. Lo demostraremos por contraposición, es decir, mostrando que si H no es resistente a colisión, entonces h no lo es tampoco.

Sean M y M' dos mensajes de longitud arbitraria y sean $H(M)$ y $H(M')$ sus valores de *hash*. Supongamos que $M \neq M'$ y sin embargo $H(M) = H(M')$, lo que constituye una colisión.

M consta de $k-1$ bloques de longitud n y un posible fragmento de longitud menor: $\{m_1, m_2, \ldots, m_k\}$ que se rellena con una cadena R, por su parte M' consta de $t-1$ bloques de longitud n y un posible fragmento de longitud menor: $\{m'_1, m'_2, \ldots, m'_t\}$ que se rellena con una cadena R'.

Es conveniente notar que la salida de la función *hash* es la de su última etapa, es decir:

$$H(M) = h_k^M = h(m_k \| R, h_{k-1}^M)$$

$$H(M') = h_t^{M'} = h(m'_t \| R', h_{t-1}^{M'})$$

Denotaremos con $\|$ la yuxtaposición de dos cadenas binarias. Así que estamos suponiendo que:

- $M \neq M'$
- $H(M) = h_k^M = h(m_k \| R, h_{k-1}^M) = h(m'_t \| R', h_{t-1}^{M'}) = h_t^{M'} = H(M')$

Hay que considerar dos casos:

Si las longitudes de M y M' son diferentes entonces $R \neq R'$ y en las entradas de la última etapa $m_k \| R \neq m'_t \| R'$ y sin embargo $h(m_k \| R, h_{k-1}^M) = h(m'_t \| R', h_{t-1}^{M'})$ lo que significa que ha ocurrido una colisión en h.

Si las longitudes de M y M' son iguales entonces $k = t$, el número de iteraciones para obtener $H(M)$ y $H(M')$ es el mismo y los rellenos usados son iguales, la diferencia entre los argumentos de h no necesariamente ocurre en la última etapa, pero dado que M y M' no son iguales, debe haber un primer bit de derecha a izquierda en el que son diferentes. Digamos que ese bit está contenido en el bloque j de ambos mensajes. Entonces, en la j-ésima iteración:

$$h_j^M = h(m_j, h_{j-1}^M) = h(m'_j, h_{j-1}^{M'}) = h_j^{M'}$$

con $m_j \neq m'_j$, así que la función de compresión h obtuvo el mismo valor a partir de argumentos diferentes, lo que es una colisión.

Hemos probado entonces que para que haya colisión en H, debe haber colisión en h, lo que se traduce en que, encontrar una colisión en H es al menos tan difícil como encontrar una en su función de compresión.

□

Muchas funciones *hash* criptográficas conocidas son del tipo Merkle-Damgård: MD5, SHA1 y la familia SHA2, por ejemplo. MD5 fue publicada en 1992 y su seguridad fue rota en 2013. SHA1 fue publicada en 1995 y en 2017 se encontró una colisión que además permite generar muchas colisiones más. SHA2 es una familia de seis funciones *hash*, basadas en un cifrado de bloque estructurado al estilo Davies-Meyer, con tamaños de bloque de salida de 224, 256,384 o 512 bits. A la fecha de escritura de este documento se han montado ataques exitosos de resistencia de pre-imagen para SHA-256 y SHA-512 pero sólo con un número de iteraciones menor al establecido por el estándar, así que formalmente se mantienen vigentes.

La construcción de Merkle-Damgård ha hecho posible construir algunas de las funciones *hash* más robustas, sin embargo poseen una debilidad inherente que radica en el hecho de que el resultado final del *hash* no es más que el vertido del estado que guarda en ese momento la función de compresión. Si fuera posible establecer arbitrariamente ese estado, entonces se podría

simular que se ha recibido la entrada que lo genera y a partir de allí alcanzar un nuevo estado con una entrada complementaria arbitraria. Es decir, supongamos que poseemos el valor de *hash* de un mensaje M_1, este es el estado que alcanzó la función de compresión luego de recibir M_1; si podemos restablecer este estado, aún sin conocer M_1, entonces la función *hash* puede continuar a partir de allí procesando un mensaje complementario M_2 de tal forma que terminaremos con $H(M_1 \| M_2)$ aún cuando nunca hayamos tenido M_1. A esto se le conoce como *ataque de extensión de longitud* (*length extension attack*).

6. SHA-256

Muchas de las funciones *hash* que son y han sido populares siguen el diseño de Merkle-Damgård. En particular los estándares federales para procesamiento de información (FIPS) norteamericanos desde 1993 y hasta antes de 2015 están basados en este paradigma. Al momento de escribir este documento permanecen vigentes varios de estas funciones estándar, todas ellas muy similares, nos abocaremos a describir aquí una de ellas: SHA-256, la que forma parte de una familia de funciones *hash* añadidas al estándar FIPS-180 desde 2002 con el fin de subsanar la posibilidad (hoy un hecho), de que resultaran exitosos los ataques a la función SHA-1, cuyo bloque de salida era de 160 bits. En 2002 se integraron al estándar funciones con diversos tamaños de salida además del de 160 de SHA-1: 224, 256, 384 y 512.

Las funciones que constituyen el corazón criptográfico de SHA-256 son las siguientes, en ellas el operador $a \text{ ROR } b$ significa rotar a la derecha los bits de a tantos lugares como indique b:

$$Ch(x,y,z) = (x \cdot y) \oplus (\bar{x} \cdot z) \tag{6.1}$$

$$Maj(x,y,z) = (x \cdot y) \oplus (x \cdot z) \oplus (y \cdot z) \tag{6.2}$$

$$\Sigma_0(x) = (x \text{ ROR } 2) \oplus (x \text{ ROR } 13) \oplus (x \text{ ROR } 22) \tag{6.3}$$

$$\Sigma_1(x) = (x \text{ ROR } 6) \oplus (x \text{ ROR } 11) \oplus (x \text{ ROR } 25) \tag{6.4}$$

$$\sigma_0(x) = (x \text{ ROR } 7) \oplus (x \text{ ROR } 18) \oplus (x \text{ SHR } 3) \tag{6.5}$$

$$\sigma_1(x) = (x \text{ ROR } 17) \oplus (x \text{ ROR } 19) \oplus (x \text{ SHR } 10) \tag{6.6}$$

1. El mensaje original M_{orig} se divide en bloques de 512 bits de longitud.
2. Se procede a rellenar. El relleno de SHA-256 añade al mensaje lo siguiente, suponiendo que el mensaje tiene una longitud ℓ en bits:
   a) Un bit con valor "1".
   b) Se le añaden luego k bits con valor "0", donde k es el menor entero no negativo tal que: $\ell + 1 + k \equiv 448 \pmod{512}$.

c) El valor de ℓ escrito en 64 bits.

Nótese que si el mensaje ya mide 512 bits o un múltiplo de eso, de todas formas se rellena. Si mide 512 bits (64 bytes) se le añadirá el "1", luego 447 ceros ($(512 + 1 + 447) = 960 \equiv 448 \pmod{512}$) y luego el número 512 representado en 64 bits, o sea, se añaden 512 bits a los 512 originales. El mensaje más grande posible para que cupiera en un sólo bloque de 512 bits ya con todo y el relleno es de 446 bits (55 bytes).

Si por ejemplo el tamaño de M fuera de 70 bytes, es decir 560 bits. Entonces $k = 399$ dado que: $560 + 1 + 399 = 960 \equiv 448 \pmod{512}$.

Por cierto, el poner la longitud del mensaje en 64 bits al final, implica que el tamaño máximo de mensaje es de $2^{64} - 1$ bits, es decir algo cercano a 2 mil petabytes[1]. Denotaremos como M al mensaje original con el relleno (*pad*) añadido $M = M_{orig} \| pad$. Podemos considerar a M como una secuencia de n bloques de 512 bits $M = (M_1, M_2, \ldots, M_n)$.

3. Cada bloque de entrada de 512 bits M_i se divide en 16 palabras de 32 bits cada una:
$$M_i = M_i^0, M_i^1, \ldots, M_i^{15}$$

4. Se define el vector de inicialización (VI), al que denotaremos como H_0, que ya tiene la longitud de bloque de salida, es decir, 256 bits. H_0 se define como un vector de 8 entradas de 32 bits cada una: $H_0 = (H_0^0, H_0^1, \ldots, H_0^7)$ donde H_0^i son los primeros 32 bits de la parte fraccionaria de la raíz cuadrada del i-ésimo número primo.

$$\begin{aligned} H_0^0 &= \texttt{6A09E667} \\ H_0^1 &= \texttt{BB67AE85} \\ H_0^2 &= \texttt{3C6EF372} \\ H_0^3 &= \texttt{A54FF53A} \\ H_0^4 &= \texttt{510E527F} \\ H_0^5 &= \texttt{9B05688C} \\ H_0^6 &= \texttt{1F83D9AB} \\ H_0^7 &= \texttt{5BE0CD19} \end{aligned}$$

[1] Un petabyte son 2^{50} bytes.

5. Se definen las 64 constantes $K_0, K_1, \ldots, K_{63}$, de 32 bits cada una, que son las partes fraccionarias de las raices cúbicas de los primeros 64 números primos.

6. Para cada bloque del mensaje M_i ($i \in \{1, \ldots N\}$) se definen las 64 variables:

$$W_t = \begin{cases} M_i^t, & 0 \leq t \leq 15 \\ \sigma_1(W_{t-2}) \oplus W_{t-7} \oplus \sigma_0(W_{t-15}) \oplus W_{t-16}, & 16 \leq t \leq 63 \end{cases} \quad (6.7)$$

7. Se inicializan las variables en función del *hash* obtenido hasta el bloque previo $(i-1)$:

$$\begin{aligned} a &= H_{i-1}^0 \\ b &= H_{i-1}^1 \\ c &= H_{i-1}^2 \\ d &= H_{i-1}^3 \\ e &= H_{i-1}^4 \\ f &= H_{i-1}^5 \\ g &= H_{i-1}^6 \\ h &= H_{i-1}^7 \end{aligned}$$

   La primera vez, cuando $i = 1$ se utiliza el valor del vector de inicialización descrito en el paso 4.

8. Para $t = 0$ y hasta $t = 63$:

$$\begin{aligned} T_1 &= h + \Sigma_1(e) + Ch(e, f, g) + K_t + W_t \\ T_2 &= \Sigma_0(a) + Maj(a, b, c) \\ h &= g \\ g &= f \\ f &= e \\ e &= d + T_1 \\ d &= c \\ c &= b \\ b &= a \\ a &= T_1 + T_2 \end{aligned}$$

9. Calcular el i-ésimo valor intermedio de H:

$$\begin{aligned} H_i^0 &= a + H_{i-1}^0 \\ H_i^1 &= b + H_{i-1}^1 \\ H_i^2 &= c + H_{i-1}^2 \\ H_i^3 &= d + H_{i-1}^3 \\ H_i^4 &= e + H_{i-1}^4 \\ H_i^5 &= f + H_{i-1}^5 \\ H_i^6 &= g + H_{i-1}^6 \\ H_i^7 &= h + H_{i-1}^7 \end{aligned}$$

10. El resultado final es el N-ésimo valor de la función de *hash*, es decir la concatenación de:

$$H_N^0\ H_N^1\ H_N^2\ H_N^3\ H_N^4\ H_N^5\ H_N^6\ H_N^7$$

Debido a la popularidad de la construcción de Merkle-Damgård el número de ataques contra ella es relevante y siguiendo la premisa de "no poner todos los huevos en una sola canasta" el NIST de los Estados Unidos decidió no tener solamente un estándar de funciones *hash* (SHA2) basado en esta construcción. Fue así que surgió SHA3.

7. Keccak

Keccak es una función de *hash* con un diseño diferente al predominante Merkle-Damgård. Constituye la base del estándar SHA-3, de hecho el estándar es una versión reducida de Keccak. Fue diseñado por Guido Bertoni, Joan Daemen, Michaël Peeters, y Gilles Van Assche y aprobado como estándar en agosto de 2015. Keccak funciona en dos grandes fases:

1. Absorción. Durante esta fase, todo el mensaje del que se requiere el *hash* es introducido, bloque a bloque, en Keccak. El mensaje es "absorbido" en el estado S del *hash*.

2. Estrujamiento. Se producen datos de salida, en principio, de longitud arbitraria, aunque en el estándar se limitan a un bloque de 224, 256, 384 o 512 bits.

Antes de que tengan lugar estos procesos el mensaje pasa por un preprocesamiento en el que se rellena el mensaje para garantizar un tamaño múltiplo del tamaño de bloque de entrada de Keccak.

Las fases de absorción y estrujamiento caracterizan lo que se denomina *funciones esponja*.

El *estado* en el que se absorberán los datos de entrada y del que se extraerán los de salida puede ser visto como un arreglo tridimensional como se observa en la figura 7.1. El arreglo está constituido de w rebanadas de 5×5 bits cada una. El valor de w depende de otro parámetro llamado ℓ en el estándar [3]. En la tabla 7.1 se muestran los valores de w en particular y de algunos otros parámetros de operación de Keccak en función de ℓ.

En general:

- El número de rebanadas es $w = 2^{\ell}$.

- El número de bits en el estado es $b = 25\, w$.

- El número de rondas usadas para procesar cada bloque de entrada es $R = 12 + 2\,\ell$.

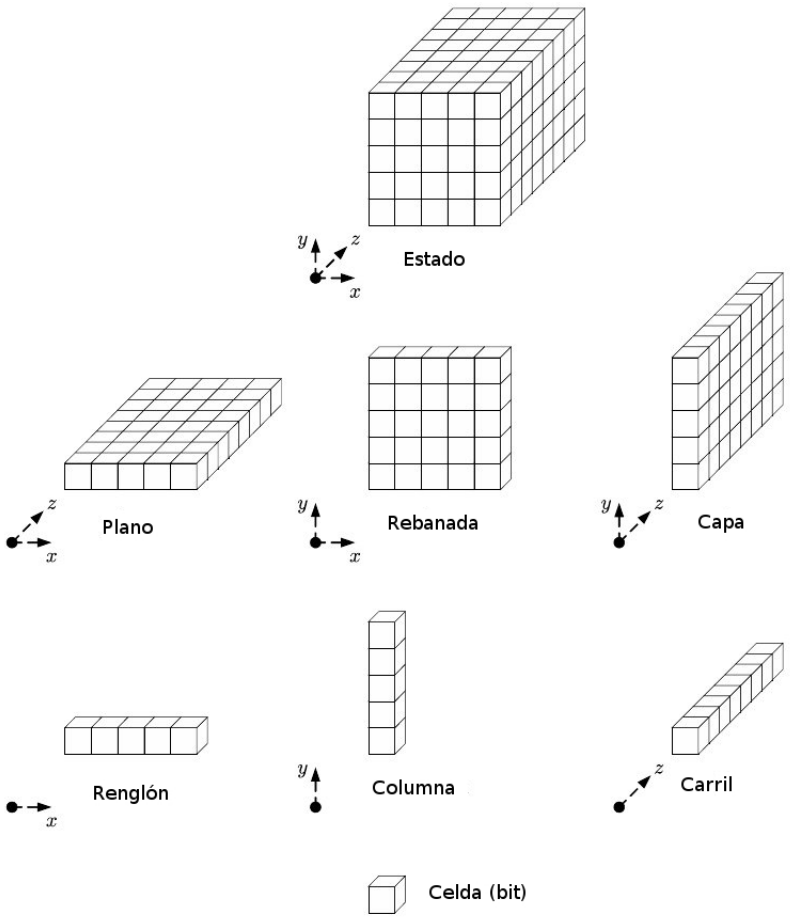

Figura 7.1: Representación y nomenclatura del estado de Keccak.

ℓ	w	b	R
0	1	25	12
1	2	50	14
2	4	100	16
3	8	200	18
4	16	400	20
5	32	800	22
6	64	1600	24

Tabla 7.1: Valores de algunos parámetros de operación de Keccak en función de ℓ: el número de rebanadas $w = 2^\ell$, el número de bits en el estado $b = 25 \times w$ y el número de rondas $R = 12 + 2\,\ell$.

k	c	r	Seguridad	Tamaños de bloque de salida
8	256	1344	128	224, 256
9	512	1088	256	384, 512

Tabla 7.2: Valores posibles de operación de Keccak en SHA-3 suponiendo $b = 1600 = c + r$.

La operación de Keccak depende de un par de parámetros adicionales llamados la *tasa de absorción* r (llamado *bit rate* en inglés) y la *capacidad* c (*capacity* en inglés).El parámetro r determina el tamaño de bloque de datos a ser procesados como entrada de una vez. La capacidad es una medida de la garantía de seguridad de Keccak, este parámetro es decidido en principio por el usuario, se elige de la forma 2^k para poder correlacionarlo con las medidas de seguridad similares en los cifrados de bloque. En la tabla 7.2 se muestran los valores de c para las posibilidades contempladas de k suponiendo que $b = 1600$ ya que debe ocurrir que $b = r + c$. En la tabla, la columna etiquetada "Seguridad" contiene el valor de $c/2$, de acuerdo con nuestro análisis previo usando la paradoja del cumpleaños. El parámetro r queda entonces determinado por la decisión del valor de c.

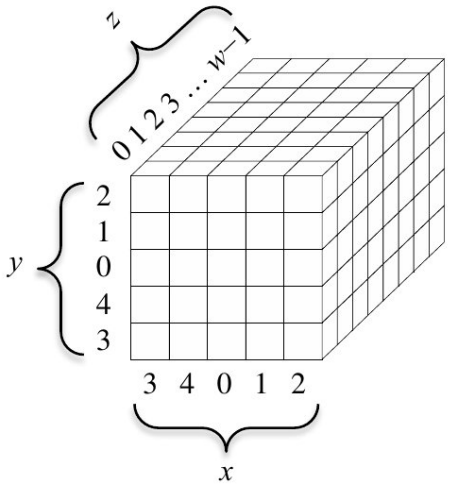

Figura 7.2: Representación del estado S de Keccak.

Siendo r el tamaño de bloque de entrada entonces en la etapa de preprocesamiento el mensaje se rellena con una cadena que garantice que su tamaño es múltiplo de r. De hecho al mensaje se le añade la cadena

$$P = 1|0^j|1$$

(la barra "|" representa concatenación).

Al añadir P al mensaje original de tamaño m, debe resultar un bloque de datos de tamaño múltiplo de r, así que j es el entero más pequeño tal que:

$$m + j + 2 \equiv 0 \pmod{r}$$

Cada bloque de datos p_i de tamaño r del mensaje con relleno incluido deberá entonces ser absorbido en el estado S, que, por cierto, es inicializado en ceros y es tratado como un arreglo tridimensional (véase la figura 7.2). En cada paso de absorción se incorporan los siguientes r bits de la entrada. En la figura 7.3 se muestra esquemáticamente la operación general de Keccak, como se puede observar en ella, la función f, llamada *función Keccak-f* o *permutación Keccak-f* es la operación fundamental.

La función f procesa durante R rondas cada bloque de entrada p_i antes de pasar el estado S actualizado al XOR con el bloque siguiente p_{i+1}. Esta función divide su trabajo en cinco operaciones más simples:

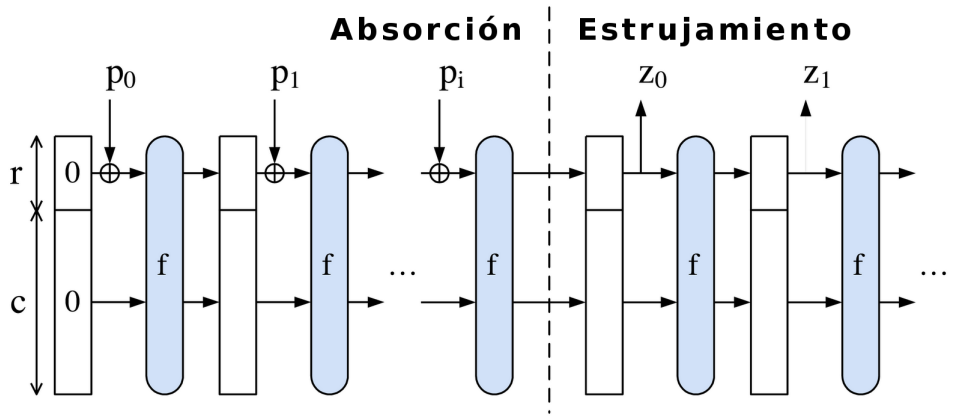

Figura 7.3: Representación esquemática del funcionamiento de Keccak.

Θ

El estado de cada celda (bit) de S se recalcula con base en su valor actual y la paridad de dos columnas, la que está a su izquierda y la que está enfrente y a la derecha. El valor de $S[x][y][z]$ se calcula haciendo el XOR de su valor actual y el de la paridad de la columna con coordenada $x_e = x - 1$ y coordenada $z_e = z$ y el de la paridad de aquella con coordenada $x_b = x + 1$ y $z_b = z - 1$. En la figura 7.4 se muestran gráficamente los operandos. Por supuesto todos los índices son calculados de acuerdo al módulo que corresponda: en el eje z, módulo w y en x y y módulo 5.

ρ

Se rotan los bits de cada carril de cada capa de S un número determinado de posiciones mostradas en la tabla 7.3, claro módulo el número de rebanadas w de S. Por ejemplo para $w = 8$ el efecto se puede ver en la figura 7.5. En el caso más común con $w = 64$ el 153 de la entrada $(x = 3, y = 2)$ se convierte en un 25, ya que $25 = 153 \pmod{64}$.

π

Esta operación consiste en hacer una permutación de los carriles de S:

$$S'[x][y] = S[(x + 3y) \pmod 5][x] \tag{7.1}$$

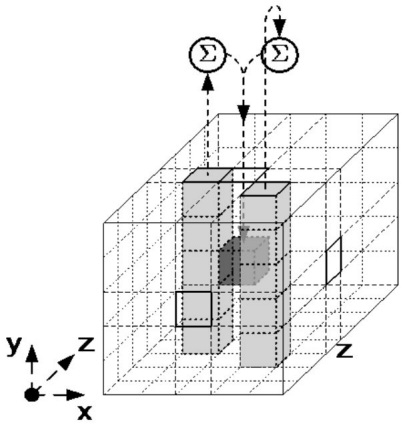

Figura 7.4: Representación esquemática de la función Θ.

	$x=3$	$x=4$	$x=0$	$x=1$	$x=2$
$y=2$	153	231	3	10	171
$y=1$	55	276	36	300	6
$y=0$	28	91	0	1	190
$y=4$	120	78	210	66	253
$y=3$	21	136	105	45	15

Tabla 7.3: Valores de desplazamiento de los bits de acuerdo con la función ρ. Cada entrada de la tabla ha de ser tomada módulo w (la profundidad o número de rebanadas de S).

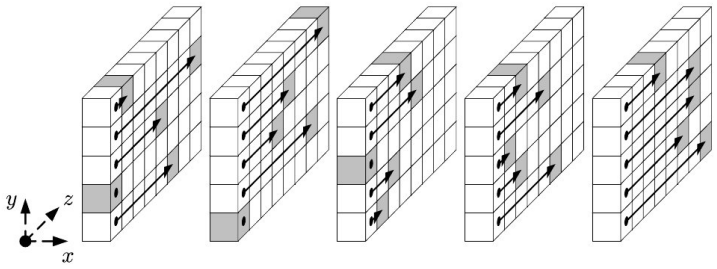

Figura 7.5: Representación esquemática de la función ρ suponiendo una profundidad de 8 bits.

Donde S' es un almacenamiento temporal para la versión actualizada de S. Equivalentemente:

$$S'[y][(2x + 3y) \pmod 5] = S[x][y]$$

El efecto de 7.1 se puede ver en la figura 7.6.

χ

Tomando el almacenamiento temporal usado en el paso previo como entrada, el efecto de esta operación es:

$$S[x][y][z] = S'[x][y][z] \oplus (S'[(x+1) \pmod 5][y][z] \cdot S'[(x+2) \pmod 5][y][z])$$

En la figura 7.7 se puede ver lo que sería el alambrado de la función sobre cada renglón de S.

ι

Esta operación sólo altera el carril en la posición $(0, 0)$ del estado usando para ello las constantes de ronda mostradas en la tabla 7.4. Es decir en la i-ésima ronda ocurre:

$$S[0][0] = S'[0][0] \oplus RC[i]$$

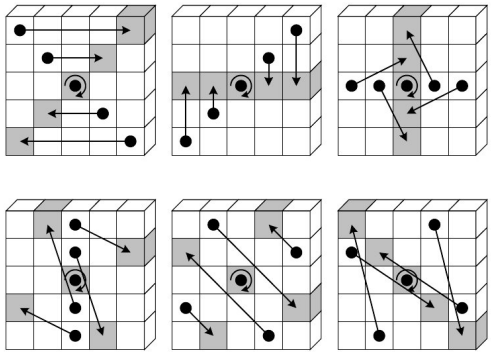

Figura 7.6: Representación esquemática de la función π.

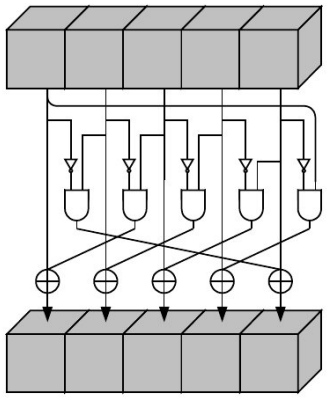

Figura 7.7: Representación esquemática de la función χ.

RC[0]=0x0000000000000001	RC[1]=0x0000000000008082
RC[2]=0x800000000000808A	RC[3]=0x8000000080008000
RC[4]=0x000000000000808B	RC[5]=0x0000000080000001
RC[6]=0x8000000080008081	RC[7]=0x8000000000008009
RC[8]=0x000000000000008A	RC[9]=0x0000000000000088
RC[10]=0x0000000080008009	RC[11]=0x000000008000000A
RC[12]=0x000000008000808B	RC[13]=0x800000000000008B
RC[14]=0x8000000000008089	RC[15]=0x8000000000008003
RC[16]=0x8000000000008002	RC[17]=0x8000000000000080
RC[18]=0x000000000000800A	RC[19]=0x800000008000000A
RC[20]=0x8000000080008081	RC[21]=0x8000000000008080
RC[22]=0x0000000080000001	RC[23]=0x8000000080008008

Tabla 7.4: Constantes de ronda usadas en la aplicación de la función ι expresadas en hexadecimal.

Las constantes de ronda realmente son calculadas con un Registro de Desplazamiento con Retroalimentación Lineal, los conocidos como LFSR por sus siglas en inglés. En esta caso con el polinomio irreducible de alambrado es:

$$x^8 + x^6 + x^5 + x^4 + 1$$

Luego de aplicar R veces estas operaciones sobre un bloque, el resultado, como se puede ver en la figura 7.3, es una de las entradas para la aplicación de las siguientes R rondas sobre el bloque de datos siguiente. Esto continúa hasta que ya no hay más bloques del mensaje original que procesar.

Una vez que toda la entrada es procesada de esta manera, la salida de tamaño r está disponible para ser tomada como el *hash* de la entrada. Pero Keccak puede hacer mucho más. Esta salida puede ser una nueva entrada para la aplicación nuevamente de la función f y este proceso puede continuar indefinidamente dando lugar a una secuencia de bloques de bits pseudo-aleatorios. Esto, claro, excede lo requerido para el estándar de SHA-3, así que no es relevante en ese contexto, sin embargo es potencialmente

útil en contextos diferentes. A esta fase en la que se pueden seguir obteniendo bloques de *hash* sucesivos se le denomina *estrujamiento* (*squeezing*). Lo novedoso de las funciones esponja es que se pueden seguir exprimiendo indefinidamente.

Appendices

A. Recordando a Maclaurin y la exponencial

Recordemos el desarrollo en serie de Maclaurin de una función f alrededor del cero.

$$f(x) = f(0) + f'(0)\,x + f''(0)\,\frac{x^2}{2!} + f^{(3)}(0)\,\frac{x^3}{3!}\ldots \tag{A.1}$$

En el caso de $f(x) = e^{-x}$:

$$\begin{aligned} f(x) &= e^{-x} \quad ; \quad f(0) = 1 \\ f'(x) &= -e^{-x}; \quad f'(0) = -1 \\ f''(x) &= e^{-x} \quad ; \quad f''(0) = 1 \\ f^{(3)}(x) &= -e^{-x}; \quad f^{(3)}(0) = -1 \end{aligned}$$

Así que, alrededor de cero, usando A.1:

$$e^{-x} = 1 - 1\cdot x + \frac{x^2}{2!} - \frac{x^3}{3!}\ldots$$

Si hacemos el cambio de variable $x = \frac{i}{2^n}$ en la expresión anterior, tenemos:

$$e^{-\frac{i}{2^n}} = 1 - \frac{i}{2^n} + \frac{i^2}{2^{2n}\,2!} - \frac{i^3}{2^{3n}\,3!}\ldots$$

Los términos de la suma, claro, se van haciendo cada vez más pequeños. Podemos quedarnos con una buena aproximación así:

$$e^{-\frac{i}{2^n}} \approx 1 - \frac{i}{2^n} \tag{A.2}$$

Bibliografía

[1] Bertoni, G., J. Daemen, M. Peeters y G. van Assche, *The Keccak reference*, 2001. http://keccak.noekeon.org

[2] Knight, William, y D. M. Bloom. "E2386". *American Mathematical Monthly*, 80, no. 10 (1973): 1141-142. doi:10.2307/2318556.

[3] NIST, *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*, FIPS PUB 202, National Institute of Standards and Technology, agosto de 2015. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf

[4] NIST, *Secure Hash Standards (SHS)*, FIPS PUB 180-4, National Institute of Standards and Technology, agosto de 2015. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf

[5] Paar, C. y Pelzl J., *Understanding Cryptography*, Springer, 2010.

[6] Wikipedia contributors, *Comparison of cryptographic hash functions*, Wikipedia, The Free Encyclopedia, 2018. https://en.wikipedia.org/w/index.php?title=Comparison_of_cryptographic_hash_functions

[7] Wikipedia contributors, *Hash function security summary*, Wikipedia, The Free Encyclopedia,2018. https://en.wikipedia.org/w/index.php?title=Hash_function_security_summary&oldid=804965504