Diseño e Implementación de una Función Hash Basada en Caos

César José Ramírez López, Leobardo Hernández Audelo

Laboratorio de Seguridad Informática, Centro Tecnológico Aragón, Facultad de Estudios Superiores Aragón, Universidad Nacional Autónoma de México, Av. Rancho Seco S/N, Col. Impulsora, C.P 57130, Cd. Nezahualcóyotl, Tel. (55)56231070, Estado de México, México.

cesar_jose_540@yahoo.com.mx, leo@servidor.unam.mx

Resumen. En este trabajo se presenta el diseño y la implementación de una Función Hash basada en Caos llamada *Lúthien–Tinúviel* que procesa bloques de mensajes con longitudes menores a 2^{256} bits y entrega un valor hash de 512 bits. El diseño combina mapas caóticos y aritmética de campos finitos de Galois (GF) para realizar el proceso. Para generar las constantes de ronda se utiliza el mapa bidimensional caótico Baker, el atractor de Lorenz y operaciones en GF para terminar dicho proceso. Estas últimas también son utilizadas para procesar el bloque de datos de entrada correspondiente al i-ésimo bloque de datos del mensaje de entrada. Se utilizan dos tipos diferentes de constantes predeterminadas, las cuales son calculadas a partir de la iteración del atractor de Lorenz y son fijas para cada una de las doce rondas que definen la función hash. El diseño está basado en la función hash Whirlpool propuesta por Paulo S.L.L. Barreto y Vincent Rijmen.

Palabras clave: Función hash, mapa Baker, campos finitos de Galois, sustituciones, permutaciones, irreducibles, atractor de Lorenz, no lineabilidad, Whirlpool.

1 Introducción

Actualmente las funciones hash son de trascendental importancia en aplicaciones donde se requiere eficiencia para implementar verificación de integridad y autenticación, a través de la generación de códigos MIC/MAC, entre otras aplicaciones. Es de resaltar el hecho de que la función hash MD5, que se había venido utilizando en la mayoría de aplicaciones para estos fines, ha dejado de ser opción debido a las vulnerabilidades encontradas en los últimos años. Aunque en su lugar se está utilizando SHA con 384 bits, es un hecho que se continúa la búsqueda de una nueva función hash que venga a reemplazar a MD5.

El presente trabajo se enmarca en el contexto anterior y pretende que el diseño de la función hash que en él se describe pueda ser una opción viable para dichos fines, ya que actualmente es una propuesta académica de combinación de características de

mapas caóticos con operaciones de campos finitos. De este modo, en el presente trabajo se describe el diseño y la implementación de una función hash, a la que denominamos "*Lúthien – Tinúviel*" (cuyo nombre hace referencia al personaje de ficción creado por J. R. R. Tolkien en el libro *El Silmarillion* hija del rey Thingol y Melian), la cual procesa mensajes cuya longitud máxima debe ser a lo más de 2^{256} bits, obteniendo como resultado un valor hash de 512 bits.

El diseño de *Lúthien - Tinúviel* está basado en la iteración de una misma función de compresión, la cual procesa bloques del mensaje de entrada de 512 bits a manera de cifrado con una llave autogenerada de 512 bits, mediante 12 rondas.

2 Trabajos Previos

La utilización de mapas caóticos en criptografía se ha enfocado principalmente en operaciones de cifrado de imágenes, pero esto no ha limitado su utilización con respecto a las funciones hash.

En la International Journal of Computer Science and Network Security (IJCSNS), Vol. 8 No. 2, publicada en febrero 2008, Mahmoud Maqableh, Azman Bin Samsudim y Mohammad A. Alia, proponen CHA-1 (Chaos Hash Algorithm-1) una función hash basada en teoría del caos y en SHA-1.

De manera general las principales características de CHA-1 [1] son:

- Longitud de bloque 160 bits
- Longitud del valor hash 160 bits
- Longitud máxima del mensaje 2^{80} bits

Puesto que una función hash puede ser vista como un generador de números pseudoaleatorios, dada la aleatoriedad de la salida requerida, un sistema dinámico es muy adecuado para ser utilizado como motor de la función hash. Sin embargo, no todos los mapas caóticos son adecuados para ello; entre los mapas caóticos más prometedores criptográficamente hablando está el mapa logístico.

El mapa logístico tiene entre sus propiedades el ser simple, rápido, sensible a las condiciones iniciales e impredecibilidad, las cuales se pueden transportar a una función hash. El mapa logístico se define como sigue: $X_n = r(1-X_{n-1})X_{n-1}$; donde $r \in [0, 4]$, $X_n \in (0, 1)$ y $n \in N$.

2.1 Chaos Hash Algorithm-1 (CHA-1)

El diseño de la función Chaos Hash Algorithm – 1 tiene cuatro fases principales:

- Primera fase: Utiliza la función $\boldsymbol{R}$ para producir una salida de 160 bits que será la entrada $\boldsymbol{X_0}$ del mapa logístico en la fase 3.
- Segunda fase: La función $\boldsymbol{X_0}$ es utilizada para producir el valor $\boldsymbol{r}$ que es el segundo parámetro del mapa logístico de la fase 3.
- Tercera fase: Consiste en encontrar un valor real entre [3.592, 4.0] en la que se define el comportamiento caótico del mapa logístico. Los valores iniciales $\boldsymbol{R}$ y

$\boldsymbol{X_0}$ son usados como parámetros de entrada del mapa logístico. CHA-1 toma los primeros 64 bits del resultado de la función $\boldsymbol{R}$ para ser agregados al valor de $\boldsymbol{r}$ para calcular el siguiente valor de $X(X_{n+1})$.

- Cuarta fase: CHA-1 usa el resultado de la fase 3 como valor inicial y recursivamente produce X_n usando un valor de 64 bits a la vez del mensaje original.

2.2 Función X_0

El objetivo de la función $\boldsymbol{X_0}$ es convertir un bloque de 160 bits a un valor real dentro del intervalo [0.33, 0.59] que fungirá como el valor inicial del mapa logístico de la fase 3. Para lograr esto utiliza dos funciones adicionales las cuales producirán los valores $\boldsymbol{X_0}$ y $\boldsymbol{r}$ del mapa logístico.

Su definición es simple ya que usa 3 operaciones no conmutativas *XOR* suma mod 2^{32} y corrimientos circulares a la izquierda de 32 bits. El mensaje de entrada es dividido en bloques de 160 bits; dichos bloques a su vez son divididos en 5 de 32 bits, para formar el buffer *ABCDE* que será procesado por la función $\boldsymbol{X_0}$.

La cual produce una salida de 160 bits a la que se le aplica la operación XOR con el bloque siguiente. El resultado se utiliza como entrada a la siguiente iteración de la función $\boldsymbol{X_0}$. Este proceso continúa hasta que la función termina de procesar el mensaje. Para el caso en el que el último bloque no tenga la longitud requerida se procede a efectuar el relleno concatenando un 1-bit seguido de 0-bit hasta completar los 160 bits necesarios para formar el bloque.

2.3 Función R

La función $\boldsymbol{R}$ utilizada en CHA-1 es similar a la función $\boldsymbol{X_0}$ procesa bloques de 160 bits generando una salida de la misma longitud. Esta salida es sumada *XOR* con el siguiente bloque de entrada; el resultado de esta operación es la siguiente entrada a la función $\boldsymbol{R}$ en la siguiente iteración. Después de procesar todos los bloques del mensaje, el resultado es un bloque de 160 bits. Este valor se utiliza como el valor incremental $\boldsymbol{r}$ en el mapa logístico de la fase 3. Este proceso se repite hasta terminar el mensaje. La función $\boldsymbol{R}$ es una función simple que usa tres operaciones no conmutativas similares a la función $\boldsymbol{X_0}$.

Los primeros 64 bits del bloque de salida de la función $\boldsymbol{R}$ son convertidos a un número real de 160 bits entre 0 y 1. Este número se usa en la fase 3 como el valor incremental $\boldsymbol{r}$ del mapa logístico.

2.4 Finalización

Después de aplicar el mapa logístico en la fase 3 se tiene un valor de 160 bits como salida, el cual se usa como valor inicial para el mapa logístico de la fase 4. En la fase 4 se toman 64 bits del mensaje original (cada vez) para ser usados como el valor incremental $\boldsymbol{r}$. Este proceso se repite con valores de 64 bits hasta que el mensaje se

procese en su totalidad. Al final un valor de 160 bits será un punto en el diagrama de bifurcación y este punto será el valor hash del mensaje.

3 Desarrollo

La función hash *Lúthien – Tinúviel*, es una función Merkle (cf. [2, algoritmo 9.25]) basada en una función de cifrado por bloques a la que se denominará en adelante como *LT-CHA*, la cual opera sobre un bloque de entrada de 512 bits que llamaremos bloque *LT-CHA*. Esta función hash utiliza una llave de estado derivada de los datos de entrada de la ronda en turno. En esta sección se definirán los componentes esenciales que conforman la función hash propuesta, así como los procesos que definen la generación de las llaves.

3.1 Entrada y Salida

El bloque de datos de la función *Lúthien – Tinúviel* consiste de 512 bits, el cual corresponde a un bloque de 64 bytes, los cuales pueden ser agrupados en 8 bloques de 8 bytes. Este bloque se denomina *estado LT-CHA*, y puede ser visto como una matriz $\mu:GF(2^8)^{64} \rightarrow M_{8x8}[GF(2^8)]$ y su inversa:

$$\mu(a) = b \Leftrightarrow b_{ij} = a_{8i+j},\ 0 \leq i, j \leq 7$$

Gráficamente corresponde a lo siguiente:

$$\text{Bloque LT-CHA} = \begin{vmatrix} P_0 & P_1 & P_2 & P_3 & P_4 & P_5 & P_6 & P_7 \\ P_8 & P_9 & P_{10} & P_{11} & P_{12} & P_{13} & P_{14} & P_{15} \\ P_{16} & P_{17} & P_{18} & P_{19} & P_{20} & P_{21} & P_{22} & P_{23} \\ P_{24} & P_{25} & P_{26} & P_{27} & P_{28} & P_{29} & P_{30} & P_{31} \\ P_{32} & P_{33} & P_{34} & P_{35} & P_{36} & P_{37} & P_{38} & P_{39} \\ P_{40} & P_{41} & P_{42} & P_{43} & P_{44} & P_{45} & P_{46} & P_{47} \\ P_{48} & P_{49} & P_{50} & P_{51} & P_{52} & P_{53} & P_{54} & P_{55} \\ P_{56} & P_{57} & P_{58} & P_{59} & P_{60} & P_{61} & P_{62} & P_{63} \end{vmatrix}$$

Tabl 1. Definición del bloque LT-CHA

3.2 Función de Sustitución

La función *Lúthien – Tinúviel* define una sustitución no lineal ($M_{8x8}[GF(2^8)] \rightarrow M_{8x8}[GF(2^8)]$) para cada uno de los elementos que conforman al bloque *LT-CHA*. De manera individual, dicha sustitución puede expresarse de la siguiente forma:

$$SubBytes(a) = b \Leftrightarrow b_{ij} = S[a_{ij}],\ 0 \leq i, j \leq 7$$

Esta función de sustitución se define mediante la siguiente tabla:

		Y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
X	0	96	FD	53	EF	04	1E	2A	E3	AF	E5	A2	3C	B8	D9	2C	11
	1	0A	EB	5F	79	8C	CA	B3	F3	81	FA	31	18	CB	F0	25	CE
	2	A8	FE	58	CF	02	FF	61	4B	9B	D6	C8	4D	74	EC	54	D0
	3	1D	A3	A0	BD	35	93	D1	06	48	4A	D5	77	3F	B0	BA	5D
	4	F9	8F	D2	E1	F1	B9	3A	D4	DC	68	52	82	6D	36	08	C2
	5	10	EA	B6	7A	C9	70	7B	91	E7	E6	AB	BE	F7	85	C5	6A
	6	23	7D	0C	EE	8D	A1	73	AD	47	DD	64	7F	45	B5	AE	B1
	7	89	F4	88	ED	37	BF	66	1F	32	42	F5	09	80	20	03	17
	8	21	16	1A	62	C4	A6	2D	A4	55	2F	01	DA	C0	1C	B7	7C
	9	C3	90	99	E8	84	40	9C	67	6B	4E	C6	5A	A9	6C	BC	2E
	A	A5	00	D8	30	86	41	E0	F8	49	DF	95	FB	60	B2	65	24
	B	5E	8B	DE	D7	78	44	F2	28	26	76	6F	50	4F	6E	98	5B
	C	4C	E4	13	3E	DB	BB	AA	39	1B	57	0D	87	14	AC	0B	D3
	D	0E	05	43	83	9F	97	12	F6	0F	C7	07	63	8A	9A	75	3B
	E	69	3D	A7	59	E9	C1	2B	38	46	33	72	19	9E	71	22	7E
	F	B4	15	FC	E2	27	92	29	CC	9D	34	CD	8E	5C	51	56	94

Tabl2 2. S Box de la función Lúthien – Tinúviel

La tabla de sustitución está calculada con base en el siguiente algoritmo:

- Se obtienen los inversos multiplicativos en GF(2^8) del byte correspondiente, teniendo como polinomio irreducible a $x^8 + x^5 + x^3 + x + 1$.
- Se efectúa la operación matricial consistente en el producto del inverso multiplicativo, obtenido en el paso anterior, representado como un byte, con la matriz *8x8* mostrada a continuación. Al resultado se le suma la constante aditiva $x^7 + x^4 + x^2 + x$ en GF(2^8), la cual corresponde al *0x96* en hexadecimal, con el objetivo de difundir más los bits, teniendo como salida el valor por el cual se sustituirá el byte de entrada.

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \bullet \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Fig. 1. Operación $\Delta(P_i)$

- A esta operación se le denominará $\Delta(P_i)$.

3.3 Funciones de Permutación

En la función *Lúthien – Tinúviel* se definen dos funciones de permutación: una basada en el mapa Baker para la generación de las constantes de ronda y una función

adicional basada en corrimientos a nivel de columna y renglón, utilizada en el procesamiento del bloque *LT-CHA*.

El mapa Baker [3] está descrito con las siguientes expresiones:

$$B(x, y) = (2x, y/2) \quad \text{donde} \quad 0 \le x \le ½$$

$$B(x, y) = (2x - 1, y/2 + ½) \quad \text{donde} \quad ½ \le x \le 1$$

Lo anterior corresponde a la biyección caótica de un cuadrado unitario $|x|$ sobre sí mismo. La columna de la izquierda [0, ½) x [0, 1) es ampliada horizontalmente y reducida verticalmente a [0, 1) x [0, ½). La columna vertical derecha [½, 1) x [0, 1) tiene una correspondencia semejante a la anterior pero sobre [0, 1) x [½, 1).

Basado en la generalización del mapa Baker, se define la permutación utilizada para generar las constantes de ronda de la siguiente manera:

0	1	2	3	4	5	6	7		13	5	22	14	6	23	15	7
8	9	10	11	12	13	14	15		37	29	21	38	30	47	39	31
16	17	18	19	20	21	22	23		61	53	45	62	54	46	63	55
24	25	26	27	28	29	30	31	→	8	0	9	1	10	2	3	4
32	33	34	35	36	37	38	39		24	16	17	18	19	11	20	12
40	41	42	43	44	45	46	47		33	25	34	26	35	27	36	28
48	49	50	51	52	53	54	55		40	32	41	42	51	43	52	44
56	57	58	59	60	61	62	63		56	48	57	49	58	50	59	60

Fig. 2. Permutación basada en el mapa Baker

La anterior corresponde a una de las permutaciones del mapa Baker para una imagen de 8 pixeles cuando los enteros *n* no son divisores de 8. En el presente caso se utilizan los valores 3 y 5.

Después de efectuar dicho mapeo, se procede a realizar una permutación de corrimiento a nivel columna, quedando finalmente definida, en términos de los elementos que componen al bloque de constantes *LT-CHA*, como sigue:

$$\begin{vmatrix} P_{0,0} & P_{0,1} & P_{0,2} & P_{0,3} & P_{0,4} & P_{0,5} & P_{0,6} & P_{0,7} \\ P_{1,0} & P_{1,1} & P_{1,2} & P_{1,3} & P_{1,4} & P_{1,5} & P_{1,6} & P_{1,7} \\ P_{2,0} & P_{2,1} & P_{2,2} & P_{2,3} & P_{2,4} & P_{2,5} & P_{2,6} & P_{2,7} \\ P_{3,0} & P_{3,1} & P_{3,2} & P_{3,3} & P_{3,4} & P_{3,5} & P_{3,6} & P_{3,7} \\ P_{4,0} & P_{4,1} & P_{4,2} & P_{4,3} & P_{4,4} & P_{4,5} & P_{4,6} & P_{4,7} \\ P_{5,0} & P_{5,1} & P_{5,2} & P_{5,3} & P_{5,4} & P_{5,5} & P_{5,6} & P_{5,7} \\ P_{6,0} & P_{6,1} & P_{6,2} & P_{6,3} & P_{6,4} & P_{6,5} & P_{6,6} & P_{6,7} \\ P_{7,0} & P_{7,0} & P_{7,2} & P_{7,3} & P_{7,4} & P_{7,5} & P_{7,6} & P_{7,7} \end{vmatrix} \rightarrow \begin{vmatrix} P_{0,7} & P_{1,5} & P_{0,5} & P_{2,6} & P_{1,6} & P_{0,6} & P_{2,7} & P_{1,7} \\ P_{3,7} & P_{4,5} & P_{3,5} & P_{2,5} & P_{4,6} & P_{3,6} & P_{5,7} & P_{4,7} \\ P_{6,7} & P_{7,5} & P_{6,5} & P_{3,5} & P_{7,6} & P_{6,6} & P_{5,5} & P_{7,7} \\ P_{0,4} & P_{1,0} & P_{0,0} & P_{1,1} & P_{0,1} & P_{1,2} & P_{0,2} & P_{0,3} \\ P_{1,4} & P_{3,0} & P_{2,0} & P_{2,1} & P_{2,2} & P_{2,3} & P_{1,3} & P_{2,4} \\ P_{3,4} & P_{4,1} & P_{3,1} & P_{4,2} & P_{3,2} & P_{4,3} & P_{3,3} & P_{4,4} \\ P_{5,4} & P_{5,0} & P_{4,0} & P_{5,1} & P_{5,2} & P_{6,3} & P_{5,3} & P_{6,4} \\ P_{7,4} & P_{7,0} & P_{6,0} & P_{7,1} & P_{6,1} & P_{7,2} & P_{6,2} & P_{7,3} \end{vmatrix}$$

Fig. 3. Permutación utilizada para la generación de constantes de LT-CHA.

A esta permutación se le denominará $\Pi(P_i)$.

Se define una permutación adicional, la cual será aplicada sólo al bloque *LT-CHA* como parte de la obtención del valor hash durante cada una de las rondas. Dicha permutación se define, en base al bloque *LT-CHA*, como sigue:

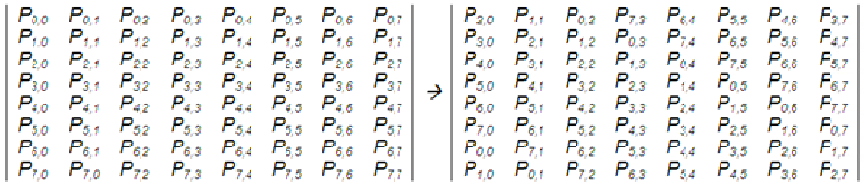

Fig. 4. Permutación definida para el bloque LT-CHA.

3.4 Capa de Difusión Lineal

En esta sección se define el procedimiento mediante el cual se combinan los renglones que conforman el bloque *LT-CHA* con una matriz *A* en GF(2^8), que se define de la siguiente forma:

$$A = \begin{bmatrix} 01_x & 01_x & 03_x & 02_x & 08_x & 04_x & 09_x & 05_x \\ 05_x & 01_x & 01_x & 03_x & 02_x & 08_x & 04_x & 09_x \\ 09_x & 05_x & 01_x & 01_x & 03_x & 02_x & 08_x & 04_x \\ 04_x & 09_x & 05_x & 01_x & 01_x & 03_x & 02_x & 08_x \\ 08_x & 04_x & 09_x & 05_x & 01_x & 01_x & 03_x & 02_x \\ 02_x & 08_x & 04_x & 09_x & 05_x & 01_x & 01_x & 03_x \\ 03_x & 02_x & 08_x & 04_x & 09_x & 05_x & 01_x & 01_x \\ 01_x & 03_x & 02_x & 08_x & 04_x & 09_x & 05_x & 01_x \end{bmatrix}$$

Fig. 5. Matriz de difusión lineal de la función LT-CHA.

Cada byte que conforma el bloque *LT-CHA* se interpreta como un elemento en GF(2^8), y a partir de este se realiza una multiplicación matricial con *A*, de la siguiente manera:

$$\begin{bmatrix} s'_{i,0} \\ s'_{i,1} \\ s'_{i,2} \\ s'_{i,3} \\ s'_{i,4} \\ s'_{i,5} \\ s'_{i,6} \\ s'_{i,7} \end{bmatrix} = \begin{bmatrix} s_{i,0} \\ s_{i,1} \\ s_{i,2} \\ s_{i,3} \\ s_{i,4} \\ s_{i,5} \\ s_{i,6} \\ s_{i,7} \end{bmatrix} \bullet A = \begin{bmatrix} s_{i,0} \\ s_{i,1} \\ s_{i,2} \\ s_{i,3} \\ s_{i,4} \\ s_{i,5} \\ s_{i,6} \\ s_{i,7} \end{bmatrix} \bullet \begin{bmatrix} 01_x & 01_x & 03_x & 02_x & 08_x & 04_x & 09_x & 05_x \\ 05_x & 01_x & 01_x & 03_x & 02_x & 08_x & 04_x & 09_x \\ 09_x & 05_x & 01_x & 01_x & 03_x & 02_x & 08_x & 04_x \\ 04_x & 09_x & 05_x & 01_x & 01_x & 03_x & 02_x & 08_x \\ 08_x & 04_x & 09_x & 05_x & 01_x & 01_x & 03_x & 02_x \\ 02_x & 08_x & 04_x & 09_x & 05_x & 01_x & 01_x & 03_x \\ 03_x & 02_x & 08_x & 04_x & 09_x & 05_x & 01_x & 01_x \\ 01_x & 03_x & 02_x & 08_x & 04_x & 09_x & 05_x & 01_x \end{bmatrix}$$

Fig. 6. Definición de la capa de difusión lineal en LT-CHA.

Como consecuencia de esta operación matricial el primer byte del renglón, por ejemplo, es remplazado por:

$$S'_{i,0} = S_{i,0} \oplus (S_{i,0} \cdot 05x) \oplus (S_{i,2} \cdot 09x) \oplus (S_{i,3} \cdot 04x) \oplus (S_{i,4} \cdot 08x) \oplus (S_{i,5} \cdot 02x) \oplus (S_{i,6} \cdot 03x) \oplus S_{i,7}$$

El símbolo "⊕" expresa la suma en GF(2^8), la cual corresponde a la operación XOR a nivel de bits. Las multiplicaciones indicadas se realizan modulo el polinomio irreducible que, para el caso de *LT-CHA*, es $x^8 + x^5 + x^3 + x + 1$.

3.5 Suma de la Llave de Ronda

La suma de la llave $\sigma[k] : M_{8x8}[GF(2^8)] \rightarrow M_{8x8}[GF(2^8)]$ generada se realiza a nivel de bits (xor), visualizándola como una matriz $k \in M_{8x8}[GF(2^8)]$:

$$\sigma[k](a) = b \Leftrightarrow b_{ij} = a_{ij} \oplus k_{ij}, 0 \leq i, j \leq 7$$

Esta operación se utiliza también para la generación de las constantes de ronda.

3.6 Función de Generación de Llaves de Ronda

En esta sección se definirá el algoritmo utilizado para calcular las constantes de ronda, las cuales fungirán como llaves de ronda para la función *Lúthien – Tinúviel*. Para obtenerlas se utilizará el atractor de Lorenz de la siguiente forma:

ALGORITMO_GENERADOR_DE_LLAVES_DE_RONDA:
ObtieneConstantesDelAtractorLorenz()

Calcula las constantes de ronda a partir del atractor de Lorenz utilizando como punto fijo el establecido, así como la organización y concatenación de los valores obtenidos.

Variables:

1. Punto inicial del atractor de Lorenz: $P_0 = <x, y, z> = <1, 1, 1>$
2. Definición de parámetros: A = 10, B = 28, C = 8/3.
3. Incremento en el valor de los parámetros $\Delta = 0.01$
4. *i, j* = enteros.

Pasos:

1. Se inicializan las variables $i = 0, j = 0$.
2. Se obtiene una lista de 104 constantes de 64 bits agrupadas en 13 grupos de 8.

   – While $j < 104$ do

   - For $i = 0$ to 10 do
     - $x' = a(x - y)$
     - $y' = x(b - z) - y$
     - $z' = xy - cz$
     - $x = x + \Delta x'$
   - Se suprime la parte entera de *x*, *y* e *z*. La parte fraccionaria se multiplica por 100000000 ó $1x10^8$, ($\lfloor a_i \rfloor$ obtiene sólo la parte fraccionaria).
     - $x = 100000000 * \lfloor x \rfloor$
     - $y = 100000000 * \lfloor y \rfloor$
     - $z = 100000000 * \lfloor z \rfloor$

- Se convierte a hexadecimal el valor absoluto de los valores anteriores, con lo que se obtienen 3 constantes de 32 bits en cada iteración.
- Los valores obtenidos se agrupan en bloques de 64 bits, siguiendo el siguiente orden: $x_0 \mid y_0 \mid z_0 \mid x_1 \mid y_1 \mid z_1 \mid \ldots \mid x_n \mid y_n \mid z_n$

3. Regresa los 13 grupos de 8 constantes de 64 bits calculados.

3.7 La Función de Ronda *LT-CHA*

Con los planteamientos anteriormente descritos, la función hash *Lúthien – Tinúviel* se define de la siguiente manera:

$$LTCHA[k] = SubBytes(k) \circ PermutaBytes(k) \circ DifusionLineal(k) \circ AddKey(k^r)$$

Dicho de otra manera: para procesar cada bloque M(i) – bloque i^{th} de 512 bits – es necesario realizar los siguientes pasos:

```
INICIA LT-CHA
  FASE DE PREPROCESAMIENTO DEL MENSAJE (M)
  SE OBTIENEN T BLOQUES DE 64 BYTES
  PARA i = 0 HASTA T HAZ
  INICIO
      // Se forma el bloque LT-CHA(S) con el i^th bloque xor
      // el hash anterior.
      S = M[i] ⊕ H[i-1]  con  H[0] = 0
      //Se asigna la llave inicial del conjunto de Lorenz
      K[0] = H[i-1] ⊕ KR[0][0..7]
      // Se calcula el hash como sigue
      PARA r = 1 HASTA R HACER
      INICIA
        // Primero se calculan las llaves de ronda
        L = Substitución de bytes del bloque LT-CHA (K^r-1)
        L = Permutación Baker del bloque LT-CHA (K^r-1)
        L = Difusión lineal del bloque LT-CHA (K^r-1)
        L = Suma las constantes de ronda (K^r-1, KR^r)

        // Con esto se obtienen las constantes de ronda K^r
        // Se procesa con éstas el bloque LT-CHA
        S = Substitución de Bytes del bloque LT-CHA (S)
        S = Permutación del bloque LT-CHA (S)
        S = Mezcla de renglones del bloque LCH (S)
        S = Suma la constante de ronda (S, K^r)
      FIN PARA
      // Se aplica la operación Miyaguchi-Preneel.
      H[i+1] = H[i] ⊕ S ⊕ M[i]
  FIN PARA
FIN LT-CHA
```

3.8 Relleno y Fortaleza MD

El algoritmo *LT-CHA* sigue lo indicado por R. Merkle y Damgård [3] para la fase de expansión del preprocesamiento. La longitud máxima en bits $\boldsymbol{L}$ de un mensaje $\boldsymbol{M}$ válido, para la función hash Lúthien–Tinúviel (LT-CHA), es $\boldsymbol{L} < 2^{256}$. A partir de este se le concatena un 1-bit, seguido de los 0-bits necesarios para cumplir la condición de formar un bloque cuya longitud sea un múltiplo impar de 256 y, finalmente, una representación binaria de 256 bits, la cual representa la longitud original del mensaje, con lo que se obtiene un mensaje rellenado m, el cual será dividido en t bloques m_1, …, m_t. Estos bloques son vistos como arreglos de bytes por la agrupación secuencial de pedazos de 8 bytes.

4 Pruebas y Resultados

Se realizó un análisis estadístico a la función *Lúthien-Tinúviel* y se compararon los resultados con los mismos resultados conocidos para la función hash *Whirlpool* [3].

4.1 Análisis Estadístico

Para el análisis estadístico se procede, en primera instancia, a generar el espacio muestral definido de la siguiente manera: "*El conjunto formado por los valores hash de las cadenas de 0-bits cuya longitud L esta en el intervalo* $0 \leq L < 2048$". Se obtuvieron los siguientes resultados:

LT-CHA	32.0434	31.91992	31.97412	31.96387	31.93604	31.88379	31.81104	31.92285
Whirlpool	31.9458	31.97217	32.02637	31.97314	31.93359	31.98047	32.1333	32.14746

Tabla 3. Media del número de bits por cada bloque de 64 bits.

LT-CHA	4.024651	4.094875	3.883536	3.937954	3.923625	3.981914	3.978081	4.010595
Whirlpool	3.995419	3.934486	4.006256	3.959112	3.959695	3.970424	4.120773	4.005701

Tabla 4. Desviación media de cada uno de los bloques de 64 bits

Graficando los resultados obtenidos, se observa lo siguiente:

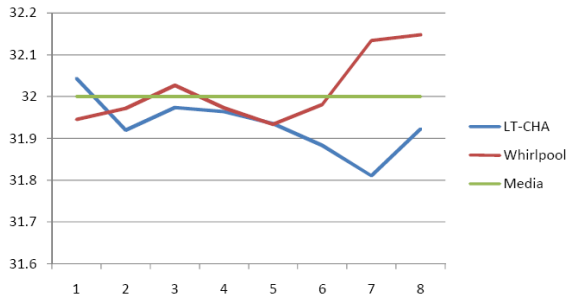

Fig. 7. Media en bits de cada una de las funciones comparadas

Un resultado obtenido de este espacio muestral es que la función *LT-CHA* contiene en promedio 255.4550781 bits con valor 1-bit. En la función Whirlpool la cantidad es de 256.1123 bits; con lo que se podría decir que el hash calculado por Whirlpool tiene al menos la mitad con valor de 1-bit y LT-CHA con valor de 0-bit. Dicha distribución no es uniforme para el *estado* definido, sino que es relativamente diferente entre ambas funciones, como se ve en la gráfica anterior, la cual indica la cantidad de 1-bit para cada bloque de 64 bits. En esta gráfica se observa un comportamiento muy marcado en los bloques 1, 6, 7 y 8, siendo semejante en los restantes.

Graficando la desviación estándar se tiene:

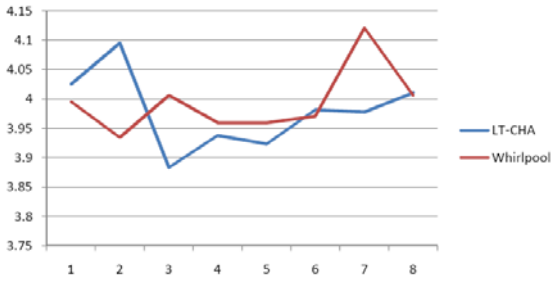

Fig. 8. Desviación estándar para cada una de las funciones hash.

Un comportamiento semejante se obtiene al calcular la desviación estándar del espacio muestral definido.

Resultados semejantes se obtienen al procesar el espacio muestral definido como: "*El conjunto S formado por todos los hash values de las cadenas de 1024-bits conteniendo un solo 1-bit*". Estos resultados son los siguientes:

LT-CHA	31.95605	32.03906	31.82324	32.03809	32.08105	31.87598	32.17578	31.98438
Whirlpool	31.96387	31.80957	32.22949	32.2207	31.9375	32.03027	31.97949	31.93457
Media	32	32	32	32	32	32	32	32

Table 5. Media del número de bits por cada bloque de 64 bits.

LT-CHA	4.031979	3.997367	3.984223	3.966598	3.926357	4.023764	3.979483	4.017023
Whirlpool	3.983813	4.034261	3.982024	4.001479	3.99487	4.138244	3.919424	4.029992

Table 6. Desviación media de cada uno de los bloques de 64 bits

4.2 Pruebas de Rendimiento

Se realizaron pruebas de rendimiento entre las funciones *LT-CHA* y Whirlpool, con el fin de medir el rendimiento de las mismas cuando éstas son aplicadas a bloques de datos de diferentes longitudes.

A continuación se describen las diferentes pruebas de rendimiento que fueron efectuadas para realizar la comparación:

1. "***Procesar un mismo bloque de datos el número de veces necesario para simular un mensaje de longitud L***": En esta prueba no se realizan accesos directos al disco duro, por lo que el tiempo obtenido sólo corresponderá al tiempo que tarda cada función en procesar la cantidad de bloques necesarios para un mensaje de longitud específica.

2. ***"Iterar 100000000 veces la función hash (LT-CHA y Whirlpool) con un mismo bloque de datos"***: El tiempo obtenido corresponderá al necesario para realizar esta operación.

Estas pruebas se realizaron en un equipo con las siguientes características: "AMD Turion 64 Mobile Processor, 1.8 GHz, 512 KB L2 Cache, 1800 MHz de bus de sistema, 1.5 Gb en RAM, 80 Gb de disco duro". Se obtuvieron los siguientes resultados:

DATOS	LT-CHA(s)	Whirlpool(s)
1150 KB	0.0099999998	0.0099999998
2 MB	0.0299999993	0.0199999996
10 MB	0.1400000006	0.1199999973
50 MB	0.7200000286	0.5799999833
500 MB	7.2399997711	5.8499999046
1 GB	14.8299999237	11.9700002670
8 GB	118.2099990845	95.9499969482
40 GB	590.9799804688	479.4299926758

Table 7. Tabla comparativa ***(LT-CHA y Whirlpool)*** para procesar un bloque de longitud L.

Graficando estos resultados obtenemos:

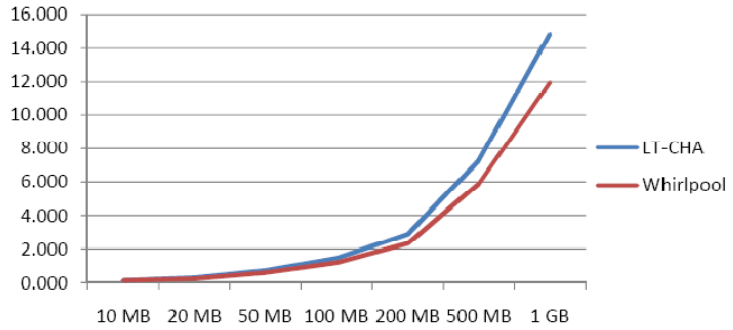

Fig. 9. Rendimiento de Whirlpool y LT-CHA.

En la gráfica se observa que para un conjunto de valores, *LT-CHA* tiene el mismo rendimiento que Whirlpool pero, para bloques de datos mayores a 4MB, el

rendimiento de Whirlpool tiende a ser mejor que *LT-CHA* y esta diferencia se incrementa conforme aumenta la cantidad de bloques a procesar.

Los resultados correspondientes a la segunda prueba son:

No. de Veces	LT-CHA	Whirlpool
100000000	1618.8900146484	1350.4899902344
	1625.2900390625	1348.9399414062
	1620.0000000000	1348.9200439453
	1619.7900390625	1348.6899414062
	1618.3399658203	1350.2800292969

Table 8. Rendimiento en la iteración de las funciones Whirlpool y LT-CHA.

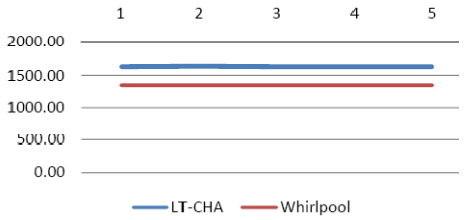

Fig. 10. Comparación en eficiencia al iterar a la función Whirlpool y LT-CHA

Se observa que *LT-CHA* toma más tiempo que Whirlpool en procesar $1x10^8$ veces la función hash aunque este resultado no sea notorio al procesar una pequeña cantidad de bloques de datos.

4.3 Seguridad de la Función de Compresión

El esquema Miyaguchi-Preneel es uno de los pocos métodos sin romper utilizados para la construcción de funciones hash a partir de un cifrado por bloques [4]. Sus propiedades de seguridad son discutidas en [2, sección 9.4.1]. En particular, esta es "probablemente segura" si varias propiedades ideales se mantienen para el cifrado por bloque. Los resultados de desarrollos recientes realizados por Black, Rogaway y Shrimpton [5] acerca de las propiedades de seguridad del esquema Miyaguchi-Preneel y otros esquemas, desde la perspectiva de una caja negra, corroboran de manera cuantitativa la elección realizada para Lúthien-Tinúviel.

5 Mejoras por Realizar

Una característica particular del diseño de la función hash presentada en este trabajo es la utilización de mapas caóticos para la generación de las llaves de ronda. Para realizar esto se utilizan valores precalculados (constantes obtenidas a partir de la

iteración del atractor de Lorenz), además de la definición de un algoritmo para su generación basado en operaciones en GF(2^8). No obstante, la utilización de mapas caóticos es sólo para las constantes de ronda. Una posible mejora al diseño es que también se utilicen estos mapas en el procesamiento del bloque *LT-CHA*.

La utilización de mapas caóticos continuos en la definición de primitivas criptográficas es relativamente reciente debido, en gran parte, a lo costoso en ciclos de reloj del cálculo de valores de punto flotante, ya que la exactitud de los mismos está limitada a la capacidad de poder expresarlos y a la arquitectura utilizada. Por esto, generalmente se utiliza en combinación con funciones no lineales que tengan operaciones de corrimiento, sustitución a nivel de bits a fin de que el cálculo sea lo más "determinístico" posible. Una solución a este problema es la definición de ventanas de rangos de valores que permitan "establecer" el valor calculado previamente y que será utilizado. Esta solución resolvería en parte el problema de la exactitud del cálculo, pero la selección y el orden de elección de los valores debe ser resistente al criptoanálisis diferencial, no debe tener una secuencia lineal y que los valores definidos estén lo más distribuidos posibles en el rango de valores idóneos. Ya que de este rango depende la definición del objeto fractal, se deben considerar los valores que estén fuera de la órbita generada por la definición del objeto, tales como los definidos por el conjunto de Julia.

Cabe destacar que este planteamiento aplica sólo para mapas caóticos continuos, para el caso de mapas discretos no aplica por obvias razones, el planteamiento original de esta propuesta es la posibilidad de utilizar mapas caóticos lineales para el desarrollo de funciones hash.

Una prueba que está en realización es el criptoanálisis diferencial (del algoritmo generación de sub-llaves, junto con el procesamiento del bloque *LT-CHA*) con un conjunto de valores mayor, a fin de determinar si existen o no características diferenciales, así como también determinar su probabilidad, para asegurarse que la definición de la función es resistente a este tipo de ataques.

Se realizó un criptoanálisis diferencial para establecer si se efectuaba o no el mapeo Baker para la generación de constantes y para el procesamiento del bloque *LT-CHA*, llegando a la conclusión de que sólo se utilizaría para la generación de constantes, debido a que al utilizarlo en ambos se tiene un alto grado de lineabilidad.

Otra de las pruebas que se encuentran en realización son las referentes a los ataques del cumpleaños, de primera y de segunda preimagen, a fin de determinar el grado de seguridad que se obtiene. Se realizó un análisis estadístico para determinar el grado de distribución de los valores en los que se observó que la función tiene una distribución más uniforme en los conjuntos de mensajes que se definieron.

6 Conclusiones

En el presente trabajo se presentó el diseño y análisis de la función *Lúthien-Tinúviel*, cuyo diseño se basa en la utilización del mapa Baker, el atractor de Lorenz y en operaciones de GF(2^8), cuyo carácter es académico. Los dos primeros son utilizados como el factor no lineal más importante para la generación de constantes,

tomando ventaja de sus características caóticas no lineales. Las operaciones en GF(2^8) son utilizadas para las operaciones aritméticas no lineales en el procesamiento del bloque LT-CHA. El diseño aquí planteado combina operaciones caóticas con operaciones de campos finitos. Dicha combinación no tiene antecedentes conocidos en la literatura, por lo que puede considerarse innovador en este sentido.

En este mismo sentido el mapa Baker ha sido utilizado en algoritmos de cifrado caótico de imágenes pero no se tiene constancia de su utilización en funciones hash (al menos no encontramos un documento que así lo establezca).

La utilización del mapa Baker tanto en la generación de constantes como en el procesamiento del bloque LT-CHA, genera lineabilidad debido a que si el bloque de entrada presente pocos cambios sólo varía en un bit el mapa Baker solo permuta a los bytes como si fueran los pixeles de una imagen (a nivel de renglón y columna), para las regiones significativas no refleja dicho cambio, al utilizar una permutación a nivel de renglón en su mayor parte los cambios se reflejan mejor, con lo que se pierde dicha lineabilidad.

La combinación de elementos caóticos con operaciones no lineales a nivel de bytes, utilizando operaciones en GF(2^8), permiten mantener una distribución más uniforme de los bits, según lo mostrado en los resultados del análisis estadístico que evalúan la variación con respecto a la media y a la desviación estándar. Esto muestra que el diseño presentado tiene mejores resultados que los obtenidos por Whirlpool.

No obstante, es necesario realizar las pruebas de seguridad faltantes a fin de verificar el nivel de resistencia a colisiones, primera y segunda preimagen. Dependiendo de los resultados que se obtengan en las pruebas que están realizándose, se podrán derivar algunas otras conclusiones sobre el presente diseño.

7 Referencias

1. Mahmoud Maqableh, Azman Bin Samsudim y Mohammad A., "*New Hash Function Based on Chaos Theory (CHA-1)*", en International Journal of Computer Science and Network Security (IJCSNS), vol. 8 No. 2, publicada en febrero 2008

2. A. J. Menezes, P.C. van Oorschot, and S.A. Vanstone, "*Handbook of applied cryptography*", CRC Press, 1997.

3. Van Rompay Bart, "*Analysis and Design of Cryptographic Hash Functions, MAC Algorithms and Block Ciphers*", Tesis para alcanzar el grado de doctor por la Katholieke Universiteit Leuven – Faculteit Toegepaste Wetenschappen Departement Elektrotechniek-Esat, Junio 2004, pp. 240.

4. B. Preneel, Analysis and design of cryptographic hash functions, Ph. D. thesis, Katholieke Universiteit Leuven, January 1993.

5. J. Black, P. Rogaway, and T. Shrimpton, "*Black-box analysis of the block-cipher-based hash-function constructions from PGV*", Advances in Cryptology – Crypto'2002. Lectures Notes in Computer Science, vol. 2442, Springer-Verlag, 2002, pp. 320-335.