UNIVERSIDAD NACIONAL

AUTÓNOMA DE MÉXICO

FACULTAD DE ESTUDIOS SUPERIORES

CAMPUS ARAGÓN

“Diseño e implementación de la función hash caótica Lúthien - Tinúviel”

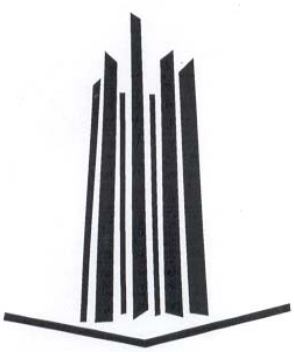
T E S I S

QUE PARA OBTENER EL TÍTULO DE

INGENIERO EN COMPUTACIÓN

P R E S E N T A:

CÉSAR JOSÉ RAMÍREZ LÓPEZ.



ASESOR: M. EN C. LEOBARDO HERNÁNDEZ AUDELO.

SAN JUAN DE ARAGÓN, EDO. DE MEX. 2009

Dedicatoria

A mi madre Claudina: *Por brindarme su amor incondicional y consejo en los momentos difíciles, sin los cuales no me encontraría aquí, además de ser mi ejemplo a seguir. Te amo mamá.*

A mi padre José Guadalupe: *Por apoyarme en todo momento y por enseñarme el significado de ser un hombre. Gracias por todas las enseñanzas brindadas a lo largo de mi vida. Y ser un ejemplo de perseverancia. Te amo papá.*

A mi hermano José Bardomiano: *Porque me ha mostrado que se puede salir adelante, ser mi amigo y mejor compañero de equipo en este juego llamado vida. Gracias por estar ahí, todo mi cariño.*

A Jimena y Peluchin por ser mis fieles amigos que están siempre en todo momento.

Dedicada en especial a Lúthien – Tinúviel por quién fue, por quién es y por quién será. Por supuesto que existe.

A mis amigos:

Claudia Alejandra Coronado Pastor,
Verónica Cruz Rosales,
Lizbeth Castillo Yáñez,
Ramón Navarro Díaz,
Roberto Galicia,
por estar conmigo en los buenos y malos momentos.

A mis camaradas del Colegio de Bachilleres sirva este trabajo, como un pequeño presente en recuerdo a todas las locas ideas y sueños que tratamos de alcanzar en esa época. Recordando que el Bachilleres nos quitó la venda para que descubriéramos todo lo que podíamos alcanzar y de lo que éramos capaces. Presento ante todos ustedes mis respetos.

Santa Cruz Gutiérrez, Antonio Nieto Cuevas, Janet Romero Hernández, Jorge Manuel Bautista, José Gálvez Caballero, Ricardo Sánchez Hernández, Gabriel Moctezuma Guerrero Machuca, Verónica Pérez Figueroa, Alma Delia González Hernández, Alma Rosa Granada Cordero, Francisco Paz González.

"Quieres cambiar tu vida, haz algo. No creas en tus propias racionalizaciones. No te encierres y finjas ser feliz".

"Es precioso soñar, pero con la condición de creer en nuestros sueños. De examinar con atención la vida real, de confrontar nuestra observación con nuestros sueños, y de realizar escrupulosamente nuestra fantasía". Lenin.

Agradecimientos

La palabra más sencilla siempre es la más justa. ***Gracias.***

El presente trabajo es la conclusión de un esfuerzo compartido en cada una de sus diversas etapas.

A mis amigos de la FES Aragón que contribuyeron en mi formación académica. Por soportar todos mis desplantes ideológicos a veces injustificados y charlas filosóficas, así como el apoyo incondicional que nos hemos proporcionado en todos los aspectos de la vida. Muchas gracias camaradas míos.

Sigfrido Pérez Coapango, Carlos Iván Morán Guevara, Carlos Alberto Hernández Mendoza, Juan Pablo Hernández Aguilar, Rodrigo Renteria Anaya, José Martín Morales López, Gonzalo Santos Martínez, Orlando Santaella, Elizabeth Rojas, Elvira Labra Ramos.

A mis compañeros y amigos del LSI por haberme mostrado nuevos horizontes en las diversas áreas de investigación de la seguridad informática. Muchas gracias.

A todos mis compañeros y amigos del equipo Centuriones en los que la convivencia y el acuerdo son las bases fundamentales del trabajo de equipo por que sigamos así y nunca caiga el ánimo.

Por sus sugerencias y consejos para el mejoramiento de este trabajo así como el apoyo, dirección y valioso tiempo brindado para su culminación.

Mat. Luis Ramírez Flores
M. en C. Leobardo Hernández Audelo
M. en C. Ernesto Peñaloza Romero
M. en C. Marcelo Pérez Medel
M. en C. Jesús Hernández Cabrera

Muchas Gracias.

A David Martínez por las facilidades que me proporciono para que mis responsabilidades laborales no interfirieran en la realización de esta meta.

A México para que algún día sea el país que todos aspiramos. "El día que el triunfo alcancemos, ni esclavos ni hambrientos habrá, la Tierra será el paraíso de toda la humanidad".

"La única forma de hacer cualquier cosa bien. La única forma de hacer que tu vida importe, es vivir más y mejor de lo que posiblemente puedas".

"No impidas que tu sentido de la moral te impida hacer lo correcto", Asimov.

“Diseño e implementación de la función hash caótica Lúthien - Tinúviel”

INDICE

Índice de Figuras

Índice de Tablas

Introducción

El hombre –y por ende la civilización- por naturaleza no cambia a menos que su bienestar o su conformación se encuentre en peligro de desaparecer. Debido a esto una necesidad fundamental inconsciente en él es el poseer información veraz para tomar una decisión y a su vez una correcta difusión según sus propios intereses.

De ahí surge la contradicción aparente del término intercambio secreto, la primera implica compartir algo –por necesidad o conveniencia–, la segunda implica ocultar –no difundir de manera masiva lo que se tiene o lo que se sabe–, esto claramente es visible siguiendo lo planteado por Karl Marx en su obra “El Capital” al plantear los diferentes modos de producción –comunidad primitiva, esclavismo, feudalismo, capitalismo y comunismo-, en los que tanto la propiedad de los medios de producción como el conocimiento en el uso o detalles de los mismos se limita a un cierto sector dominante, cuya naturaleza es la de restringir su acceso o en su caso los difunde con el objetivo de producir nuevos para posteriormente apropiárselos.

En los primeros años de la formación del hombre como hombre a partir del mono -cuando nuestros antepasados dejaron las selvas y la seguridad de los bosques, para explorar nuevos territorios- cuando se comenzaba a tener conciencia de sí mismo y del medio que lo rodeaba, se hizo necesaria la formación de tribus para la sobrevivencia en estas todos debían trabajar para todos, cuando se descubre la agricultura y la ganadería surgió la separación de tareas, sin embargo, también era necesario la acumulación de conocimientos esenciales para la vida cotidiana por lo que a los ancianos se les encargo la recabación y el resguardo de los mismos. El paso de conocimientos en esta época era discrecional ya que dependía del buen estado mental como de la buena salud de los ancianos, debido a esto fue necesario crear un sistema que dejará un registro confiable de dicho conocimiento con la invención de la escritura se logra mantener una constancia de las tradiciones y de la cultura de las tribus avanzadas.

La aparición de la escritura tuvo como principal consecuencia el dejar un legado para las generaciones posteriores a manera de enciclopedia, empero, este legado dependía de la exactitud del mismo y por ende de su integridad. Con la aparición de las sociedades esclavistas como evolución de la llamada comunidad primitiva la difusión tanto de los

conocimientos antiguos como de los nuevos se limito a la clase dominante –representantes del clero, nobleza y ejército– e incluso entre estos la difusión era restringida.

Durante esta etapa se tiene el auge de los ejércitos encargados de resguardar el bienestar del país que lo formaba o contrataba según fuera el caso. Por lo que las decisiones tomadas entre los jefes militares y los reyes debían ser restringidas a fin de evitar la divulgación de la estrategia a tomar ya fuera tanto para la defensa como para el ataque de las posiciones. Con lo que surgieron los primeros sistemas de cifrado utilizados para la defensa de los intereses de un estado, un ejemplo de este es el cifrado utilizado por Julius Caesar (100-44 a. C.) el cual utilizaba sustitución simple del alfabeto normal utilizado en las comunicaciones gubernamentales o la traducción entre el latín y el griego.

La seguridad de este se basaba en que en dicha época muy pocas personas sabían leer, empero, la integridad del mensaje se basaba en que se tuviera la misma llave para poder descifrar el mensaje. Para ello se rapaba a esclavos y se grababa en su cráneo la llave a utilizar, la integridad radicaba en que debía llegar con cabello en la cabeza a fin de asegurar que el mensaje llegaba sin haber sido interceptado, como se ve dicho sistema de confianza y de integridad era muy vulnerable.

Con la llegada del feudalismo la “generación” del conocimiento fue monopolizada por la Iglesia, por lo que la difusión del mismo fue acotada y con fines claros de bienestar para sí misma. A la llegada del renacimiento y de los primeros albores del capitalismo se empieza la era de la divulgación acotada del conocimiento así como de su propia protección, para la era del mercantilismo era importante saber lo referente al valor de intercambio de las cosas como de los nuevos descubrimientos de tierras. Debido a esto era necesario manejar con discreción las decisiones gubernamentales con respecto a los nuevos tipos de cambio según conviniera al rey o señor feudal correspondiente.

Con la llegada de la revolución industrial se empiezan a cifrar como en su tiempo hicieron los fenicios los secretos industriales, pero al compartir o transmitir dichos secretos era necesario asegurar la confidencialidad y la integridad de dicho mensaje. Se comenzó con la utilización de sellos con lo que se aseguraba que si el sello estaba roto entonces el mensaje había sido previamente leído, la integridad del sello implicaba también la integridad del mensaje, delegando la confidencialidad del mensaje al correo encargado de entregarlo. Este sistema fue utilizado tanto en la edad media como en la edad moderna en los albores

del capitalismo actual, como correspondencia entre las cortes y como órdenes con los frentes militares en las expediciones de los países-reinos.

Después de la industrialización de los países capitalistas surgieron las guerras cuyo fin era la obtención de áreas geográficas estratégicas para la obtención de materia prima para sus industrias o para el aseguramiento de mercados. Las mayores guerras se presentaron en Europa entre ingleses, prusianos, germanos, franceses, italianos y rusos entre las que destacan la 1ª y 2ª guerra mundial. En la 1ª guerra mundial los rusos combatieron como un estado feudalista para la 2ª guerra combatieron como un estado comunista. Durante la 2ª guerra mundial tuvieron auge las máquinas cifradoras entre las que se encuentran la Enigma, Typex y Blue. En esta época la criptografía tiene un gran auge y por ende el criptoanálisis ya que a partir de mensajes correctamente descifrados y a tiempo podrían hacer la diferencia en cuanto a las acciones militares.

Con la formación del primer estado comunista (Rusia a partir del antiguo imperio ruso) y con la posterior formación de la URSS (Unión de Repúblicas Socialistas Soviéticas) por un lado y con el mundo occidental orientado al capitalismo por el otro, se desarrollo el llamado período de la guerra fría en la que no solo ambas potencias y modelos competirían en la carrera armamentista, social, deportivo, económico sino también en cuanto a lo cultural y científico. Durante esta época cada bloque interviene de tercera forma en cada uno de los conflictos bélicos que surgen durante este período, el flujo de información debe ser asegurado contra espías del enemigo así como su integridad.

Hacen su aparición los sistemas modernos de cifrado en los que no solo se basan en operaciones de permutación y sustitución sino que se empiezan a incorporar conceptos matemáticos más avanzados teoría de grupos y campos.

Cada bloque económico formaría su agencia de seguridad propia (KGB y NSA) cuya tarea principal es la de espiar al enemigo e informar de sus movimientos a su respectivo gobierno, así como de investigar nuevas técnicas para realizar la confidencialidad, la integridad y la autenticidad de la información gubernamental. Surgen las valijas diplomáticas para el intercambio de información de manera segura entre cada una de las naciones la integridad de la información es asegurada con la integridad de la valija.

Con la disolución de la URSS la carrera entre ambas potencias es detenida y las ex-repúblicas soviéticas comienzan a mezclar el modelo soviético con el capitalismo con sus respectivas consecuencias.

Una de las consecuencias de la guerra fría fue la evolución de los sistemas digitales –la cual jugó un papel importante para la investigación del espacio y la llamada carrera espacial– la aparición de las computadoras y su posterior comunicación vía la Internet hizo posible una mejor distribución de la información así como de incrementar los escenarios y los servicios utilizados por las organizaciones.

Actualmente el uso de la computadora ha permitido una gran digitalización de la vida, así como la llamada virtualización de los bienes, un mayor control de la información y grandes avances en lo referente a la investigación en campos inexplorados hasta hace poco tiempo como fue el caso de los fractales en la que los matemáticos fueron capaces de verificar sus teorías mediante el uso de la computadora.

En lo referente al ámbito social, cada individuo tiene un registro en algún sistema computarizado, el cual va desde el folio del acta de nacimiento, los respectivos a cada una de las diferentes identificaciones que se pueden poseer, el número de contribuyente, de seguridad social, la matricula del servicio militar, el número de pasaporte, el de cuenta bancaria e incluso el folio del acta de defunción. Con esta digitalización de la vida se hace necesario asegurar la confidencialidad de la información que proporcionamos por parte de las entidades a las cuales se las confiamos, así mismo, dichas entidades deben ser capaces de asegurar que la información que tienen sea integra y auténtica.

Por tanto los dos mayores requerimientos exigidos (tanto por individuos como por organizaciones) actualmente son la confidencialidad de la información -considerando a está el activo más importante dentro de la toma de decisiones- que proporcionamos o que poseemos y la autenticidad de está. Este último requerimiento tiene dos aspectos ya que se debería checar quien es el autor de cierta pieza de información (autenticación del origen) y que esta no haya sido modificada por alguien (integridad de datos).

Anteriormente se utilizaba una combinación de medios físicos junto con entidades de confianza, antiguamente como mencionamos para documentos en papel era necesario sellar los sobres o cartas que contenían dichos documentos (lo cual permitía la detección

de una intrusión o de alteración no autorizada) o encerrarlos en un lugar seguro (lo que prevenía de la intrusión) logrando la confidencialidad y la integridad en un solo paso, la autenticidad de la información era verificada con el juego de sellos y firmas que contenía el documento.

Lo mismo puede ser aplicado para la protección de las comunicaciones entre personas cara a cara, ya que al ver a la persona el origen de la información se verifica y con las debidas precauciones se puede lograr la confidencialidad de la misma evitando la intromisión.

Empero, durante muchos siglos se han utilizado técnicas criptográficas para proteger los secretos militares y diplomáticos. La mayoría de estas técnicas son esquemas de cifrado los cuales convierten un mensaje en un criptograma mediante una operación invertible (denominada cifrado) dependiente de una pequeña pieza de información secreta (denominada la llave). El criptograma es ininteligible para una persona no autorizada que lo intercepte, pero puede ser reconvertida (descifrada) en el mensaje por el receptor autorizado quien posee el conocimiento compartido la llave.

A lo largo de la historia, la criptología se ha enfocado principalmente al problema de la confidencialidad de la información junto con la autenticidad de la misma. La integridad de la información era delegaba a los medios físicos disponibles dicha protección dependía fuertemente del esquema de cifrado y de la forma en que se utiliza.

Actualmente la criptografía se puede dividir en tres grandes ramas las cuales son:

- Funciones de cifrado y descifrado las cuales permiten implementar la confidencialidad.
- Funciones hash las cuales permiten la verificación de la integridad de un bloque de información durante la transmisión.
- Funciones de firma digital las cuales permiten implementar el no repudio de una operación generada.

En este trabajo de investigación nos enfocaremos solamente a las segundas y propondremos una nueva función basada en sistemas con características caóticas. Como mencionamos anteriormente las funciones hash son las encargadas de verificar la integridad de la información, para esto la entidad emisora debe generar una huella que sea

característica del mensaje, a su vez la entidad receptora debe ser capaz de producir la huella correspondiente del mensaje recibido y al compararla verificar que sea la misma.

Las funciones hash son de dos tipos, aquellas que dependen de una llave secreta para su cálculo y las que no. Las que utilizan una llave a menudo son conocidas como funciones o algoritmos MAC. Desde el punto de vista criptográfico se presta una mayor atención sobre una clase particular de estas las que no necesitan de una llave, la mayoría de los algoritmos se basa en un diseño introducido por R. Rivest para la función hash MD4.

Al igual que los conceptos de seguridad los conceptos matemáticos también evolucionaron casi de la mano de los avances criptográficos aunque en algunos casos fueron retomados mucho después. En 1832 en la época de la revolución francesa murió Évariste de Galois (Bourg-la-Reine 1811, París 1832), matemático francés que creó la actual teoría de grupos y de campos denominados con su nombre en su honor. Dicha teoría es la base actual de AES (Advanced Encryption Standard) y de Whirlpool una de las funciones hash dentro del portafolio de seguridad del proyecto NESSIE.

Otro matemático importante Benoît Mandelbrot (nació en Varsovia en 1924) acuño el término fractal, como la descripción de un objeto matemático cuya característica es la de encerrar una región del espacio cuya dimensión no es un múltiplo entero. Dicha descripción actualmente ha sido utilizada para crear mundos o paisajes virtuales con base en la repetición de patrones naturales, la descripción de la dinámica poblacional, el modelaje de algunas formas de vida, el posible diagnostico de enfermedades como ejemplo de esto la retinopatía diabética, empero, los fractales son un subconjunto especial de una rama matemática denominada teoría del caos la cual aglomera tanto objetos matemáticos cuya descripción es representada por la iteración de ecuaciones fractales como por la teoría de la incertidumbre, en la que caen todos los sistemas dinámicos.

La aplicación de la teoría del caos para la implementación de algoritmos de cifrado o de funciones hash es relativamente nueva, debido a que en la mayoría de los casos el problema es la discretización del cálculo de los valores dinámicos está es la principal limitante en los sistemas de cómputo actuales, ya que se depende tanto del grado de precisión como de la arquitectura del equipo, empero, su utilización ha tenido un relativo auge en la utilización de mapas caóticos para el cifrado de objeto bidimensionales, tales como una imagen, en la que cada pixel puede ser tratado como un punto en el plano y los

valores de dicho pixel las condiciones iniciales para el procesamiento de dicha región según la ecuación no lineal utilizada.

Para el caso de las funciones hash se tienen una propuesta denominada CHA-1 la cual será mostrada en su momento, en dicha ecuación se utiliza el denominado mapa logístico, las operaciones clásicas de una función hash tales como la sustitución, permutación y generación de una llave, lo interesante de dicha función hash es la utilización del mapa logístico para el cálculo de cada uno de los valores. El problema radica en el mapeo que se debe realizar de la parte real del valor obtenido a un valor discreto para ser utilizados como los valores de los parámetros de la función, como se verá este es el principal problema para la implementación de los algoritmos caóticos en el diseño de funciones en el ámbito discreto.

El objetivo del presente trabajo es realizar un diseño de función hash con características caóticas así como su implementación, para lograr esto organizaremos el trabajo de la siguiente forma. Como primer instancia se formará un marco teórico en el que proporcionaremos los conceptos más importantes que se utilizarán en el desarrollo del trabajo, los cuales comprenderán tanto los matemáticos como los relacionados a los conceptos de seguridad.

El marco teórico está compuesto de tres capítulos, el primero y segundo estarán relacionados con las funciones hash, el primero hará referencia del concepto de función hash, características principales y ataques conocidos. El segundo hará un recuento de las funciones hash actuales desde un ámbito algorítmico, una descripción general de sus características así como la definición del algoritmo en sí mismo, para el caso de la familia de la función hash SHA-x se especificarán las diferentes formas para obtener el valor hash de acuerdo a la longitud. En el tercer capítulo se establecerán los diferentes conceptos de dimensión, mapas logísticos, objetos fractales y una breve introducción al caos.

En el cuarto capítulo se hará la propuesta de un algoritmo de función hash con características caóticas cuyo valor hash de salida será de 512 bits, en el capítulo cinco se detallará la implementación especifica de dicho algoritmo a partir de lo especificado en el capítulo anterior, los programas utilizados para realizar los diferentes cálculos y para la obtención de las diferentes tablas como cálculos de los irreducibles, permutaciones y

valores en GF, en el seis se efectuarán y realizarán algunas pruebas de carácter estadístico para determinar las principales características de las funciones hash.

Por último como escribió Goethe en el fragmento que a continuación citaremos toda persona debe reconocer tanto las influencias históricas como las contemporáneas ya que quien crea que atisba un grado de absoluta originalidad está perdido, la cuestión importante es darle otro sentido realizar un descubrimiento y difundirlo aunque este llegue a ser incomprendido en su momento, como en su momento lo hizo Galois, Pushkin y Lenin, quienes no solo vivieron su vida sino que dejaron un legado que es nuestro deber continuarlo, en la medida de nuestras posibilidades todos ellos tenían algo común quisieron cambiar su realidad, sin olvidar su propia realidad.

> *"… very little do we have and inclose which we can call our own in the deep sense of the word. We all have to accept and learn, either from our predecessors or from our contemporaries. Even the greatest genius would not have achieved much if he had wished to extract everything from inside himself. But there are many good people, who do not understand this, and spend half their lives wondering in darkness with their dreams of originality. I have known artists who were proud of not having followed any teacher and of owing everything only to their own genius. Such fools!"*
>
> [Goethe, Conversations with Eckermann, 17.2.1832]

1. Introducción al estudio de las Funciones Hash

"Es increíble que la matemática, habiendo sido creada por la mente humana, logre describir la naturaleza con tanta precisión…"

"La mayoría de las ideas fundamentales de la ciencia son esencialmente sencillas y, por regla general pueden ser expresadas en un lenguaje comprensible para todos…"

Albert Einstein

En el presente capítulo explicaremos los conceptos básicos de las funciones hash, abordaremos los diferentes tipos que de éstas existen y proporcionaremos una descripción de los algoritmos actuales, los criterios de diseño y de evaluación de seguridad, así como sus formalidades y su utilización en esquemas para proteger la autenticidad de la información.

Las funciones hash generan como salida un valor de longitud fija en bits, llamado *valor hash* (**hash value**) o solamente *hash*, que es la representación de una entrada de longitud arbitraria cuya característica es ser única. De acuerdo al tipo de función a realizar, ésta debe satisfacer algunos requerimientos para ser utilizada en aplicaciones criptográficas, por ejemplo, proteger la autenticidad de los mensajes enviados a través de un canal inseguro o verificar su integridad durante una transacción. A continuación se explica este concepto de forma grafica:

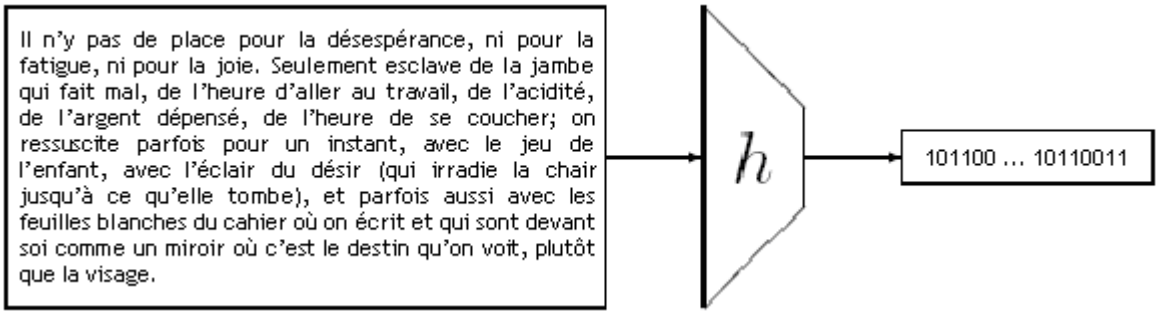


Figura 1. Función Hash

Los códigos de autenticación del mensaje (MAC) están relacionados con las funciones hash, ya que éstos generan, a partir de una entrada de longitud arbitraria, una salida fija en bits, pero para obtenerla necesitan una entrada secundaria de longitud fija, conocida como llave.

Este elemento es la principal diferencia con respecto a las funciones hash, cuyo única entrada es el mensaje, como se mostrará más adelante, puesto que cada tipo de función hash cumple ciertos requisitos y es utilizada para diferentes aplicaciones criptográficas.

Para iniciar con el estudio de las funciones hash, es necesario establecer las principales definiciones para formar de manera adecuada el marco teórico. Durante el desarrollo del presente trabajo se manejarán conceptos en inglés y otros idiomas debido al origen de las citas, sin embargo, esto no debe implicar problemas en su comprensión debido a que es un inglés técnico no muy rebuscado y, para los casos en los que se considere necesaria una interpretación específica, se proporcionará ésta en una forma concreta.

Comenzaremos dando las principales definiciones y conceptos matemáticos utilizados, una breve descripción de los ataques a las funciones hash, las características principales de diseño y algunas de sus aplicaciones.

1.1 Conceptos Matemáticos

En esta sección definiremos los conceptos que serán utilizados en el cuerpo de este trabajo; partiendo del concepto de función y sus tipos; fundamentos de aritmética de campos finitos (empleada por una de las funciones hash y por la que propondremos); hasta el concepto de función hash y de sus propiedades.

1.1.1 Función

Alfred Menezes define a una función de la siguiente manera [1] (pp. 6): "*A function is defined by two sets **X** and **Y** and a rule **f** which assigns to each element in **X** precisely one element in **Y**. The set **X** is called the domain of the function and **Y** the codomain. If **x** is an element of **X** (usually write $x \in X$) the image of **x** is the element in **Y** which the rule **f** associates with **x**; the image **y** of **x** is denoted by y = f(x). Standard notation for a function **f** from set **X** to set **Y** is **f: X→Y**. If $y \in Y$, then a preimage of **y** is an element $x \in X$ for which f(x) = y. The set of all elements in **Y** which have at least one preimage is called the image of **f**, denoted **Im(f)***". [1]

1.1.2 Funciones 1-1

Una función es 1-1 si cada elemento en el codominio ***Y*** es la imagen de *al menos* un elemento en el dominio ***X***.

Una función es inyectiva si cada elemento en el codominio ***Y*** es la imagen de un elemento en el dominio. Equivalentemente, una función ***f: X→Y*** es inyectiva si ***Im(f)= Y.***

Si una función ***f: X → Y*** es 1-1 y ***Im(f) = Y***, entonces ***f*** es llamado una biyección.

[1] Una función es definida por dos conjuntos *X* e *Y*, una regla *f* la cual asigna a cada elemento en *X* de forma precisa un elemento en *Y*. El conjunto *X* es llamado el dominio de la función e *Y* el codominio. Si *x* es un elemento de *X* (usualmente escrito $x \in X$) la imagen de *x* es el elemento en *Y* para el cual la regla *f* asocia a este; la imagen *y* de *x* es denotada por *y=f(x)*. La notación estándar para una función *f* del conjunto *X* al conjunto *Y* es *f:X→Y*. Si $y \in Y$, entonces una preimagen de *y* es un elemento $x \in X$ para el cual *f(x)=y*. El conjunto de todos los elementos en *Y* para el cual se tiene al menos una preimagen es llamada la imagen de *f*, denotado *Im(f)*.

Si ***f: X → Y*** es 1-1 entonces ***f: X → Im(f)*** es una biyección. En particular si ***f: X → Y*** es 1-1, y ***X*** e ***Y*** son conjuntos finitos del mismo tamaño, entonces ***f*** es biyectiva.

Si ***f*** es una biyección de ***X*** a ***Y*** entonces hay una forma simple para definir una biyección ***g*** de ***Y*** a ***X*** como sigue: para cada $y \in Y$ se define $g(y) = x$ donde $x \in X$ y $f(x) = y$. Esta función ***g*** obtenida de ***f*** es llamada función inversa de ***f*** y es denotada como $g = f^{-1}$.

1.1.3 Funciones unidireccionales y unidireccionales con puerta trasera.

Las funciones unidireccionales (en inglés one-way) son utilizadas en criptografía y se definen como:

*"Una función **f** con dominio **X** y codominio **Y** (**f: X→ Y**), es llamada función unidireccional (one-way) si **f(x)** es fácil de calcular para toda **x ∈ X**, pero que a partir de una imagen **y ∈ Y** es computacionalmente infactible encontrar cualquier **x ∈ X** tal que **f(x) = y**".*

Cabe resaltar que la idea de infactible es ambigua, pero aquí la emplearemos en el sentido de que computacionalmente encontrar una solución es muy difícil, es decir, el mejor algoritmo para encontrarla es de tiempo polinomial.

"*Una función unidireccional con puerta trasera (trapdoor one-way) es una función unidireccional **f: X → Y** con la propiedad adicional que dada alguna información extra (llamada información trapdoor) hace factible encontrar para cualquier **y ∈ Im (f)**, una **x ∈ X** tal que **f(x) = y**"*.

A continuación, daremos una introducción a la aritmética de campos finitos que forma parte de la definición de la función hash Whirlpool y de la que propondremos más adelante.

1.1.4 Aritmética de Campos Finitos.

Un campo numérico se define como aquel conjunto ***F*** que define dos operaciones suma (+) y multiplicación (·), que satisfacen las siguientes propiedades matemáticas:

a) (F, +) es un grupo abeliano con la identidad aditiva denominada 0.
b) (F \ {0}, ·) es un grupo abeliano con la identidad multiplicativa denominada 1.
c) Las leyes distributivas se mantienen: (a + b) · c = a·c + a·b ∀ a, b, c ∈ **F**.

Si adicionalmente el número de elementos que conforman el campo **F** es finito, entonces se le denomina como un campo finito[2].

Se define a la operación resta como: para *a, b* ∈ ***F***, *a - b = a + (-b)* donde *–b* es el único elemento en ***F*** tal que *b + (-b)=0* y se le llama el negativo de *b*.

La división es definida en términos de la multiplicación: para *a, b* ∈ ***F*** con *b ≠ 0*, $a/b = a \cdot b^{-1}$ donde b^{-1} es el único elemento en ***F*** tal que $b \cdot b^{-1}=1$ y se le llama el inverso de *b*.

Al número de elementos en un campo finito se le denomina orden. Existe un campo finito ***F*** de orden ***q***; si y sólo si ***q*** es una potencia de un número primo $q = p^m$ donde ***p*** es un número primo y es llamado la característica de ***F*** y ***m*** es un entero positivo. Si ***m = 1***, entonces ***F*** es un campo primo. Si ***m ≥ 2***, entonces es una extensión del campo.

A partir de esto se tiene que para cualquier campo finito cuyo orden sea una potencia de ***q*** existe solamente un campo de orden ***q***, por lo que estructuralmente son lo mismo y se les denomina isomórfico y es denotado como F_q.

Una forma de poder representar los campos finitos y sus extensiones, es a partir de una representación polinomial de la siguiente manera:

Sea ***p*** un primo y ***m ≥ 2***, entonces $F_p[z]$ representa el conjunto de todos los polinomios con coeficientes en F_p. Sea ***f(z)*** un polinomio de grado ***m*** irreducible (que no puede ser factorizado como el producto de dos polinomios de menor grado) con coeficientes en F_p. Los elementos de F_{p^m} son los polinomios en F_p de grado menor a ***m***.

$$F_{p^m} = \{a_{m-1}z^{m-1} + a_{m-2}z^{m-2} + ... + a_2z^2 + a_1z + a_0 : a_i \in F_P\}$$

[2] Estas definiciones se utilizarán cuando se describa a la función Whirlpool.

La suma de los elementos es similar a la de los polinomios salvo que para los coeficientes se utiliza aritmética en F_p. El producto se realiza de la forma usual módulo ***f(z)***, a esta operación se le llama reducción polinomial ***f(z)***.

Campos binarios

Si definimos a ***p = 2*** se obtendrán campos finitos de orden 2^m también llamados campos binarios o campos finitos de característica 2. Utilizando la representación basada en polinomios con coeficientes en $F_2 = \{0,1\}$.

$$F_{2^m} = \{a_{m-1}z^{m-1} + a_{m-2}z^{m-2} + ... + a_2 z^2 + a_1 z + a_0 : a_i \in \{0,1\}\}$$

Se le llama polinomio binario *irreducible* ***f(z)*** de grado ***m*** al polinomio que no puede ser factorizado como un producto de polinomios binarios de grado menor a ***m***. La suma se efectúa utilizando aritmética modular módulo 2 para los coeficientes. El producto es realizado a partir de la reducción polinomial ***f(z)***.

Subcampos de un campo finito

Un subconjunto ***k*** del campo ***K***, es un subcampo de ***K*** si ***k*** a su vez es un campo con las respecto a las operaciones de ***K***, es decir, una extensión de campo de ***k***. Un campo finito F_{p^m} tiene un subcampo de orden p^l para cada divisor positivo de ***m***; los elementos de estos subcampo son los elementos $a \in F_{p^m}$ que satisfacen $a^{p^l} = a$.

1.2 Función Hash

Iniciaremos definiendo a la criptografía como [1] (pp. 4): "*Cryptography is the study of mathematical techniques related to aspect of information security such as confidentiality, data integrity, entity authentication, and data origin authentication".* [3]

Para entender el concepto de función hash se proporcionarán diferentes definiciones, cada una con un cierto grado de formalidad.

[3] La criptografía es el estudio de las técnicas matemáticas relacionadas al aspecto de la seguridad de la información tal como la confidencialidad, integridad de datos, autenticación de entidades y autenticación del origen de los datos.

Una *función hash* puede ser definida de manera simple; como una función ***h*** la cual cumple con las siguientes propiedades:

1. **Compresión** - ***h*** realiza el mapeo de una entrada ***x*** de longitud arbitraria en bits a una salida ***h(x)*** de longitud fija "***n***" en bits
2. **Fácil de calcular** – dada una entrada ***x*** para la función ***h***, ***h(x)*** es fácil de calcular.

Más formalmente se puede definir como: Sea $h : D \rightarrow R$ donde el dominio $D = \{0,1\}*$ y el rango $R = \{0,1\}^n$ para alguna $n \geq 1$.

Una definición más completa incluye el procedimiento para su obtención, estableciéndolo como un proceso iterativo [2] [pp. 173]: *"El hash iterado de la función de compresión* $f : (\{0,1\}^n x \{0,1\}^b) \rightarrow \{0,1\}^n$ *es la función hash* $h : (\{0,1\}^b)^* \rightarrow \{0,1\}^n$ *definido por* $h(X_1 ... X_t) = H_t$ *donde* $H_i = f(H_{i-1}, X_i)$ *para* $1 \leq i \leq t$ *(siendo* $H_0 = IV$*)"*.

1.2.1 Función de Compresión

Una función de compresión puede definirse como: "*Una función cuya entrada es de longitud "**n**" y produce una salida de longitud "**m**", con la condición de que **n > m***".

Más formalmente:

$f : D \rightarrow R$ donde $D = \{0,1\}^a x \{0,1\}^b$ y $R = \{0,1\}^n$ para alguna $a, b, n \geq 1$ con $a + b \geq n$.

1.2.2 Resistencia a una Preimagen.

El concepto "*preimage – resistence*" lo traduciremos como "resistencia a una preimagen" y lo definiremos simplemente como: "*A partir de un valor hash dado, es muy difícil encontrar un mensaje, cuyo valor hash sea éste*".

O de otra forma: "*Dada una Y en Im(h), es muy difícil encontrar un mensaje X tal que h(X) = Y"*.

En las definiciones anteriores no se hace referencia a un elemento importante, el algoritmo encargado de hallar dicha condición, por lo tanto en esta se agregará dicho elemento, así como sus características [2] (pp. 173): *"A hash function h: {0, 1}* → R is **preimage-resistant** of strength (t, ∈) if there exists no probabilistic algorithm I_h that accepts input Y $\in_R$ R and outputs a value X ∈ {0, 1}* in running time at most t, where h(X) = Y with probability at least ∈, assessed over the random choices of both Y and I_h"*.

1.2.3 Resistencia a una Segunda Preimagen.

El concepto "*second preimage – resistance"* lo traduciremos como "*resistencia a una segunda preimagen*" y lo definiremos de la siguiente manera: "*A partir de un mensaje y de su valor hash es muy difícil encontrar otro mensaje diferente tal que los valores hash sean iguales para ambos mensajes*".

O dicho de otra forma: "Dado un $h(X) \in \mathrm{Im}(h)$ y un *X*, es muy difícil encontrar un $X' \neq X$ tal que $h(X) = h(X')$ ".

De la misma forma que en el caso anterior no se hace referencia al algoritmo que trata de alcanzar esta condición, así que una definición más completa sería [2] (pp. 173):

"Let S be a finite subset of {0, 1}. A hash function h: {0, 1}* → R is **second preimage-resistant** of strength (t, ∈, S) if there exists no probabilistic algorithm S_h that accepts input X $\in_R$ S and outputs a value X' ∈ {0, 1}* in running time at most t, where X' ≠ X and h(X') = h(X) with probabilistic at least ∈, assessed over the random choices of both X and S_h"*.

Nótese que uno puede definir una noción de fortaleza de (*second*) *preimage resistant*, en donde el valor *Y* ∈ *R* (o el valor *X* ∈ *S*) se fija a partir de un punto aleatorio, y se maximicen los valores en la totalidad de puntos.

1.2.4 Resistencia a Colisiones.

Se entenderá por una colisión que para dos valores X, X' ∈ {0, 1} se tenga un mismo valor hash*.

Una definición más completa sería [2] (pp. 173): *"A hash function h: {0, 1}* → R is collision-resistant of strength (t, ∈) if no probabilistic algorithm C_h is known that outputs values X, X' ∈ {0, 1}* in running time at most t, where X' ≠ X and h(X) = h(X') with probability at least ∈, assessed over the random choices of C_h"*.

Nótese que dado que *{0, 1}** es infinito y *R* es finito, las colisiones para *h* existen. Y dado que C_h no tiene entradas, existe un algoritmo muy eficiente, tal que genera inmediatamente las salidas *(X, X')* para alguna colisión fija. Sin embargo, para una buena función hash con *R = {0, 1}*^{n}, el mejor algoritmo para encontrar colisiones conocido con alta probabilidad de éxito deberá tener un tiempo de ejecución de alrededor de $2^{n/2}$.

Con las definiciones anteriores establecimos el marco conceptual que utilizaremos a continuación proporcionaremos una clasificación de las funciones hash así como una de sus aplicaciones como lo son las funciones MAC.

1.3 Clasificación de las Funciones Hash

Existen varios criterios de clasificación de las funciones hash pero el principal es dividirlo de acuerdo a los parámetros necesarios para su funcionamiento.

Por consiguiente se puede dividir en:

- Códigos de Detección de Modificaciones (*MDC – Modification Detection Codes*).
- Códigos de Autenticación de Mensajes (*MAC - Message Authentication Codes*).

1.3.1 Códigos de Detección de Modificaciones (MDC)

Los códigos de detección de modificaciones, también conocidos como de detección de manipulaciones (*Manipulation Detection Codes*), son denominados más comúnmente como de integridad del mensaje (*Message Integrity Codes*).

El principal propósito de éstos es proveer un código representativo del mensaje, llamado *hash,* el cual, por lo menos satisface las propiedades anteriores (representación única y fácil cálculo). Este tipo de funciones es muy utilizado junto con otros mecanismos de cifrado

para la verificación de la integridad de datos de acuerdo al nivel requerido por las aplicaciones o servicios que lo utilicen.

Este tipo de funciones pueden dividirse a su vez de la siguiente manera:

- Funciones Hash Unidireccionales (*One-Way Hash Functions - OWHFs*) las cuales deben cumplir con:
    - El argumento ***X*** es de longitud arbitraria y el resultado denotado como ***h(X)*** tiene una longitud fija de "***n***" bits.
    - Debe ser unidireccional, resistente a primera y segunda preimagen.

- Funciones Hash Resistentes a Colisiones (*Collision Resistant Hash Functions - CRFHs*) cumplen con:
    - El argumento ***X*** es de longitud arbitraria y el resultado ***h(X)*** tiene una longitud fija de "***n***" bits.
    - La función debe ser resistente a ataques del tipo de primera y segunda preimagen.
    - Debe ser resistente a colisiones, es decir, la probabilidad de encontrar dos mensajes distintos que produzcan el mismo hash como resultado sea mínima ($X \neq X'$ tal que $h(X) = h(X')$).

Con base en lo expuesto es necesario hacer notar lo siguiente:

Si un atacante puede encontrar una *segunda preimagen*, puede también encontrar una *colisión*, aunque la condición de una *segunda preimagen* en esta definición es redundante. Sin embargo, una *resistencia a preimagen* no siempre implica una *resistencia a colisión*.

La mayoría de las funciones hash son construcciones iteradas, que están basadas en una o más funciones de compresión con entradas de longitud fija, que procesan cada bloque del mensaje. La entrada *X* es rellenada con base en una regla no ambigua para un múltiplo del tamaño del bloque.

Típicamente esto también incluye agregar la longitud total en bits de la entrada. El relleno de la entrada es dividido en *t* bloques denotados como X_1 hasta X_t. La función hash

involucra una función de compresión *f* y un intercambio en la variable H_j, entre el estado *i - 1* y estado *i*:

$$H_0 = IV,$$
$$H_i = f(H_{i-1}, X_i), 1 \le i \le t$$
$$h(X) = g(H_t)$$

Aquí ***IV*** (*Initial Value*) representa el valor inicial (una constante definida para ésta función hash) y ***g*** denota la transformación (opcional) de la salida. Una función de compresión resistente a colisiones puede ser extendida a una función hash también resistente a colisiones tomando entradas de longitud arbitrarias.

1.3.2 Códigos de Autenticación de Mensajes (MAC)

El propósito no formal de una función MAC es verificar la integridad del mensaje así como la autenticidad de la fuente generadora de éste sin usar otro mecanismo adicional. Contrariamente a las funciones hash, el cálculo depende de un elemento adicional, una llave secreta, podemos dar una definición informal de la misma como sigue:

Un código de autenticación del mensaje o MAC es una función ***h*** la cual satisface las siguientes condiciones:

1. La entrada ***X*** puede ser de una longitud arbitraria y el resultado ***h*(*K, X*)** tiene una longitud fija de "***n***" bits. Esta función tiene una entrada adicional, la llave ***K***, con una longitud fija de ***k*** bits.
2. Dado ***h***, ***K*** y una entrada ***X***, el cálculo de ***h*(*K*, *X*)** debe ser fácil.
3. Dado un mensaje ***X*** (pero sin conocer ***K***), debe ser difícil determinar ***h*(*K*, *X*)**. Así mismo, cuando se tiene un conjunto de pares **{X_i, *h*(*K*, X_i)}** conocidos, es difícil determinar la llave ***K*** o calcular ***h*(*K*, *X*')** para cualquier nuevo mensaje **X' ≠ X_i (∀*i*)**.

Siguiendo la misma estructura, a continuación daremos una definición formal [2] (pp. 196):

"A MAC is a function h: K x M → R where the key space K = {0, 1}^{k}, the message space M = {0, 1} and the range R = {0, 1}^{n} for some k, n ≥ 1. When given a key k ∈ K and a message X ∈ M, the function produces a MAC value Y ∈ R"*.

Empero, las funciones hash o las aplicaciones que de estas deriven no están exentas de ataques que exploten alguna de sus propiedades o implementaciones según sea el caso, por lo que en el siguiente apartado mostraremos los más comunes en primer instancia los relacionados específicamente a las funciones hash (exploten una de sus características preimagen, segunda preimagen o colisiones) de forma genérica y posteriormente los ataques que son realizados hacia las funciones MAC.

1.4 Clasificación de Ataques

Aunque el cálculo de las funciones hash *unidireccionales* y *resistentes a colisión* no involucran ninguna información secreta, esto no significa que sea imposible para un atacante realizar ataques con la información disponible, puesto que su meta sería encontrar una *primera* o *segunda preimagen* o incluso generar una *colisión*. La *resistencia a colisiones* no es requerida para todas las aplicaciones, razón por la cual es considerada una categoría aparte de las funciones *unidireccionales*.

La seguridad de una función hash puede ser vista a través del análisis de su función de compresión. Una función resistente a colisiones puede ser extendida a una función hash resistente a colisiones tomando entradas de longitud arbitrarias.

Por otro lado, ataques a las funciones de compresión no necesariamente significan ataques sobre la función hash; un ataque que encuentre *preimágenes* o *colisiones* para ***f*** escogiendo $\boldsymbol{H_{i-1}}$ deja un ataque sobre ***h***, pero un ataque que encuentre *preimágenes* o *colisiones* para ***f*** con un $\boldsymbol{H_{i-1}}$ aleatorio todavía no es un ataque sobre ***h*** (a menos que el ***IV*** pueda ser cambiado).

1.4.1 Ataques sobre las Funciones Hash

En este apartado se mostrarán los ataques genéricos a la funciones hash del tipo *Códigos de Detección de Modificaciones* (*Modification Detection Codes),* a los que a partir de este momento denotaremos como funciones hash o simplemente hash, distinguiéndolos de los *Códigos de Autenticación de Mensajes* (*Message Authentication Codes)* que denominaremos en adelante como funciones MAC.

Básicamente existen dos tipos de ataques sobre este tipo de funciones, los cuales pueden ser aplicados sin importar el diseño de las mismas, ya que se basan en las propiedades generales por definición de una función hash.

1.4.1.1 Ataque de la (Segunda) Preimagen Aleatoria.

Este ataque puede ser aplicado a cualquier función hash y depende solamente del tamaño "***n***" en bits del valor hash resultante. Un atacante simplemente selecciona un mensaje aleatorio y genera un resultado hash. Si la función tiene el requisito de aleatoriedad, su probabilidad de éxito es igual a $1/2^n$. Este ataque puede ser realizado en paralelo, es decir, se espera que pueda ser encontrada una imagen en aproximadamente 2^n operaciones.

1.4.1.2 Ataque del Cumpleaños

Este ataque está basado en la llamada paradoja del cumpleaños en la que la probabilidad de encontrar en un grupo de ***r*** personas que dos de ellas tengan el mismo cumpleaños sea mayor al 50%. El planteamiento de la paradoja es el siguiente:

La probabilidad de que en un grupo de "***r***" personas ninguna de estas tenga el mismo cumpleaños es:

$$q = \frac{365 \cdot 364 \cdot 363 \cdot \ldots \cdot (365 - r + 1)}{365^r} = \prod_{i=0}^{r-1} (1 - \frac{i}{365})$$

Generalizando la idea a un conjunto de ***n*** elementos y de un subconjunto de *r* elementos se tiene [3] (pp. 237):

$$(1-\frac{1}{n})(1-\frac{2}{n})...(1-\frac{r-1}{n}) = \prod_{i=0}^{r-1}(1-\frac{i}{n})$$

Tomando en cuenta que $x = \frac{i}{n}$ es un número real pequeño se considera a $1 - x \approx e^{-x}$. Por lo que expandiendo se tiene lo siguiente:

$$e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \frac{x^4}{4!} - \frac{x^5}{5!}...$$

Lo que implica en la generalización:

$$\prod_{i=1}^{r-1}(1-\frac{i}{n}) \approx \prod_{i=1}^{r-1} e^{\frac{-i}{n}}$$

$$\prod_{i=1}^{r-1}(1-\frac{i}{n}) = e^{\frac{-r(r-1)}{n}}$$

Si consideramos que la probabilidad de que ocurra al menos una colisión es: $1 - e^{\frac{-r(r-1)}{n}}$

Desarrollando para una probabilidad de éxito de ε se tiene:

$$e^{\frac{-r(r-1)}{n}} \approx 1-\varepsilon$$

$$\frac{-r(r-1)}{n} \approx \ln(1-\varepsilon)$$

$$r^2 - r \approx n \ln\frac{1}{1-\varepsilon}$$

Si no consideramos el término *–r* se tiene: $r \approx \sqrt{n \ln\frac{1}{1-\varepsilon}}$

Si tomamos que la probabilidad de éxito sea ε = 0.5 (50%) se tiene: $r \approx 1.17\sqrt{n}$

En el caso específico de la paradoja del cumpleaños se tiene ***n* = 365** entonces ***r* = 23**.

A partir del planteamiento anterior se puede establecer que para encontrar una colisión con una probabilidad de éxito de al menos 50% se debe tener un conjunto de al menos ***r*** elementos. Ya que este ataque depende sólo del tamaño de "***n***" en bits del *hash value*.

Para el caso especifico de una función hash se elige un conjunto de ***r*** entradas *aleatorias* en las que se espera encontrar que al menos dos tengan el mismo *hash value* (es decir

encuentre una colisión). La probabilidad de éxito puede ser calculada remplazando los días del año por |*R*|.

Si consideramos el caso mostrado por Bart Van Rompay [4] (pp. 13) $r = O(\sqrt{|R|})$ y $|R| \to \infty$, entonces la probabilidad de una colisión aproximadamente es: $p \approx 1 - e^{\frac{-r^2}{2|R|}}$

P. Flajolet y A. Odlyzko han mostrado que el número de entradas necesarias para una colisión es aproximadamente:

$$r = \sqrt{\frac{\pi \cdot |R|}{2}}$$

Una variante del ataque del cumpleaños consiste en escoger dos conjuntos distintos y encontrar una colisión entre sus elementos. Si el primer conjunto tiene r_1 elementos y el segundo r_2. Para el caso en el que $r = r_1 = r_2$, $r = O(\sqrt{|R|})$ y $|R| \to \infty$, la probabilidad de una colisión entre los conjuntos es aproximadamente:

$$p \approx 1 - e^{\frac{-r^2}{|R|}}$$

Para $r = r_1 = r_2 = \sqrt{|R|}$ esta probabilidad es cerca de $1 - e^{-1}$ o de 63%.

Basado en esta variante G. Yuval propone un escenario para un ataque práctico. El atacante genera r_1 variaciones sobre un mensaje auténtico y r_2 variaciones sobre uno falso. La probabilidad que hay de que el mensaje auténtico y el falso tengan el mismo hash es cerca del 63% cuando $r = r_1 = r_2 = \sqrt{|R|}$.

Para encontrar una colisión se almacena el hash value correspondiente a un conjunto de mensajes en una tabla. Después de ordenar los elementos (en un tiempo del orden de $O(r \log r)$), se calculan los hash value del otro conjunto hasta que alguno coincida. El principal problema en la implementación práctica es el requerimiento de memoria: el espacio necesario para el almacenamiento es alrededor de *O(r)* mensajes.

Los diferentes tipos de ataques a las funciones hash se pueden clasificar a partir de los requerimientos y del objetivo a conseguir de la siguiente forma [4] (pp. 34):

Tipo de ataque	Requerimientos	Objetivo	Propiedad
Preimage	*H. Y*	*X*	*f(H, X) = Y*
Second preimage	*H, X*	*X'*	*f(H, X) = f(H, X')*
Collision	*H*	*X, X'*	*f(H, X) = f(H, X')*
Pseudo-preimage	*Y*	*H, X*	*f(H, X) = Y*
Random IV collision	*-*	*H, X, X'*	*f(H, X) = f(H, X')*
Pseudo-collision	*-*	*H, H', X*	*f(H, X) = f(H', X)*

Tabla 1 Clasificación de ataques de las funciones hash.

1.4.2 Ataques sobre Funciones MAC

1.4.2.1 Adivinando el MAC

El atacante selecciona un mensaje en particular y posteriormente adivina su valor MAC, el cual se puede utilizar con base en dos suposiciones, propias de la definición de función MAC, *adivina* el valor o *adivina* la *llave* y calcula en consecuencia.

De modo que si la longitud de salida es igual a "***n***" bits y la longitud de la llave secreta es igual a "***k***" bits, la probabilidad de éxito es $p = \max(2^{-n}, 2^{-k})$.

La desventaja de este ataque es que no es posible verificar el resultado debido a que se parte de la suposición de que se adivina correctamente alguno de los dos valores ya sea el MAC o la llave, por lo que este ataque puede ser realizado solamente en línea, por lo que el número de pruebas es dependiente del número de errores permitido por la aplicación.

1.4.2.2 Búsqueda Exhaustiva de la Llave.

Consiste en probar una a una los diferentes valores posibles de la llave hasta que la llave elegida en ese instante sea la correcta. El número de intentos esperados es de $\boldsymbol{2^{k-1}}$ (es decir una búsqueda de al menos la mitad del espacio de llaves), el ataque puede ser verificado cuando ***k/n*** pares de *texto/MAC* están disponibles. A diferencia del ataque anterior éste puede ser realizado fuera de línea con la condición antes mencionada de la obtención de un espacio de ***k/n*** pares para su verificación.

1.4.3 Ataques de Encadenamiento

Los ataques de encadenamiento no son propiamente hacia las propiedades de una función hash, como los anteriores, sino que están más enfocados al diseño de las mismas, principalmente para aquellas funciones hash o MAC que utilizan procesos iterativos para el cálculo de sus respectivos valores de salida intermedios, los cuales se encadenan durante el proceso, dando lugar a las llamadas *hash value* o *MAC value* intermedios o simplemente variables de encadenamiento.

1.4.3.1 Ataque de Conocimiento del Centro

El ataque de conocimiento del centro (en inglés *meet in the middle*) es una variación del ataque de cumpleaños el cual, en lugar de comparar el valor hash, se comparan las variables de encadenamiento intermedias. Este ataque permitiría generar una *segunda preimagen*, lo cual no es posible con un simple ataque de cumpleaños, si el ataque puede ser aplicado a la función hash.

Para realizar este ataque se asume primero la búsqueda de una *segunda preimagen* para una entrada ***X*** dada a la función hash ***f***, considerando que el valor de encadenamiento inicial (***H_0 = IV***) y el valor hash ***h(X)*** son fijos. El criptoanalista debe identificar, en alguna parte, un punto de ataque entre dos bloques de la entrada ***X'*** candidata.

Después debe generar ***r_1*** variaciones sobre la primer parte de un mensaje falso y ***r_2*** variaciones de la última parte, comenzando desde el valor inicial ***IV*** y en sentido inverso al valor hash, la probabilidad para una concordancia de las variables intermedias es $1-\exp(-r_1 \cdot \frac{r_2}{2^n})$.

Una condición necesaria para que este ataque funcione es la capacidad del criptoanalista de invertir en forma eficiente la función de compresión, es decir, dado H_{i+1} encontrar un par H_i, X_i tal que $f(H_i, X_i) = H_{i+1}$ (pseudo-preimagen).

1.4.3.2 Ataque de Bloque Corregido

Este ataque se basa en la búsqueda de una segunda preimagen ***X*** de ***t*** bloques, para una entrada dada. Para realizar esto se elige un bloque X_i de la entrada y es sustituido por un bloque alternativo X_i' de tal forma que $f(H_i, X_i) = f(H_i, X_i')$. Si los otros bloques de la entrada alternativa ***X'*** son iguales a los bloques correspondientes de ***X***, el mismo valor hash puede ser obtenido y por ende una segunda preimagen es encontrada.

Para un ataque de colisión, empezamos con dos mensajes arbitrarios ***X*** y ***X'***; agregamos uno o más bloques corregidos denotados con ***Y*** e ***Y'***, tal que el mensaje extendido ***X*** **||** ***Y*** y ***X'*** **||** ***Y'*** tengan el mismo hash.

1.4.3.3 Ataque del Punto Fijo.

Este ataque es aplicable para aquellas funciones hash en las cuales se tiene como parámetro adicional la variable de encadenamiento, por lo que un punto fijo de ***f*** es un par ***H***, ***X*** para el cual $f(H, X) = H$. Es decir, es posible insertar un número arbitrario de mensajes iguales a ***X*** sin modificar el valor hash, produciendo colisiones o una segunda preimagen si la variable de encadenamiento puede ser igual a ***H***, esto es sólo posible si el valor inicial de encadenamiento no es fijo (el atacante elige ***IV = H***) o si se pueden encontrar un gran número de puntos fijos. Este ataque solo funciona cuando el padding no incluye la longitud de la entrada.

1.4.3.4 Ataque Diferencial

El criptoanálisis diferencial es una poderosa herramienta para el análisis no exclusivo de los cifrados por bloque sino también para las funciones hash. Este método de ataque ejemplifica diferentes entradas para una función de compresión y las correspondientes diferencias de la salida. Una coalición es obtenida si la diferencia es cero entre estas.

1.4.4 Debilidades Analíticas

La gran mayoría de los ataques se basan en la función de difusión del bloque de entrada de datos de la función hash; esto significa que los cambios en la entrada no tienen efecto o pueden ser fácilmente cancelados en el siguiente estado. Aunque la mayoría de esta clase de ataques son los desarrollados por Dobbertin sobre la familia *MDx*. Estos combinan técnicas de optimización convencional (Simulated Annealing, algoritmos genéticos) con técnicas convencionales de criptoanálisis. Otros ataques se basan en las diferencias entre los bloques intermedios de la función de compresión para ciertos puntos específicos del algoritmo; el criptoanálisis diferencial es el más importante de estos, donde típicamente todas las diferencias son controladas por un cierto extent.

1.5 Criterios de Evaluación de Algoritmos Criptográficos.

El proyecto NESSIE (*New European Schemes for Signatures, Integrity, and Encryption*) establece los siguientes criterios de evaluación para los algoritmos de cifrado, de firma digital, funciones hash y MAC:

1. **Nivel de seguridad**: Este nivel es difícil de cuantificar pero existen criterios, tales como el número de operaciones requeridas o ciertos requisitos mínimos en los parámetros de dichas funciones. **NESSIE** maneja los niveles de High, Normal y Normal-Legacy, de acuerdo al tipo de función que se trate.
2. **Funcionalidad**: De acuerdo al diseño de la función cual es la más adecuada para alcanzar el objetivo de seguridad en base a las propiedades básicas que la conforman.
3. **Métodos de operación**: Cuando se aplican en varios caminos y con varias entradas, típicamente presentaran diferentes características, tales como un podría proveer una diferente funcionalidad dependiendo del modo de operación y de su uso.
4. **Rendimiento y eficiencia**: Se refiere a la eficiencia de la función en un particular modo de operación.
5. **Facilidad de implementación**: Se refiere a la dificultad de realizar las primitivas en un uso práctico. Esto podría ser en el entorno de software y hardware.

NESSIE pondera los requerimientos de seguridad [2] (pp. 32) en niveles para cada tipo de función criptográfica, los cuales se enumeran a continuación:

Para las funciones criptográficas que usen llave simétrica:

1. **Cifrados por bloque.**
    a. ***High***: Longitud de llave de al menos 256 bits de bloque al menos 128 bits.
    b. ***Normal***: Longitud de llave de al menos 128 bits de bloque al menos 128 bits.
    c. ***Normal-Legacy***: Longitud de llave de al menos 128 bits de bloque al menos 64 bits.
2. **Cifrado por flujo síncronos.**
    a. ***High***: Longitud de la llave de al menos 256 bits. Memoria interna de al menos 256 bits.
    b. ***Normal***: Longitud de la llave de al menos 128 bits. Memoria interna de al menos 128 bits.
3. **Cifrados por flujo sincronizados por sí mismos.**
    a. ***High***: Longitud de llave al menos 256 bits. Memoria interna de al menos 256.
    b. ***Normal***: Longitud de llave al menos de 128 bits. Memoria interna de al menos 128 bits.
4. **Códigos de autenticación de mensajes (MAC)**. Debe de soportar todas las longitudes de salida (múltiplos de 32 bits) longitud de la llave (inclusive).
    a. ***High***: Longitud de llave de al menos 256 bits.
    b. ***Normal***: Longitud de llave de al menos 128 bits.
5. **Funciones hash resistentes a colisiones**.
    a. ***High***: Longitud de la salida de al menos 512 bits.
    b. ***Normal***: Salida de longitud de al menos 256 bits.
6. **Funciones hash unidireccionales (One-Ways Hash Functions)**: Estas funciones deberán ser resistentes tanto a primera como a segunda preimagen.
    a. ***High***: Salida de al menos 256 bits.
    b. ***Normal***: Salida de al menos 128 bits.
7. **Familia de funciones pseudo aleatorias**: Longitud fija de bloque de al menos 128 bits.
    a. ***High***: Longitud de la llave de al menos 256 bits.
    b. ***Normal***: Longitud de la llave de al menos 128 bits.

Para las funciones asimétricas establece que los parámetros de seguridad deberán ser elegidos de tal forma que el ataque más eficiente requiera un esfuerzo computacional del orden del 2^{80} cifrados 3-DES.

1. **Esquemas de cifrado asimétrico (determinístico o aleatorio)**. El esfuerzo computacional mínimo para un ataque debe ser del orden de 2^{80} cifrados 3-DES.
2. **Esquemas de firma digital**: El esfuerzo mínimo para un ataque debe ser del orden de 2^{80} cifrados 3-DES.
3. **Esquemas de identificación**: El esfuerzo mínimo para un ataque debe ser del orden de 2^{80} cifrados 3-DES. La probabilidad de personificación deberá ser más pequeña que 2^{-32}.

Existen ciertos criterios que deben ser considerados para el buen diseño de una función hash que son comunes para varias de estas. Cabe destacar que estos criterios son genéricos y que no todas las funciones hash lo implementan, pero básicamente todas tienen una estructura similar cuyos elementos son dependientes de la definición de ésta.

1.6 Diseño de Funciones Hash

Una función hash está compuesta básicamente de los siguientes elementos:

- Una entrada de longitud arbitraria de "*n*" bits.
- Padding o función de expansión (o relleno) de la entrada.
- Función de compresión iterada *f.*
- Transformación de salida (opcional).

Estos elementos han sido previamente definidos y son particulares para cada función hash. A continuación, se mostrarán algunas condiciones y requerimientos generales de diseño.

1.6.1 Longitud de Salida requerida

Debido a que los ataques a una función hash (*preimagen aleatoria* y del *cumpleaños*) dependen de la longitud, esta se vuelve un elemento importante en el diseño, ya que su complejidad en el tiempo depende de la longitud del *valor hash*.

Para una función hash con *hash value* de ***n***-bits como salida, es decir, el espacio de salida es representado por $R = \{0, 1\}^n$. Por lo que la función hash tiene una seguridad ideal, si cumple con los siguientes requerimientos:

- Encontrar una preimagen o segunda preimagen toma alrededor de 2^n operaciones.
- Producir o encontrar una colisión requiere de alrededor de $2^{n/2}$ operaciones.

Estos valores corresponden a los valores resultantes de complejidad en el tiempo para los mejores ataques conocidos de preimage aleatoria ($|R| = 2^n$) y del cumpleaños ($r = \sqrt{|R|} = 2^{n/2}$).

Debido a que estos ataques dependen de la capacidad de cálculo y del algoritmo utilizado que implemente dicho ataque, se estima que para que una función hash sea resistente a las colisiones debe tener al menos ***n* = 160** bits y para una función hash unidireccional (la cual necesita ser resistente a los ataques de primera y segunda preimagen) la longitud de salida debe ser de al menos ***n* = 80** bits. Si se quiere tener un nivel de seguridad mayor y por más tiempo se recomendarían longitudes de 192 y 96 bits respectivamente.

1.6.2 Función Hash Iterada

Por definición, la función hash tiene una entrada de longitud arbitraria y una salida de longitud fija ***n***. Debido a esto es necesario crear una función iterativa que permita dividir la entrada en bloques de longitud fija; opere o procese cada bloque y proporcione un resultado final el cual debe ser comprimido o resumido a la longitud establecida como longitud del *hash value*. Esto puede ser expresado como sigue:

Sea una entrada ***X*** compuesta de ***t*** bloques X_0, X_1, X_2,..., X_{t-1} la función debe calcular de forma iterativa el resultado hash de la siguiente forma:

$$H_0 = IV$$
$$H_{i+1} = f(H_i, X_i); 0 \le i \le t$$
$$h(X) = g(H_t)$$

Lo que gráficamente sería:

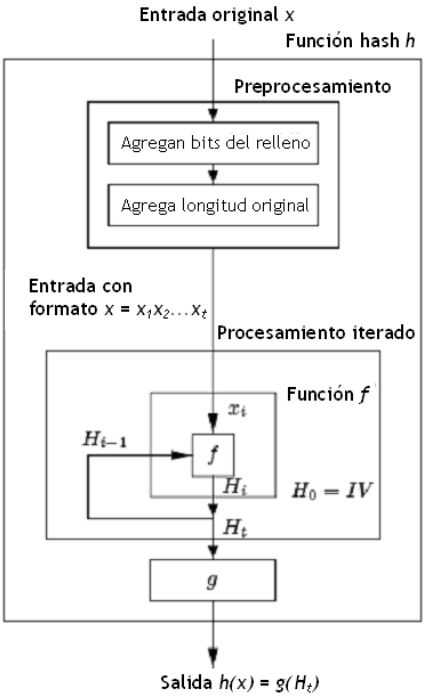


Figura 2. Esquema general de una función hash

Cabe resaltar en el diagrama la sección denominada padding o relleno, de la cual no hemos indicado o mencionado su función durante el proceso. Debido a que la entrada es de longitud arbitraria "***x***" y que la entrada a la función hash iterativa es de longitud fija "***b***", en ocasiones se debe expandir o rellenar la entrada a un múltiplo de "***b***".

Cabe destacar que una forma de conseguir este objetivo que además no permita ambigüedades, es colocar un *1*-bit seguido de la mínima cantidad de *0*-bits necesarios a fin de obtener un múltiplo de "***b***". Sin embargo, esto acarrea un problema de seguridad de la función hash, ya que "*no previene de ataques basados en puntos fijos, en el cual el atacante intenta producir una colisión o una segunda preimagen insertando bloques extras en la entrada*", puesto que sabe dónde termina.

Esta propiedad de resistencia a la colisión fue dada de forma independiente por R. Merkle e I. Damgård. La cual se enuncia como sigue:

"*If the IV is fixed and if the padding procedure include the length of the input into the padding bits, then h is collision-resistant if f is collision-resistant*" [4] (pp. 32).

Debido a lo anteriormente mencionado a la característica de las funciones hash de mantener fijo un ***IV*** como entrada a la función iterativa -además de la concatenación de una representación de la longitud del mensaje, durante el proceso de padding- se le denomina fortaleza – MD (*MD-strengthening)*.

Referente a la propiedad de resistencia de *segunda preimagen* Lai y Massey, discuten la relación entre una función hash y la función de compresión subyacente enunciando lo siguiente:

*"Assume that the **IV** is fixed and that the padding procedure includes the length of the input into the padding bits. Moreover, let the input **X** contains at least 2 blocks (without padding). Then finding a second preimage for **h** requires 2^n operations if and only if finding a second preimage for **f**, with an arbitrary chaining variable input, requires 2^n operations"* [4] (pp. 33).

La razón por lo que es necesaria esta condición es el argumento siguiente: Si toma en promedio 2^s operaciones encontrar una *segunda preimagen* para ***f*** (con ***s < n***), entonces podemos usar un ataque de *meet-in-the-middle* para producir una *segunda preimagen* para ***h*** en alrededor de $2(2^{(n+s)/2})$ operaciones. La descripción de dicho ataque fue mostrado anteriormente.

1.7 Función Hash basada en Bloques de Cifrado.

Un diseño alterno de funciones hash es la utilización de algoritmos de cifrado, los cuales pueden derivar una función hash, con un poco de costo extra (en diseño, evaluación e implementación). Una desventaja es que estas funciones hash son usualmente más pequeñas que las de propósito dedicado (especialmente cuando el cifrado por bloque tiene una calendarización de llave pequeña). Cabe destacar que la utilización de una función de cifrado por bloque en otro contexto podría revelar debilidades de esta que no son relevantes para el cifrado. Otra desventaja a considerar es que son menos eficientes que las funciones hash dedicadas por lo que se debe tener en cuenta este hecho en su utilización.

Las funciones hash basadas en cifrados por bloque se clasifican de acuerdo a la cantidad de bloques de la que se conforma la salida.

1.7.1 Construcciones de un solo Bloque de Longitud

La longitud del valor hash obtenido por este tipo de funciones es igual a la longitud del bloque de cifrado (***n* = *b***). A partir de este hecho se recomienda utilizar funciones de cifrado con una longitud de bloque adecuado suficiente para que sea resistente a colisiones y a los ataques de preimage.

Debido a que las funciones de cifrado no están diseñadas para generar valores hash es necesario utilizarlas de forma adecuada para que los valores resultantes sean resistentes a las colisiones y ataques de preimage, a la forma en que se utilizan los parámetros a la función de cifrado se le llama construcción.

Las construcciones atribuidas a *Matyas-Meyer-Oseas* y *Davies-Meyer* son las más utilizadas. Éstas se definen de la siguiente manera:

En cada paso de la iteración el valor previo de la variable de encadenamiento ($\boldsymbol{H_i}$) sirve como llave y el bloque de mensaje a ser procesado ($\boldsymbol{X_i}$) es el texto en claro de la función de cifrado (o viceversa). La entrada en claro es añadida a la salida de la función de cifrado, la cual realiza una operación de retroalimentación.

El resultado de la retroalimentación es utilizado como el nuevo valor de encadenamiento para la siguiente iteración. Esta relación se muestra a continuación:

- *Matyas-Meyer-Oseas*: $H_{i+1} = f(H_i, X_i) = E_{Z(H_i)}(X_i) \oplus X_i$,
- *Davies-Meyer*: $H_{i+1} = f(H_i, X_i) = E_{X_i}(H_i) \oplus H_i$

Gráficamente representa:

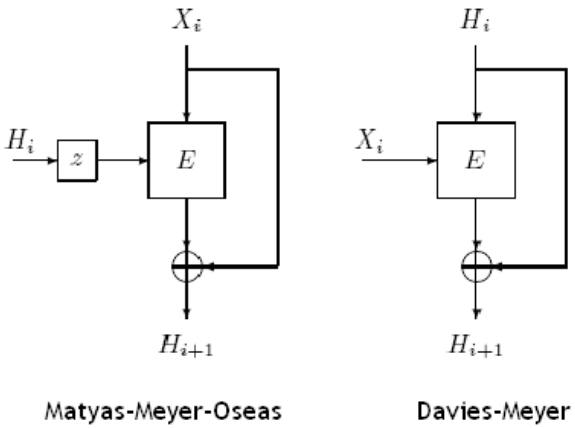


Figura 3. Construcciones de un solo bloque

Cabe hacer notar que estos esquemas no usan una transformación de salida al final, por lo que la longitud de la variable de encadenamiento es igual a la longitud de salida. En caso de que la longitud de la llave del bloque de cifrado es diferente de la longitud de su longitud de bloque, la construcción *Matyas-Meyer-Oseas* necesita una función $z(\cdot)$ que realiza un mapeo del valor de encadenamiento H_i a un conjunto de llaves elegibles para *E* (*z* va de $\{0,1\}^b$ a $\{0,1\}^k$).

El esquema de hash de *Miyaguchi-Preneel* es una variación del *Matyas-Meyer-Oseas*, donde la única diferencia es que la salida $\boldsymbol{H_i}$ del estado previo es también sumada de forma exclusiva a nivel de bits (*XOR*ed) para este estado actual.

- Miyaguchi-Preneel: $H_{i+1} = E_{g(H_i)}(X_i) \oplus X_i \oplus H_i$

Donde ***g*** es la función que realiza el mapeo de $\boldsymbol{H_i}$ a una llave de sustitución para ***E***.

1.7.2 Construcciones de Doble Longitud de Bloque.

La longitud del valor hash resulta muy pequeña -y por tanto es vulnerable a colisiones- al utilizar un mecanismo de cifrado para obtener éste. Por lo que una práctica común es que la longitud del valor hash sea un múltiplo de la longitud de la salida.

Los esquemas más conocidos son: *MDC-2* y *MDC-4*, diseñados por *B. Brachtl*. Donde la función de compresión de *MDC-2* usa dos cálculos paralelos de un esquema *Matyas-Meyer-Oseas*. Estableciendo que C^L y C^R denotan las mitades izquierda y derecha, de un valor de *b* bits *C*, se puede describir de la siguiente forma:

$$H_{i+1} \| \tilde{H}_{i+1} = f(H_i \| \tilde{H}_i, X_i)$$

El cual depende de los siguientes cálculos;

$$C_{i+1} = E_{Z(H_i)}(X_i) \oplus X_i,$$
$$C_{i+1} = E_{Z(\tilde{H}_I)}(X_i) \oplus X_i,$$
$$H_{i+1} = C^L_{i+1} \| \tilde{C}^R_{i+1},$$
$$H_{i+1} = \tilde{C}^L_{i+1} \| C^R_{i+1}$$

Se debe notar que si no se hace el intercambio de las mitades, las dos cadenas serían independientes y podrían ser atacadas de forma separada. No hay ninguna transformación de salida, por lo que la longitud de la cadena de encadenamiento es igual a la longitud de salida ($c = n = 2 \cdot b$). La función de compresión de *MDC-4* tiene dos ejecuciones secuénciales de la función de compresión de *MDC-2*. El nivel de seguridad de estos dos esquemas es menor que el sugerido por el tamaño de la salida.

El mejor ataque conocido de preimage sobre *MDC-2* y *MDC-4* requiere respectivamente de $2^{3n/4}$ y 2^n operaciones. El mejor ataque de colisión toma $2^{n/2}$ operaciones para ambos esquemas. Sin embargo, la complejidad para encontrar pseudo-preimages y pseudo-colisiones para *MDC-2* es de $2^{n/2}$ y $2^{n/4}$ respectivamente. Para encontrar pseudo-colisiones para el *MDC-4* la complejidad es de alrededor de $2^{3n/8}$.

La velocidad es definida como el número de bloques de ***b***-bits de entrada que pueden ser procesados con un solo cifrado, por lo que para *MDC-2* y *MDC-4* la velocidad es de ½ y ¼. Para las funciones hash cuya longitud de la salida es de un bloque la velocidad es 1 (excepto para *Davies-Meyer* la velocidad es igual ***k/n***).

El esquema *Matyas-Meyer-Oseas* y *MDC-2* son incluidos en el ISO/IEC 10118-2, un estándar para funciones hash usando un (no especificado) cifrado por bloque. La función hash *Whirlpool* está basado en el modo de hash de *Miyaguchi-Preneel* de un cifrado por bloque.

1.8 Funciones Hash usando Aritmética Modular.

La aritmética modular puede ser utilizada también para generar un valor hash aprovechando las implementaciones existentes de criptografía de llave pública. Una ventaja de usar aritmética modular es que la seguridad puede ser escalada eligiendo un módulo ***M*** con longitud adecuada. Aunque la principal desventaja es la lentitud para generar el valor hash. Empero, la mayoría de las construcciones propuestas han sido rotas y basada en las experiencias se ha diseñado el *MASH-1* y *MASH-2*.

En la función de compresión de *MASH-1*, el módulo ***M*** tiene una longitud en bits de ***m*** (haciendo infactible factorizar ***M***). El bloque X_{i-1} del mensaje de entrada es primero expandido con redundancia de bits a un bloque $\tilde{X}_{i-1}$ de doble longitud. El bloque $\tilde{X}_{i-1}$ es sumado (or-exclusiva) a nivel de bits al valor previo de encadenamiento H_{i-1} (ambos $\tilde{X}_{i-1}$ y H_{i-1} tienen longitud en bits ***d***, cuyo valor es el máximo múltiplo de 16 menor que ***m***). Se eleva al cuadrado módulo ***M*** este valor, efectuando una retroalimentación con H_{i-1}. Este resultado se muestra en la siguiente ecuación para la función de compresión:

$$H_i = trunca_n(((H_{i-1} \oplus \tilde{X}_i) \vee A)^2 \bmod(M)) \oplus H_{i-1}$$

Donde:

A: Es una constante forzando los 4 bits más significativos a 1, *f00...00*$_x$.

$trunca_n$: Denota truncar el resultado a *n* bits menos significativos.

La seguridad de esta construcción depende en parte de la dificultad de extraer raíces modulares. La redundancia de bits es vital para el nivel de seguridad: la complejidad del mejor ataque conocido para encontrar una preimagen o una colisión es de 2^n.

MASH-2 es una variante de *MASH-1*, la única diferencia es usar una exponenciación modular con exponente $e = 2^8 + 1$ (en lugar del cuadrado modular con $e = 2$). El mejor ataque conocido para encontrar una colisión o una preimage tiene una complejidad de $2^{n/2}$ operaciones. El *MASH-1* y *MASH-2* son incluidas en ISO/IEC 10118-4, un estándar para funciones hash usando aritmética modular. En él también se define una transformación de salida adicional que debería ser usada para reducir la longitud del valor a ***n/2*** bits o menos.

1.9 Funciones Hash dedicadas.

También llamadas de propósito específico ya que en su diseño se incluye la optimización de la obtención del valor hash. Entre estas funciones destacan las basadas en el diseño del algoritmo *MD4*; al conjunto de estas se les denomina familia *MDx*; ya que aunque en 1990 había sido roto (era posible encontrar una colisión), sus descendientes se han beneficiado de este hecho al corregir los problemas encontrados por el criptoanálisis.

Estos algoritmos generalmente dividen el bloque de mensaje X_i y la variable de encadenamiento H_{i-1} en palabras de 32 bits (o 64 bits para alguna de las nuevas propuestas) durante la obtención del valor hash. La función de compresión actualiza la variable de encadenamiento a H_i, utilizando para esto el bloque de mensaje actual a procesar.

El procedimiento para calcular el valor hash se compone de varios pasos, durante los cuales se actualiza el valor de un registro –que contiene una palabra de la variable de encadenamiento- relacionándolo con una del mensaje y con los demás registros para ser procesados por la función de compresión, la cual puede ser dividida en un número de rondas y cada ronda a su vez usa todas las palabras del bloque exactamente una vez.

Como ejemplo describiremos el algoritmo de *MD4* (*Message Digest 4*), utilizando la notación de ecuación por lo que lo definiremos como:

$$A = (A + f(B,C,D) + X[j] + y)^{<<<S[j]}$$

Donde:

A, *B*, *C*, *D*: son los registros conteniendo las cuatro palabras de 32 bits de la variable de encadenamiento.

+ : Notación que implica la adición modular 2^{32}.

$(.)^{<<<s}$: significa un corrimiento (rotación circular a nivel de bits) a la izquierda de "*s*" posiciones.

f : función no lineal a nivel de bits, su definición depende de la ronda *j*.

$X[j]$: es el arreglo de bytes que representa la palabra del mensaje utilizada en ronda *j*.

y : constante aditiva especifica de la ronda.

$s[j]$: especifica la constante de rotación de la ronda (el número de bits a rotar).

Las rondas son invertibles o reversibles, es decir, que se puede calcular al revés (debido a que no se pierden valores y estos son constantes), para evitar esto, al final de la función de compresión hay una operación de retroalimentación, mediante el cual se añade el valor inicial del registro para calcular la variable de encadenamiento H_i. Esto hace a la función de compresión no invertible.

Varios algoritmos han sido derivados de *MD4*, aplicando diferentes ideas para incrementar la seguridad de estos contra ataques de preimagen y evitar en lo posible encontrar colisiones. Entre estas ideas se incluyen: el uso de más rondas, modificar la operación cambios el orden de los pasos y más variables de encadenamiento (lo cual también significa un resultado mayor de valor hash).

Un ejemplo particular de esto es *SHA-1* que usa un procedimiento para expandir el bloque de mensaje para calcular una palabra diferente para cada uno de los pasos (en lugar de justamente reordenar las palabras del mensaje entre las diferentes rondas).

La familia *RIPEMD* usa dos líneas paralelas de cálculo consistiendo de versiones modificadas de *MD4*.

Esquemáticamente se muestra un diagrama el cual muestra la relación de las funciones pertenecientes a la llamada familia *MDx*:

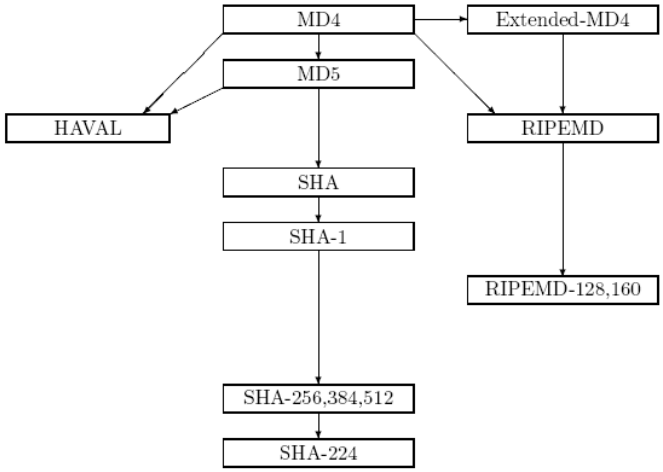


Figura 4. Relación entre funciones hash.

2. Funciones Hash Actuales

"Se debe hacer todo tan sencillo como sea posible, pero no más sencillo…"

"Si buscas resultados distintos, no hagas siempre lo mismo…"
Albert Einstein.

En esta sección describiremos las funciones hash de propósito específico de la familia *MD-x*, *SHA-x* y Whirlpool, ésta última propuesta como parte del estándar *NESSIE*. Las funciones *RIPEMD-x* y HAVAL serán definidas en el apéndice.

El principal objetivo de esta sección es mostrar cómo se aplican las características antes mencionadas de diseño de funciones hash, en cada uno de los siguientes algoritmos mostrados así como información técnica y descriptiva de los mismos.

La descripción de cada una de las funciones comprende la definición de las constantes específicas de cada una de estas, el preprocesamiento previo que realiza cada función hash al mensaje de entrada, la forma en que se realiza y calcula el valor hash especifico de cada uno.

Se mostrarán las principales características de cada una de las funciones hash entre las cuales están el tamaño del bloque de entrada, el número de rondas, la longitud máxima del mensaje de entrada y por último el tamaño del hash value, valores que son dependientes para cada una de las funciones hash que se describirán en este capítulo.

Para las funciones hash que tienen diferente longitud de hash resultante se dividirá la definición, cálculo y forma de obtención del mismo de acuerdo al tamaño definido del valor hash.

En esta sección se harán referencias a documentos denominados como RFC (Request For Comments) los cuales son una serie de notas que comenzaron a publicarse en 1969. Cada uno de estos es en sí un documento cuyo contenido puede estar comprendido en alguna de las siguientes categorías:

- Informativos.
- Propuestas de estándares nuevos.
- Históricos.

Para el caso de que sea una propuesta oficial para un nuevo estándar - ya sea de algún protocolo o algoritmo en específico – este documento RFC debe explicar a detalle las características, propiedades y definiciones del mismo, para que en caso de ser aceptado pueda ser implementado sin ambigüedades.

Cada RFC tiene un titulo y un número asignado que no puede repetirse ni eliminarse, aunque el documento quede obsoleto, estos documentos se redactan en inglés según una estructura especifica y en formato de texto ASCII.

Antes de que un documento tenga la consideración de RFC debe seguir un proceso muy estricto para asegurar su calidad y coherencia. Cuando lo consigue, prácticamente ya es un protocolo formal al que probablemente se interpondrán pocas objeciones, por lo que el sentido de sus nombre como “petición de comentarios” ha quedado prácticamente obsoleto pero que es mantenido por razones históricas.

2.1 MD4

El algoritmo MD4 [6] [RFC-1320][1] será descrito en esta sección, comenzaremos enumerando sus características principales:

- **Entrada**: cadena de bits "*x*" de longitud arbitraria $b \geq 0$.
- **Salida**: un *hash value* de 128 bits de "*x*".
- **Tamaño del bloque**: 512 bits.
- **Número de rondas**: 3

2.1.1 Definición de constantes.

Se definen el vector de inicialización (las 4 palabras iniciales de 32 bits) ***IV*** denominados buffer MD.

$h_1 = 0x67452301, h_2 = 0xefcdab89, h_3 = 0x98badcfe, h_4 = 0x10325476$.

Las constantes aditivas son:

$y[j] = 0$, $0 \leq j \leq 15$;
$y[j] = 0x5a827999$, $16 \leq j \leq 31$; (raíz cuadrada de 2)
$y[j] = 0x6ed9eba1$, $32 \leq j \leq 47$; (raíz cuadrada de 3)

El orden para acceder a las palabras fuente definido (cada lista va de 0 a 15) es:

z[0..15] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
z[16..31] = [0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15]
z[32..47] = [0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15]

Finalmente se define el número de corrimientos a la izquierda en bits (rotación):

s[0..15] = [3, 7, 11, 19, 3, 7, 11, 19, 3, 7, 11, 19, 3, 7, 11, 19]
s[16..31] = [3, 5, 9, 13, 3, 5, 9, 13, 3, 5, 9, 13, 3, 5, 9, 13]
s[32..47] = [3, 9, 11, 15, 3, 9, 11, 15, 3, 9, 11, 15, 3, 9, 11, 15]

[1] La página en la que se pueden consultar los RFC (Request For Comments) es http://www.ietf.org/rfc.html en la que solo es necesario indicar el número con el que es catalógado.

Las funciones de cada ronda se definen como:

$$F(X,Y,Z) = XY \vee (\neg X)Z$$
$$G(X,Y,Z) = XY \vee XZ \vee YZ$$
$$H(X,Y,Z) = X \oplus Y \oplus Z$$

2.1.2 Preprocesamiento.

Expansión del mensaje.

Se expande o rellena la entrada "***x***" hasta que su longitud (en bits) sea congruente a *448 mod 512*, es decir, 64 bits más del siguiente múltiplo de 512 bits, ya que esta expansión se realiza siempre, incluso cuando la longitud de la entrada ya es congruente a *448 mod 512*.

La forma de realizar esta expansión es la siguiente:

- Se concatena un **"1-bit"**[2] al final del mensaje ***M***.
- Se le agrega al resultado anterior ***r*** "**0-bit**" ($0 \leq r \leq 511$) de tal forma que la longitud de la cadena resultante sea congruente a *448 mod 512* bits.
- Se concatena al final una representación de 64 bits de la longitud inicial del mensaje (*Mlen*) antes de la fase de expansión, es decir, $b \bmod 2^{64}$, siendo ***b*** la longitud inicial del mensaje.

El mensaje resultante (salida) del preprocesamiento tiene una longitud exacta a un múltiplo de 512 bits, por lo que se forman *16*n palabras* (conjunto de 32 bits). Que pueden ser vistas como un arreglo *M[0 .. N-1]* donde ***N*** es un múltiplo de 16.

Se inicializa el buffer MD (conjunto de 4 palabras utilizadas para realizar el cálculo del *hash value*) con los valores previamente mencionados.

$$(A, B, C, D) \leftarrow (h_1, h_2, h_3, h_4).$$

2.1.3 Procesamiento

La salida *M[0..N-1]* resultante del preprocesamiento, contiene ***N*** bloques de 16 palabras de 32 bits.

[2] Durante el desarrollo se entenderá a: **0-bit** como un bit con valor 0 y a **1-bit** como un bit con valor 1.

El pseudo código definido [RFC-1320] es como sigue:

```
PARA i = 0 HASTA N/16-1 HAZ
INICIA
    /* Copiar bloque en almacenamiento temporal X. */
    PARA j = 0 HASTA 15 HAZ
            X[j] ← M[i*16+j]
    FIN PARA
    /* AImacenar. */
    A ← H1
    B ← H2
    C ← H3
    D ← H4
    /* Ronda 1. */
    /* [abcd k s] indica la operación        a = (a + F(b, c, d) + X[k]) <<< s.        Ejecutar  */
    [ABCD 0 3]      [DABC 1 7]      [CDAB 2 11]     [BCDA 3 19]
    [ABCD 4 3]      [DABC 5 7]      [CDAB 6 11]     [BCDA 7 19]
    [ABCD 8 3]      [DABC 9 7]      [CDAB 10 11]    [BCDA 11 19]
    [ABCD 12 3]     [DABC 13 7]     [CDAB 14 11]    [BCDA 15 19]
    /* Ronda 2. */
    /* [abcd k s] indica la operación     a = (a + G(b, c, d) + X[k] + 5A827999) <<< s.  Ejecutar */
    [ABCD 0 3]      [DABC 4 5]      [CDAB 8 9]      [BCDA 12 13]
    [ABCD 1 3]      [DABC 5 5]      [CDAB 9 9]      [BCDA 13 13]
    [ABCD 2 3]      [DABC 6 5]      [CDAB 10 9]     [BCDA 14 13]
    [ABCD 3 3]      [DABC 7 5]      [CDAB 11 9]     [BCDA 15 13]
    /* Ronda 3. */
    /* [abcd k s] indica la operación     a = (a + H(b, c, d) + X[k] + 6ED9EBA1) <<< s.  Ejecutar */
    [ABCD 0 3]      [DABC 8 9]      [CDAB 4 11]     [BCDA 12 15]
    [ABCD 2 3]      [DABC 10 9]     [CDAB 6 11]     [BCDA 14 15]
    [ABCD 1 3]      [DABC 9 9]      [CDAB 5 11]     [BCDA 13 15]
    [ABCD 3 3]      [DABC 11 9]     [CDAB 7 11]     [BCDA 15 15]
    /* Se actualizan los valores del buffer MD con los valores con los que fueron iniciado */
    H1 = A + H1
    H2 = B + H2
    H3 = C + H3
    H4 = D + H4
TERMINA PARA
```

De forma algebraica puede definirse de la siguiente manera:

```
PARA cada M[i] desde 0 hasta N-1,
INICIA
    /* Se copia el i^th bloque en un almacenamiento temporal */
    X[j] ← M16i+j, 0 ≤ j ≤ 15
    /* Se almacenan los valores iniciales del buffer MD */
    (A, B, C, D) ← (H1, H2, H3, H4)

    /* Se procesa el bloque con las funciones de ronda especificas, constante aditiva y de
corrimiento */
    /* Ronda 1  */
    PARA j = 0 HASTA 15 HAZ
    INICIA
        /* Se ejecuta */
        t ← ( A + F(B, C, D) + X[z[j]] + y[j] ),
        /* Se actualiza el orden y se efectúa el corrimiento correspondiente */
        (A, B, C, D) ← (D, t ← s[j], B, C)
    TERMINA PARA
    /* Ronda 2 */
    PARA j = 16 HASTA 31 HAZ
    INICIA
        t ← ( A + G(B, C, D) + X[z[j]] + y[j] ),
        /* Se actualiza el orden y se efectúa el corrimiento correspondiente */
        (A, B, C, D) ← (D, t ← s[j], B, C)
    TERMINA PARA
    /* Ronda 3 */
    PARA j = 32 HASTA 47 HAZ
    INICIA
        t ← ( A + H(B, C, D) + X[z[j]] + y[j] ),
        /* Se actualiza el orden y se efectúa el corrimiento correspondiente */
        (A, B, C, D) ← (D, t ← s[j], B, C)
    TERMINA PARA

    /* Al terminar de ejecutar las rondas se actualizan las variables de encadenamiento */
    (H1, H2, H3, H4) ←(A + H1, B + H2, C + H3, D + H4).
TERMINA PARA
```

2.1.4 Salida Hash Value.

El hash value es la concatenación: *H4* || *H3* || *H2* || *H1*. Esto es empieza con el byte de menor orden de H1 y termina con el byte de alto orden de H4.

Para este algoritmo se han encontrado colisiones para MD4 en 220 cálculos en la función de compresión. Por esta razón, MD4 no es recomendado como una función hash resistente a colisiones. La mejora en el diseño de este algoritmo es la función MD5, la cual también forma parte de la llamada familia MD-x que a continuación describiremos.

2.2 MD5

MD5 [RFC-1321] es una versión robusta de *MD4* los principales cambios con respecto a este son:

- Se agrega una cuarta ronda de 16 iteraciones con su función respectiva.
- Se remplaza la función de la segunda ronda para volverla menos simétrica.
- Se modifica el orden de acceso en las palabras del mensaje en rondas 2 y 3.
- Se modifica la cantidad de corrimientos (los cuales difieren en las diferentes rondas).
- Uso de constantes aditivas únicas en cada una de las *4x16* pasos, basado en la parte entera de $2^{32}sin(j)$ para cada *j* (requiere sobre todo 256 bytes de almacenamiento).
- Suma de la salida previa en cada uno de los 64 pasos.
- Ahora en cada paso se agrega el resultado del paso anterior. Lo cual provoca un efecto avalancha.

Las principales características son:

- **Entrada**: Una cadena de bits *x* de longitud arbitraria $b \geq 0$.
- **Salida**: Un valor hash de *128* bits de *x*.
- **Tamaño de bloque**: 512 bits.
- **Número de rondas:** 4.

2.2.1 Definición de Constantes.

Se definen el vector de inicialización ***IV*** (las 4 palabras iniciales de 32 bits) que también se denotan o llaman buffer MD.

$h_1 = 0x67452301$, $h_2 = 0xefcdab89$, $h_3 = 0x98badcfe$, $h_4 = 0x10325476$.

Las constantes aditivas están definidas de acuerdo con la ecuación:
y[j] = Parte entera del resultado de $4294967296\ (abs(sin(j+1)))$, $0 \leq j \leq 63$, donde *j* esta en radianes.

$$y[j] = 4294967296\ (abs(sin(j+1)))$$

Orden definido para acceder a las palabras fuente (cada lista va de 0 a 15):

z[0..15] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
z[16..31] = [1, 6, 11, 0, 5, 10, 15, 4, 9, 14, 3, 8, 13, 2, 7, 12]
z[32..47] = [5, 8, 11, 14, 1, 4, 7, 10, 13, 0, 3, 6, 9, 12, 15, 2]
z[48..63] = [0, 7, 14, 5, 12, 3, 10, 1, 8, 15, 6, 13, 4, 11, 2, 9]

Nótese que se redefinen las rondas 2 y 3 con respecto de MD4.

Finalmente se define el número de cambios a la izquierda en posiciones de bits (rotación):

s[0..15] = [7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22]
s[16..31] = [5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20]
s[32..47] = [4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23]
s[48..63] = [6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21]

Las funciones de ronda son definidas de la siguiente manera:

$$F(X,Y,Z) = XY \vee (\neg X)Z$$
$$G(X,Y,Z) = XZ \vee Y(\neg Z)$$
$$H(X,Y,Z) = X \oplus Y \oplus Z$$
$$I(X,Y,Z) = Y \oplus (X \vee (\neg Z))$$

Nótese que se redefine la función de la segunda ronda (G) y se define la de la cuarta, propia de esta definición de algoritmo.

2.2.2 Preprocesamiento

Expansión del mensaje.

Se sigue el mismo procedimiento que en el caso anterior. El mensaje resultante (salida) del preprocesamiento tiene una longitud igual a un múltiplo de 512 bits, con lo que se pueden forman *16*n palabras*. Que pueden ser vistas como un arreglo *M[0..N-1]* donde ***N*** es un múltiplo de 16.

2.2.3 Procesamiento

La salida *M[0..N-1]* resultante del preprocesamiento, contiene ***N*** bloques de 16 palabras de 32 bits.

El pseudo código definido [RFC-1321] es como sigue:

```
PARA i = 0 HASTA N/16-1 HAZ
INICIA
    /* Copiar bloque en almacenamiento temporal X. */
    PARA j = 0 HASTA 15 HAZ
            X[j] ← M[i*16+j]
    FIN PARA
    /* Almacenar. */
    A ← H1
    B ← H2
    C ← H3
    D ← H4
    /* Ronda 1. */
    /* [abcd k s i] indica la operación:   a = b + ((a + F(b, c, d) + X[k] + T[i]) <<< s).  Ejecutar */
    [ABCD  0 7  1]      [DABC  1 12  2]      [CDAB  2 17  3]      [BCDA  3 22  4]
    [ABCD  4 7  5]      [DABC  5 12  6]      [CDAB  6 17  7]      [BCDA  7 22  8]
    [ABCD  8 7  9]      [DABC  9 12 10]      [CDAB 10 17 11]      [BCDA 11 22 12]
    [ABCD 12 7 13]      [DABC 13 12 14]      [CDAB 14 17 15]      [BCDA 15 22 16]
    /* Ronda 2. */
    /* [abcd k s i] indica la operación:   a = b + (a + G(b, c, d) + X[k] + T[i]) <<< s).  Ejecutar */
    [ABCD  1 5 17]      [DABC  6 9 18]       [CDAB 11 14 19]      [BCDA  0 20 20]
    [ABCD  5 5 21]      [DABC 10 9 22]       [CDAB 15 14 23]      [BCDA  4 20 24]
```

[ABCD 9 5 25] [DABC 14 9 26] [CDAB 3 14 27] [BCDA 8 20 28]
[ABCD 13 5 29] [DABC 2 9 30] [CDAB 7 14 31] [BCDA 12 20 32]
/* Ronda 3. */
/* [abcd k s] indica la operación: a = b + ((a + H(b, c, d) + X[k] + T[i]) <<< s). Ejecutar */
[ABCD 5 4 33] [DABC 8 11 34] [CDAB 11 16 35] [BCDA 14 23 36]
[ABCD 1 4 37] [DABC 4 11 38] [CDAB 7 16 39] [BCDA 10 23 40]
[ABCD 13 4 41] [DABC 0 11 42] [CDAB 3 16 43] [BCDA 6 23 44]
[ABCD 9 4 45] [DABC 12 11 46] [CDAB 15 16 47] [BCDA 2 23 48]
/* Ronda 4. */
/* [abcd k s] indica la operación: a = b + ((a + I(b, c, d) + X[k] + T[i]) <<< s). Ejecutar */
[ABCD 0 6 49] [DABC 7 10 50] [CDAB 14 15 51] [BCDA 5 21 52]
[ABCD 12 6 53] [DABC 3 10 54] [CDAB 10 15 55] [BCDA 1 21 56]
[ABCD 8 6 57] [DABC 15 10 58] [CDAB 6 15 59] [BCDA 13 21 60]
[ABCD 4 6 61] [DABC 11 10 62] [CDAB 2 15 63] [BCDA 9 21 64]
/* Se actualizan los valores del buffer MD con los valores con los que fueron iniciado */
H1 = A + H1
H2 = B + H2
H3 = C + H3
H4 = D + H4
TERMINA PARA

De forma algebraica se define de la siguiente manera:

PARA cada M[i] desde 0 hasta N-1,
INICIA
/* Se copia el i^{th} bloque en un almacenamiento temporal */
$X[j] \leftarrow M16i+j,\ 0 \leq j \leq 15$
/* Se almacenan los valores iniciales del buffer MD */
$(A, B, C, D) \leftarrow (H1, H2, H3, H4)$
/* Se procesa el bloque con las funciones de ronda, constante aditiva y de corrimiento */
/* Ronda 1 */
PARA j = 0 HASTA 15 HAZ
INICIA
/* Se ejecuta */
$t \leftarrow (A + F(B, C, D) + X[z[j]] + y[j])$,
/* Se actualiza el orden y se efectúa el corrimiento correspondiente */
$(A, B, C, D) \leftarrow (D, B + (t \leftarrow s[j]), B, C)$
TERMINA PARA

```
/* Ronda 2 */
PARA j = 16 HASTA 31 HAZ
INICIA
    t ← ( A + G(B, C, D) + X[z[j]] + y[j] ),
    /* Se actualiza el orden y se efectúa el corrimiento correspondiente */
    (A, B, C, D) ← (D, B + (t ← s[j]), B, C)
TERMINA PARA
/* Ronda 3 */
PARA j = 32 HASTA 47 HAZ
INICIA
    t ← ( A + H(B, C, D) + X[z[j]] + y[j] ),
    /* Se actualiza el orden y se efectúa el corrimiento correspondiente */
    (A, B, C, D) ← (D, B + (t ← s[j]), B, C)
TERMINA PARA
/* Ronda 4 */
PARA j = 48 HASTA 63 HAZ
INICIA
    t ← ( A + I(B, C, D) + X[z[j]] + y[j] ),
    /* Se actualiza el orden y se efectúa el corrimiento correspondiente */
    (A, B, C, D) ← (D, B + (t ← s[j]), B, C)
TERMINA PARA
/* Al terminar de ejecutar las rondas se actualizan las variables de encadenamiento */
(H1, H2, H3, H4) ←(A + H1, B + H2, C + H3, D + H4).
TERMINA PARA
```

2.2.4 Finalización.

El hash value es la concatenación: H4 || H3 || H2 || H1. Esto es empieza con el byte de menor orden de H1 y termina con el byte de alto orden de H4.

Con MD4 y MD5 terminamos de describir a las llamadas funciones hash de la familia MD-x, empero, a continuación describiremos a la segunda familia importante de funciones hash la llamada familia SHA-x cuya primer función representativa es SHA-1 la cual es una variante de SHA-0 como veremos.

2.3 SHA-1

En 1993 el NIST publicó el FIPS-180, el cual describía una función hash segura llamada SHA (posteriormente conocido como SHA-0). En 1995 con el FIPS-180-1 SHA fue ligeramente modificado (una rotación a la izquierda de un bit fue agregada) y fue llamado SHA-1. En ambas versiones el tamaño del *hash value* es de 160 bits. En FIPS-180-2 se definen además las funciones SHA-256, SHA-384 y SHA-512. A continuación se describe SHA-1 para obtener el *hash value*.

Una característica importante de esta función hash es que la longitud máxima del mensaje debe ser menor a 2^{64} bits.

En resumen se establecen como características principales de SHA-0 y SHA-1 las siguientes:

- **Entrada**: Una cadena de bits *x* de longitud entre $0 \leq b < 2^{64}$.
- **Salida**: Un valor hash de *160* bits de *x*.
- **Tamaño de bloque**: 512 bits.
- **Número de rondas:** 4.

2.3.1 Definición de constantes.

Las constantes aditivas utilizadas en *SHA-1* son las siguientes:

$K[t] = 5A827999 \quad 0 \leq j \leq 19$

$K[t] = 6ED9EBA1 \quad 20 \leq j \leq 39$

$K[t] = 8F1BBCDC \quad 40 \leq j \leq 59$

$K[t] = CA62C1D6 \quad 60 \leq j \leq 79$

La funciones de ronda definidas operan sobre 3 palabras de 32 bits (*X, Y, Z*) y proporcionan como salida una palabra de 32 bits. Las definiciones son:

$F(t, X, Y, Z) = XY \vee (\neg X)Z \quad (0 \leq j \leq 19)$

$G(t, X, Y, Z) = X \oplus Y \oplus Z \quad (20 \leq j \leq 39)$

$H(t, X, Y, Z) = XY \vee XZ \vee YZ \quad (40 \leq j \leq 59)$

$I(t, X, Y, Z) = X \oplus Y \oplus Z \quad (60 \leq j \leq 79)$

Se define la función "$S_n(x)$" [7] (pp. 8) de corrimiento circular a la izquierda, donde "***x***" es una palabra de "***w***" bits y "***n***" es un entero $0 \leq n < w$, (para este caso ***w = 32***) como:

$$S_n(x) = (X << n) \vee (X >> w - n)$$

Esta operación es utilizada para definir a las funciones SHA-256, SHA-384 y SHA-512. Es equivalente a efectuar un corrimiento de ***x*** bits ***n*** posiciones a la izquierda.

Nótese que:

- La definición en cuanto a funcionalidad de la función ***F*** corresponde con la de *MD5* en los términos generales.
- La función ***G*** corresponde con la función ***H*** de *MD5*.

2.3.2 Preprocesamiento

Expansión del mensaje.

Se sigue el mismo procedimiento utilizado en el diseño de las funciones de la familia *MD-X*. El mensaje resultante (salida) del preprocesamiento tiene una longitud igual a un múltiplo de 512 bits (*n*512* bits), con lo que se pueden forman *16*n palabras*.

Que pueden ser vistas como un arreglo *M[0 .. N-1]* donde ***N*** es un múltiplo de 16.

2.3.3 Procesamiento.

Existen dos métodos para calcular el valor hash. Aquí se describirán ambos [7] (pp. 15).

Método 1.

En este método se utilizan dos buffer de 5 palabras de 32 bits y una secuencia o arreglo de 80 palabras de 32 bits. Los elementos del primer buffer se denotarán como: *A, B, C, D, E*; los del segundo como: H_0, H_1, H_2, H_3, H_4.

La secuencia de variables de ronda formada por las 80 palabras se denotarán como *W[0..79]*. Adicionalmente se tendrá un buffer auxiliar *TEMP* cuya longitud es de una palabra de 32 bits.

Después de realizar el padding en el preprocesamiento se obtienen en total ***n*** bloques de 16 palabras, cada bloque se denomina como: *M(1), M(2), ..., M(n)* para procesarlos en dicho orden.

El procesamiento de cada bloque ***M(i)*** implica 80 pasos (igual al número de constantes). Por lo que se define al vector de inicialización ***IV***; denotado aquí por el segundo buffer H_i de la siguiente manera:

$H_0 = 67452301$
$H_1 = EFCDAB89$
$H_2 = 98BADCFE$
$H_3 = 10325476$
$H_4 = C3D2E1F0$

Procesamiento

Para procesar cada ***M(i)*** (conjunto *i*-ésimo de 16 palabras) es necesario realizar los siguientes pasos:

```
PARA i = 0 HASTA N HAZ
INICIO
    /* Asignamos los primeros 15 elementos del buffer auxiliar
    W[0..79]. */
    PARA j = 0 HASTA 15 HAZ
    INICIA
        W[j] = M[i*16 + j]
    FIN PARA
    /* Completamos el buffer W[16..79] de la siguiente manera. La forma depende del tipo
     * de algoritmo aquí radica la diferencia entre ambos. */
    PARA t = 16 HASTA 79 HAZ
    INICIA
        W[t] = W[t-3] ⊕ W[t-8] ⊕ W[t-14] ⊕ W[t-16]     [SHA-0]
        W[t] = S1(W[t-3] ⊕ W[t-8] ⊕ W[t-14] ⊕ W[t-16])          [SHA-1]
    FIN PARA
    /* Se asignan al buffer los valores del IV cómo sigue*/
    A = H0
    B = H1
    C = H2
    D = H3
```

```
        E = H4
        /* Se procesan los bloques del arreglo. */
        PARA t = 0 HASTA 79 HAZ
        INICIA
            /* Se calculan los siguientes valores de ronda de la siguiente manera. */
            TEMP = S5(A)+ f(t, B, C, D) + E + W(t) + K(t)
            E = D
            D = C
            C = S30(B)
            B = A
            A = TEMP
        FIN PARA
        /* Se asignan los valores del vector de inicialización para el siguiente M[i]. */
        H0 = H0 + A
        H1 = H1 + B
        H2 = H2 + C
        H3 = H3 + D
        H4 = H4 + E
    FIN PARA
```

Método 2

Se considera y asume que existe una secuencia de 80 palabras de 32 bits *W[0..79]* cuya definición es la misma que la mostrada en el método anteriormente descrito. Si se debe optimizar el espacio, este método da una alternativa al ver una *W[t]* como una cola circular, el cual podría ser implementada como un arreglo de 16 palabras de 32 bits *W[0..15]*.

Se definen la siguiente constante *MASK = 0000 000F*.

Procesamiento.

```
    PARA i = 0 HASTA N HAZ
    INICIA
        /* Se dividen el bloque M[i] en las 16 palabras de 32 bits para establecer W[0..15]*/
        PARA j = 0 HASTA 15 HAZ
        INICIA
            W[j] = M[16*i + j]
        FIN PARA

        A = H0
```

```
B = H1
C = H2
D = H3
E = H4
PARA t = 0 HASTA 79 HAZ
INICIA
    s = t ∧ MASK
    SI (t ≥ 16)
    INICIA
        W[s]=W[(s+13)∧MASK]⊕W[(s+8)∧MASK]⊕W[(s+2)∧MASK]⊕W[s]
        W[s]= W[s]                                   [SHA-0]
        W[s]= S1(W[s])                               [SHA-1]
    FIN SI
    TEMP = S5(A) + f(t, B, C, D) + E + W[s] + K[t]
    E = D
    D = C
    C = S30(B)
    B = A
    A = TEMP
FIN PARA
H0 = A + H0
H1 = B + H1
H2 = C + H2
H3 = D + H3
H4 = E + H4
FIN PARA
```

2.3.4 Finalización

El *hash value* se forma a partir de la concatenación de H_0, H_1, H_2, H_3 y H_4 después de haber procesado la totalidad de los ***N*** mensajes. Las funciones SHA-256, SHA-384 y SHA-512 integran la llamada familia SHA-x, cuyas características principales mostraremos a continuación, entre las que destacaremos de inicio que el número que acompaña al nombre de la función (como sufijo) indica el tamaño en bits del hash value.

2.4 SHA-256, SHA-384 y SHA-512

Las funciones hash SHA-256, SHA-384 y SHA-512 fueron publicadas en Agosto de 2002 en el FIPS 180-2 con lo que el NIST las establece junto con SHA-1, como funciones hash estándar para el gobierno de los Estados Unidos. Cabe resaltar que el proyecto *NESSIE* sólo considera como funciones hash seguras a *SHA-256*, *SHA-384* y *SHA-512*. Al conjunto de estas tres funciones a menudo se les conoce como familia SHA-2.

La longitud del *hash value* de cada una de las funciones está indicada en el sufijo del nombre de la función respectivamente. Al igual que para SHA-1 una característica importante es la longitud máxima del mensaje aceptado esta depende de la definición de la función misma.

A continuación proporcionaremos las características de SHA-256, SHA-384 y SHA-512:

SHA-256

- **Entrada**: Una cadena de bits *x* de longitud entre $0 \leq b < 2^{64}$.
- **Salida**: Un valor hash de *256* bits de *x*.
- **Tamaño de bloque**: 512 bits.
- **Número de rondas:** 64

SHA-384

- **Entrada**: Una cadena de bits *x* de longitud entre $0 \leq b < 2^{128}$.
- **Salida**: Un valor hash de *384* bits de *x*.
- **Tamaño de bloque**: 1024 bits.
- **Número de rondas:** 80

SHA-512

- **Entrada**: Una cadena de bits *x* de longitud entre $0 \leq b < 2^{128}$.
- **Salida**: Un valor hash de *512* bits de *x*.
- **Tamaño de bloque**: 1024 bits.
- **Número de rondas:** 80

2.4.1 Definición de funciones y constantes.

Definimos dos funciones de corrimiento de bits a la derecha de la siguiente manera:

Corrimiento simple: $SHR_n(x) = x >> n$

Corrimiento circular: $ROTR_n(x) = (x >> n) \vee (x << (w - n))$

La relación entre éstas y las de corrimiento a la izquierda es: $S_n(x) \approx ROTL_{w-n}(x)$

Las funciones de ronda operan sobre 3 palabras de 32 bits (*X*, *Y*, *Z*) y proporcionan como salida una palabra de 32 bits. Estás son definidas como:

$$F(t, X, Y, Z) = XY \vee (\neg X)Z$$
$$G(t, X, Y, Z) = X \oplus Y \oplus Z$$
$$H(t, X, Y, Z) = XY \vee XZ \vee YZ$$
$$I(t, X, Y, Z) = X \oplus Y \oplus Z$$
$$J(t, X, Y, Z) = XY \oplus (\neg X)Z$$
$$K(t, X, Y, Z) = XY \oplus XZ \oplus YZ$$

Nótese que las primeras 4 funciones corresponden a las de SHA-1. Las constantes de ronda a diferencia de las utilizadas en SHA-1 son diferentes entre sí, las cuales se definirán a continuación en hexadecimal para cada uno de los algoritmos:

2.4.1.1 SHA-256

Utiliza un arreglo de 64 palabras de 32 bits denotadas como *K[0..63]*, cuyos valores son los primeros 32 bits de la parte fraccional de la raíz cúbica de los primeros 64 números primos, cuyos valores son:

428a2f98	71374491	b5c0fbcf	e9b5dba5	3956c25b	59f111f1	923f82a4	ab1c5ed5
d807aa98	12835b01	243185be	550c7dc3	72be5d74	80deb1fe	9bdc06a7	c19bf174
e49b69c1	efbe4786	0fc19dc6	240ca1cc	2de92c6f	4a7484aa	5cb0a9dc	76f988da
983e5152	a831c66d	b00327c8	bf597fc7	c6e00bf3	d5a79147	06ca6351	14292967
27b70a85	2e1b2138	4d2c6dfc	53380d13	650a7354	766a0abb	81c2c92e	92722c85
a2bfe8a1	a81a664b	c24b8b70	c76c51a3	d192e819	d6990624	f40e3585	106aa070
19a4c116	1e376c08	2748774c	34b0bcb5	391c0cb3	4ed8aa4a	5b9cca4f	682e6ff3
748f82ee	78a5636f	84c87814	8cc70208	90befffa	a4506ceb	bef9a3f7	c67178f2

Tabla 2. Constantes definidas para SHA-256.

Se definen funciones de corrimiento no lineales de acuerdo con la versión del algoritmo, para SHA-256 es como sigue:

Se define w = 256 bits, así las funciones son:

$$\sum_0(x) = ROTR_2(x) \oplus ROTR_{13}(x) \oplus ROTR_{22}(x)$$

$$\sum_1(x) = ROTR_6(x) \oplus ROTR_{11}(x) \oplus ROTR_{25}(x)$$

$$\sigma_0(x) = ROTR_7(x) \oplus ROTR_{18}(x) \oplus SHR_3(x)$$

$$\sigma_1(x) = ROTR_{17}(x) \oplus ROTR_{19}(x) \oplus SHR_{10}(x)$$

2.4.1.2 SHA-384 y SHA 512

Las funciones SHA-384 y SHA-512 utilizan un arreglo de 80 palabras de 64 bits denotadas como *K[0..79]*, cuyos valores son los primeros 64 bits de la parte fraccional de la raíz cúbica de los primeros 80 números primos, cuyos valores son:

428a2f98d728ae22	7137449123ef65cd	b5c0fbcfec4d3b2f	e9b5dba58189dbbc
3956c25bf348b538	59f111f1b605d019	923f82a4af194f9b	ab1c5ed5da6d8118
d807aa98a3030242	12835b0145706fbe	243185be4ee4b28c	550c7dc3d5ffb4e2
72be5d74f27b896f	80deb1fe3b1696b1	9bdc06a725c71235	c19bf174cf692694
e49b69c19ef14ad2	efbe4786384f25e3	0fc19dc68b8cd5b5	240ca1cc77ac9c65
2de92c6f592b0275	4a7484aa6ea6e483	5cb0a9dcbd41fbd4	76f988da831153b5
983e5152ee66dfab	a831c66d2db43210	b00327c898fb213f	bf597fc7beef0ee4
c6e00bf33da88fc2	d5a79147930aa725	06ca6351e003826f	142929670a0e6e70
27b70a8546d22ffc	2e1b21385c26c926	4d2c6dfc5ac42aed	53380d139d95b3df
650a73548baf63de	766a0abb3c77b2a8	81c2c92e47edaee6	92722c851482353b
a2bfe8a14cf10364	a81a664bbc423001	c24b8b70d0f89791	c76c51a30654be30
d192e819d6ef5218	d69906245565a910	f40e35855771202a	106aa07032bbd1b8
19a4c116b8d2d0c8	1e376c085141ab53	2748774cdf8eeb99	34b0bcb5e19b48a8
391c0cb3c5c95a63	4ed8aa4ae3418acb	5b9cca4f7763e373	682e6ff3d6b2b8a3
748f82ee5defb2fc	78a5636f43172f60	84c87814a1f0ab72	8cc702081a6439ec
90befffa23631e28	a4506cebde82bde9	bef9a3f7b2c67915	c67178f2e372532b
ca273eceea26619c	d186b8c721c0c207	eada7dd6cde0eb1e	f57d4f7fee6ed178
06f067aa72176fba	0a637dc5a2c898a6	113f9804bef90dae	1b710b35131c471b
28db77f523047d84	32caab7b40c72493	3c9ebe0a15c9bebc	431d67c49c100d4c
4cc5d4becb3e42b6	597f299cfc657e2a	5fcb6fab3ad6faec	6c44198c4a475817

Tabla 3. Constantes de SHA-384 y SHA-512.

Las funciones de corrimiento no lineales definidas para SHA-384 y SHA-512 son:

Se define w = 512 bits, así las funciones son:

$$\sum_0(x) = ROTR_{28}(x) \oplus ROTR_{34}(x) \oplus ROTR_{39}(x)$$

$$\sum_1(x) = ROTR_{14}(x) \oplus ROTR_{18}(x) \oplus ROTR_{41}(x)$$

$$\sigma_0(x) = ROTR_1(x) \oplus ROTR_8(x) \oplus SHR_7(x)$$

$$\sigma_1(x) = ROTR_{19}(x) \oplus ROTR_{61}(x) \oplus SHR_6(x)$$

2.4.2 Preprocesamiento

Expansión del mensaje.
Las funciones SHA-256, SHA-384 y SHA-512 tienen básicamente el mismo diseño, cada una agrega bits al final del mensaje hasta que su longitud sea un múltiplo del tamaño del bloque de la función hash 512 para SHA-256, 1024 para SHA-384 y SHA-512. Por lo que lo definiremos a ***L*** como la *longitud del mensaje*.

2.4.2.1 SHA-256

Expansión del mensaje.
El procedimiento utilizado por SHA-256 para expandir el mensaje es el mismo de SHA-1, por lo que la salida tiene una longitud múltiplo de 512 bits (*n*512* bits), a partir de la cual se forman *16*n palabras,* vistas como un arreglo *M[0 .. N-1]* donde ***N*** es un múltiplo de 16.

2.4.2.2 SHA-384 y SHA-512

Se expande la entrada "***x***" hasta que su longitud (en bits) sea congruente a *896 mod 1024*, de la siguiente forma: se concatena un **"1-bit"** al mensaje y posteriormente ***r-1*** *(≥ 0)* **"0-bit"** hasta que la longitud del mensaje sea congruente a *896 mod 1024* bits. El número de bits a ser agregados varía entre uno y *896* bits como máximo.

Concatenación de la longitud original del mensaje.
Después de realizar la expansión del mensaje se concatena la representación en bits de la longitud inicial del mensaje antes de la expansión.

Para definir los elementos que conforman el arreglo *M[0..15]* utilizado en SHA-256 se sigue el mismo procedimiento que el descrito para SHA-1.

Para SHA-384 y SHA-512 a partir del mensaje resultante (salida) del preprocesamiento cuya longitud es congruente a *1024* bits, se forman *"**N**"* bloques de *1024* bits, las cuales serán representadas como elementos de un arreglo *M[0 .. N-1]*, ya que un bloque de *1024* bits puede ser expresado como un conjunto de 16 palabras de 64 bits, se establece que

para el bloque *"**i**"* de 1024 bits, los primeros 64 bits pueden ser representados de la siguiente manera $M_0(i)$, los siguientes 64 bits como $M_1(i)$ y así sucesivamente hasta $M_{15}(i)$.

Vector de inicialización.

El vector de inicialización se define a partir del tipo de algoritmo que se trate:

SHA-256 define su vector de inicialización de la siguiente manera:

H_0 = 6a09e667

H_1 = *bb67ae85*

H_2 = *3c6ef372*

H_3 = *a54ff53a*

H_4 = *510e527f*

H_5 = *9b05688c*

H_6 = *1f83d9ab*

H_7 = *5be0cd19*

Los cuales son obtenidos tomando los primeros 32 bits de la parte fraccionaria de la raíz cuadrada de los primeros 8 números primos.

SHA-384 define su vector de inicialización a partir de que cada elemento es la representación:

H_0 = cbbb9d5dc1059ed8

H_1 = 629a292a367cd507

H_2 = 9159015a3070dd17

H_3 = 152fecd8f70e5939

H_4 = 67332667ffc00b31

H_5 = 8eb44a8768581511

H_6 = db0c2e0d64f98fa7

H_7 = 47b5481dbefa4fa4

Los cuales son obtenidos tomando los primeros 64 bits de la parte fraccionaria de la raíz cuadrada del noveno al décimo sexto número primo.

SHA-512 define su vector de inicialización a partir de que cada elemento es la representación.

H_0 = 6a09e667f3bcc908
H_1 = bb67ae8584caa73b
H_2 = 3c6ef372fe94f82b
H_3 = a54ff53a5f1d36f1
H_4 = 510e527fade682d1
H_5 = 9b05688c2b3e6c1f
H_6 = 1f83d9abfb41bd6b
H_7 = 5be0cd19137e2179

Los cuales son obtenidos tomando los primeros 64 bits de la parte fraccionaria de la raíz cuadrada de los primeros 8 números primos.

2.4.3 Procesamiento

2.4.3.1 SHA-256

El mensaje es dividido en ***N*** bloques de 512 bits los cuales serán utilizados como entradas para la función.

```
PARA i = 0 HASTA N HAZ
INICIA
    /* Se dividen el bloque M[i] en las 16 palabras de 32 bits para establecer los valores de
    W[0..15] para los valores comprendidos entre W[16..79] se calculan como se indica. */
    PARA j = 0 HASTA 63 HAZ
    INICIA
        SI j <= 15 HAZ
            W[j] = M[16*i + j]
        SINO
            W[j] = σ1(W[j-2]) + W[j-7] + σ0(W[j-15]) + W[j-16]
    FIN PARA

    /* Inicializamos el vector */
    A = H0
    B = H1
    C = H2
```

D = H_3

E = H_4

F = H_5

G = H_6

H = H_7

/* Se ejecutan las 64 rondas. */

PARA t = 0 HASTA 63 HAZ

INICIA

T_1 = H + $\sum_1$(E) + J(E, F, G) + K_t + W[t]

T_2 = $\sum_0$(A) + K(A, B, C)

H = G

G = F

F = E

E = D + T_1

D = C

C = B

B = A

A = T_1 + T_2

TERMINA PARA

H_0 = A + H_0

H_1 = B + H_1

H_2 = C + H_2

H_3 = D + H_3

H_4 = E + H_4

H_5 = F + H_5

H_6 = G + H_6

H_7 = H + H_7

TERMINA

2.4.3.2 SHA-512

El mensaje es dividido en ***N*** bloques de 1024 bits, los cuales serán utilizados como entradas para la función.

PARA i = 0 HASTA N HAZ

INICIA

/* Se dividen el bloque M[i] en las 16 palabras de 64 bits para establecer los valores de W[0..15] para los valores comprendidos entre W[16..79] se calculan como se indica. */

PARA j = 0 HASTA 63 HAZ
INICIA
    SI j <= 15 HAZ
        W[j] = M[16*i + j]
    SINO
        W[j] = σ_1(W[j-2]) + W[j-7] + σ_0(W[j-15]) + W[j-16]
FIN PARA
/* Inicializamos el vector */
A = H_0
B = H_1
C = H_2
D = H_3
E = H_4
F = H_5
G = H_6
H = H_7
/* Se ejecutan las 80 rondas. */
PARA t = 0 HASTA 79 HAZ
INICIA
    T_1 = H + $\sum_1$(E) + J(E, F, G) + K_t + W[t]
    T_2 = $\sum_0$(A) + K(A, B, C)
    H = G
    G = F
    F = E
    E = D + T_1
    D = C
    C = B
    B = A
    A = T_1 + T_2
TERMINA PARA
H_0 = A + H_0
H_1 = B + H_1
H_2 = C + H_2
H_3 = D + H_3
H_4 = E + H_4
H_5 = F + H_5
H_6 = G + H_6

$H_7 = H + H_7$

TERMINA

2.4.3.3 SHA-384

La forma de obtener este es seguir los mismos pasos que el de SHA-512, con la única diferencia de inicializar el ***IV*** con las constantes específicas para SHA-384.

2.4.4 Finalización

2.4.4.1 SHA-256

El *hash value* se forma a partir de la concatenación de H_0, H_1, H_2, H_3, H_4, H_5, H_6 y H_7 después de haber procesado la totalidad de los ***N*** mensajes.

2.4.4.2 SHA-384

El *hash value* se forma a partir de la concatenación de H_0, H_1, H_2, H_3, H_4 y H_5 después de haber procesado la totalidad de los ***N*** mensajes, es decir, se trunca a los primeros 384 bits del *hash value* final.

2.4.4.3 SHA-512

El *hash value* se forma a partir de la concatenación de H_0, H_1, H_2, H_3, H_4, H_5, H_6 y H_7 después de haber procesado la totalidad de los ***N*** mensajes.

Como hemos visto tanto las funciones de la familia *SHA-x* y *MD-x* se basan en operaciones no lineales y corrimientos a nivel de bits, al contrario de la función Whirlpool -cuya estructura es semejante a la de un algoritmo de cifrado específicamente Rijndael- la base de diseño son operaciones en el campo finito GF(2^8). Nosotros partiremos del diseño de Whirlpool para definir a nuestra función Lúthien – Tinúviel Chaos Hash.

2.5 Whirlpool

Es una función hash diseñada por Paulo Barreto y Vincent Rijmen a partir del diseño de una función de cifrado; esta es propuesta al proyecto NESSIE como una función resistente a colisiones.

El algoritmo está orientado a bytes y opera sobre bloques de 512 bits, generando un *hash value* del mismo tamaño.

Características principales:

- **Longitud del mensaje de entrada**: menor a 2^{256} bits.
- **Longitud del *hash value***: 512 bits
- **Tamaño del bloque**: 512 bits.
- **Número de rondas**: 10

Se define un bloque inicial $\boldsymbol{H_0}$ de 512 bits, formado por 0-bit.

Whirlpool [9] puede ser descrito como un algoritmo de cifrado, para obtener el *hash value*, ya que transforma dos bloques H_i (llave) y M_i (bloque de mensaje) ambos de 512 bits, para generar como salida H_{i+1} el cual es sumado a H_i y M_i.

En otras palabras se establece lo siguiente:

$$H_{i+1} = W_{H_i}[M_i] \oplus M_i \oplus H_i$$

La salida H_{i+1} y el bloque M_{i+1} correspondiente serán a su vez las entradas respectivas para la siguiente iteración.

El valor correspondiente al H_t, después de procesar los *t* bloques que conforman el mensaje es al final el *hash value*.

2.5.1 Definición de constantes.

Debido a las características del algoritmo, se definirán en primera instancia las funciones relacionadas (recuérdese que está basado en un algoritmo de cifrado) utilizadas para generar el *hash value*, las llaves (k^r) y constantes (C^r) de ronda, para esto se utilizará la terminología de NESSIE.

Se denominará *state* al bloque del mensaje sobre el cual se efectúan las operaciones; $S = (s_{i,j})$ representará una matriz de *8x8* bytes (512 bits), la cual puede ser interpretada a su vez como un conjunto de elementos de GF(2^8).

A continuación se muestra gráficamente la inicialización del *state*:

$S_{0,0}$	$S_{0,1}$	$S_{0,2}$	$S_{0,3}$	$S_{0,4}$	$S_{0,5}$	$S_{0,6}$	$S_{0,7}$		P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
$S_{1,0}$	$S_{1,1}$	$S_{1,2}$	$S_{1,3}$	$S_{1,4}$	$S_{1,5}$	$S_{1,6}$	$S_{1,7}$		P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}
$S_{2,0}$	$S_{2,1}$	$S_{2,2}$	$S_{2,3}$	$S_{2,4}$	$S_{2,5}$	$S_{2,6}$	$S_{2,7}$		P_{16}	P_{17}	P_{18}	P_{19}	P_{20}	P_{21}	P_{22}	P_{23}
$S_{3,0}$	$S_{3,1}$	$S_{3,2}$	$S_{3,3}$	$S_{3,4}$	$S_{3,5}$	$S_{3,6}$	$S_{3,7}$		P_{24}	P_{25}	P_{26}	P_{27}	P_{28}	P_{29}	P_{30}	P_{31}
$S_{4,0}$	$S_{4,1}$	$S_{4,2}$	$S_{4,3}$	$S_{4,4}$	$S_{4,5}$	$S_{4,6}$	$S_{4,7}$	S=P	P_{32}	P_{33}	P_{34}	P_{35}	P_{36}	P_{37}	P_{38}	P_{39}
$S_{5,0}$	$S_{5,1}$	$S_{5,2}$	$S_{5,3}$	$S_{5,4}$	$S_{5,5}$	$S_{5,6}$	$S_{5,7}$		P_{40}	P_{41}	P_{42}	P_{43}	P_{44}	P_{45}	P_{46}	P_{47}
$S_{6,0}$	$S_{6,1}$	$S_{6,2}$	$S_{6,3}$	$S_{6,4}$	$S_{6,5}$	$S_{6,6}$	$S_{6,7}$		P_{48}	P_{49}	P_{50}	P_{51}	P_{52}	P_{53}	P_{54}	P_{55}
$S_{7,0}$	$S_{7,1}$	$S_{7,2}$	$S_{7,3}$	$S_{7,4}$	$S_{7,5}$	$S_{7,6}$	$S_{7,7}$		P_{56}	P_{57}	P_{58}	P_{59}	P_{60}	P_{61}	P_{62}	P_{63}

Figura 5. Inicialización del state de Whirlpool a partir de un bloque de mensaje.

Al comenzar el proceso, ***state*** es inicializado con el bloque M_i (64 bytes – 512 bits) del mensaje, asignando a $(s_{i,j})$, $0 \le i, j \le 7$, el byte correspondiente a P_i , $0 \le i \le 63$. Donde P_i representa el *i-ésimo* byte que conforma el bloque de mensaje M_i. Dando como consecuencia que cada renglón de la matriz represente una palabra de 64 bits.

2.5.1.1 SubBytes $S_w(S_{i,j})$ – capa no lineal γ

Sustituye los valores de la matriz $S_{i,j}$ (k^r, C^r *o state*) con base en un mapeo fijo no lineal el cual es invertible (*S-box*), cuya definición es:

$S_W(S_{i,j})$	00_x	01_x	02_x	03_x	04_x	05_x	06_x	07_x	08_x	09_x	$0A_x$	$0B_x$	$0C_x$	$0D_x$	$0E_x$	$0F_x$
00_x	18	23	C6	E8	87	B8	01	4F	36	A6	D2	F5	79	6F	91	52
10_x	60	BC	9B	8E	A3	0C	7B	35	1D	E0	D7	C2	2E	4B	FE	57
20_x	15	77	37	E5	9F	F0	4A	DA	58	C9	29	0A	B1	A0	6B	85
30_x	BD	5D	10	F4	CB	3E	05	67	E4	27	41	8B	A7	7D	95	D8
40_x	FB	EE	7C	66	DD	17	47	9E	CA	2D	BF	07	AD	5A	83	33
50_x	63	02	AA	71	C8	19	49	D9	F2	E3	5B	88	9A	26	32	B0
60_x	E9	0F	D5	80	BE	CD	34	48	FF	7A	90	5F	20	68	1A	AE
70_x	B4	54	93	22	64	F1	73	12	40	08	C3	EC	DB	A1	8D	3D
80_x	97	00	CF	2B	76	82	D6	1B	B5	AF	6A	50	45	F3	30	EF
90_x	3F	55	A2	EA	65	BA	2F	C0	DE	1C	FD	4D	92	75	06	8A
$A0_x$	B2	E6	0E	1F	62	D4	A8	96	F9	C5	25	59	84	72	39	4C
$B0_x$	5E	78	38	8C	D1	A5	E2	61	B3	21	9C	1E	43	C7	FC	04
$C0_x$	51	99	6D	0D	FA	DF	7E	24	3B	AB	CE	11	8F	4E	B7	EB
$D0_x$	3C	81	94	F7	B9	13	2C	D3	E7	6E	C4	03	56	44	7F	A9
$E0_x$	2A	BB	C1	53	DC	0B	9D	6C	31	74	F6	46	AC	89	14	E1
$F0_x$	16	3A	69	09	70	B6	D0	ED	CC	42	98	A4	28	5C	F8	86

Tabla 4. Tabla S-box de Whirlpool

Como ejemplo el byte 34_x será sustituido por CB_x durante el procesamiento. De manera más formal se representa como la capa no lineal γ [9] (pp. 5):

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

2.5.1.2 ShiftColumns ($S_{i,j}$) – permutación cíclica π

Opera sobre las columnas de $S_{i,j}(k^r$ o *state*) donde la primer columna es mantenida sin cambios, las restantes son cambiadas en forma cíclica y circular 1, 2, ..., 7 posiciones respectivamente. Este proceso puede ser expresado definiendo una permutación cíclica π [9] (pp. 5) de la siguiente forma:

$$\pi : M_{8x8}[GF(2^8)] \rightarrow M_{8x8}[GF(2^8)]$$
$$\pi(a) = b \Leftrightarrow b_{i,j} = a_{(i-j) \bmod 8, j}, 0 \le i, j \le 7$$

En donde para cada columna ***j*** es permutada hacia abajo ***j*** posiciones. El propósito de π es dispersar los bytes de cada renglón a través de todos los renglones.

Gráficamente corresponde con:

$$\begin{matrix}
S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} & S_{0,4} & S_{0,5} & S_{0,6} & S_{0,7} \\
S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} & S_{1,4} & S_{1,5} & S_{1,6} & S_{1,7} \\
S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} & S_{2,4} & S_{2,5} & S_{2,6} & S_{2,7} \\
S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} & S_{3,4} & S_{3,5} & S_{3,6} & S_{3,7} \\
S_{4,0} & S_{4,1} & S_{4,2} & S_{4,3} & S_{4,4} & S_{4,5} & S_{4,6} & S_{4,7} \\
S_{5,0} & S_{5,1} & S_{5,2} & S_{5,3} & S_{5,4} & S_{5,5} & S_{5,6} & S_{5,7} \\
S_{6,0} & S_{6,1} & S_{6,2} & S_{6,3} & S_{6,4} & S_{6,5} & S_{6,6} & S_{6,7} \\
S_{7,0} & S_{7,1} & S_{7,2} & S_{7,3} & S_{7,4} & S_{7,5} & S_{7,6} & S_{7,7}
\end{matrix}
\quad S \rightarrow S' \quad
\begin{matrix}
S_{0,0} & S_{7,1} & S_{6,2} & S_{5,3} & S_{4,4} & S_{3,5} & S_{2,6} & S_{1,7} \\
S_{1,0} & S_{0,1} & S_{7,2} & S_{6,3} & S_{5,4} & S_{4,5} & S_{3,6} & S_{2,7} \\
S_{2,0} & S_{1,1} & S_{0,2} & S_{7,3} & S_{6,4} & S_{5,5} & S_{4,6} & S_{3,7} \\
S_{3,0} & S_{2,1} & S_{1,2} & S_{0,3} & S_{7,4} & S_{6,5} & S_{5,6} & S_{4,7} \\
S_{4,0} & S_{3,1} & S_{2,2} & S_{1,3} & S_{0,4} & S_{7,5} & S_{6,6} & S_{5,7} \\
S_{5,0} & S_{4,1} & S_{3,2} & S_{2,3} & S_{1,4} & S_{0,5} & S_{7,6} & S_{6,7} \\
S_{6,0} & S_{5,1} & S_{4,2} & S_{3,3} & S_{2,4} & S_{1,5} & S_{0,6} & S_{7,7} \\
S_{7,0} & S_{6,1} & S_{5,2} & S_{4,3} & S_{3,4} & S_{2,5} & S_{1,6} & S_{0,7}
\end{matrix}$$

Figura 6. Permutación definida para Whirlpool.

2.5.1.3 MixRows ($S_{i,j}$) – capa de difusión lineal θ

Realiza una difusión lineal sobre el ***state*** que corresponde a una operación de producto matricial en $GF(2^8)$, donde la matriz ***A*** es definida como:

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

Figura 7. Matriz de difusión lineal de Whirlpool.

Lo que analíticamente sería representado como:

$$S'_{i,0} = S_{i,0} \oplus (S_{i,1} \cdot 09x) \oplus (S_{i,2} \cdot 02x) \oplus (S_{i,3} \cdot 05x) \oplus (S_{i,4} \cdot 08x) \oplus S_{i,5} \oplus (S_{i,6} \cdot 04x) \oplus S_{i,7}$$

Donde el operador $\oplus$ en la expresión que representa la suma en $GF(2^8)$, cuya definición corresponde a la ***XOR*** (or-exclusiva). Las multiplicaciones modulares tienen como base un polinomio irreducible que para el caso de ***W*** es $x^8+x^4+x^3+x^2+1$.

Por ejemplo:

$86_x \cdot 03_x = 1000\ 0110_b \cdot 0000\ 0011_b$

$= (x^7 + x^2 + x) \cdot (x + 1) \quad mod\ x^8 + x^4 + x^3 + x^2 + 1$

$= x^8 + x^7 + x^3 + x \quad mod\ x^8 + x^4 + x^3 + x^2 + 1$

$= x^7 + x^4 + x^2 + x + 1$

$= 97_x$

Esta operación es también llamada capa de difusión θ.

2.5.1.4 AddRoundKey $\sigma[k]$

A cada byte del ***state*** se le aplica la operación ***XOR*** con el byte correspondiente de la llave de ronda k^r de 512 bits.

La definición formal de la operación $\sigma[k]: M_{8x8}[GF(2^8)] \rightarrow M_{8x8}[GF(2^8)]$ que opera con la llave: $k \in M_{8x8}[GF(2^8)]$ y el *state* es [8] (pp. 5):

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

2.5.1.5 Generación de las llaves de ronda k^r.

El objetivo es obtener 11 llaves de ronda de 512 bits (k^r) a partir de una llave de 512 bits ***K***. A partir de un proceso recursivo en el que para generar cada nueva llave de ronda parte de su predecesora.

El pseudo-código del proceso de generación es:

INICIA

K^0 = K

PARA r = 1 HASTA 10 HAZ

INICIA

S = SubBytes (k^{r-1})

S = ShiftColumns (S)

S = MixRows (S)

k^r = *AddRoundKey* (S, C^r)

FIN PARA

FIN

La constante de la ronda *r* es una matriz $C^r \in M_{8x8}[GF(2^8)]$ la cual se define como:

$C^r_{0,j} = S_w(8(r-1)+j)$ $\quad 0 \le j \le 7,$

$C^r_{i,j} = 0$ $\quad 1 \le i \le 7,\ 0 \le j \le 7$

Lo que implica que solo los primeros 64 bits tendrán valores y los demás serán igual a cero.

2.5.2 Preprocesamiento

Expansión del mensaje.

Se agregan bits al mensaje hasta que su longitud sea un múltiplo entero del tamaño del bloque de la función hash. A continuación se muestra el procedimiento:

- Se agrega un 1-bit al final del mensaje *M*.
- Se le agrega al resultado anterior ***r*** 0-bits ($0 \leq r \leq 511$) de tal forma que la longitud de la cadena resultante sea un múltiplo impar de 256. Es decir: $Mlen + 1 + r \equiv 256 \bmod 512$.
- Se concatena al final la representación de *Mlen* de 256 bits con los bits más significativos primero.

Se divide el mensaje resultante en bloques de 512 bits. Con lo que se tendrán ***t*** bloques de 512 bits ($Mlen + 1 + r + 256 = 512t$) $M_0, M_1, ..., M_{t-1}$.

2.5.3 Procesamiento.

El hash value calculado por Whirlpool se obtiene de la siguiente manera:
Se define un valor inicial $\boldsymbol{H_0}$ como una cadena de 512 0-bits H_0 = 0000...00x.

Se aplica la función de cifrado ***W*** con el bloque de mensaje actual $\boldsymbol{M_i}$ como texto en claro y el valor $\boldsymbol{H_i}$ previamente obtenido como llave. Después se calcula la or-exclusiva con las 3 cadenas de 512 bits, el resultado obtenido por ***W***, el bloque del mensaje M_i y H_i.

Para procesar cada ***M(i)*** (bloque $\boldsymbol{i^{th}}$ de 512 bits) es necesario realizar los siguientes pasos:

```
INICIO
    /* Calculamos las constantes de ronda puesto que estas no cambian. */
    PARA r = 0 HASTA 10 HAZ
    INICIA
        C[r][0][j] = SW(8(r - 1) + j),          0 ≤ j ≤ 7
        C[r][i][j]  = 0                  1 ≤ i ≤ 7, 0 ≤ j ≤ 7
    FIN PARA
```

```
/* Calculamos el hash value. */
PARA i = 0 HASTA t-1 HAZ
INICIO
    /* Asignamos el state S con los 64 bytes correspondientes al bloque i. */
    S = M[i]
    /* Asignamos la llave correspondiente a partir del valor que le corresponde. */
    K[0] = H[i]                          /* H[0] = 0 */
    S = AddRoundKey(S, K[0])             /* Se suma state y K[0] */
    PARA r = 1 HASTA 10 HAZ              /* Calculamos las llaves de ronda */
    INICIA
        K = SubBytes(K[r-1])
        K = ShiftColumns(K)
        K = MixRows (K)
        K[r] = AddRoundKey (K, C[r])
    FIN PARA
    /* Calculamos el hash según el siguiente procedimiento. */
    PARA r = 1 HASTA 10 HAZ
    INICIA
        S = SubBytes(S)
        S = ShiftColumns(S)
        S = MixRows(S)
        S = AddRoundKey(S, K[r])
    FIN PARA
    H[i+1] = H[i] ⊕ S ⊕ M[i]     /* Aplicamos la operación Miyhaguchi-Preneel. */
FIN PARA
FIN PARA
```

2.5.4 Finalización

El *hash value* corresponde al valor $\boldsymbol{H_t}$ después de procesar en su totalidad el mensaje quedando de la siguiente manera:

$$\text{WHIRLPOOL (M)} = H_t$$

3. No lineabilidad, mapas caóticos y atractores.

"Ningún hombre puede tener en su mente una imagen de cosas infinitas ni concebir la sabiduría infinita, el tiempo infinito, la fuerza infinita o el poder infinito. Cuando decimos de una cosa que es infinita, significamos solamente que no somos capaces de abarcar los términos y límites de la cosa mencionada, con lo que no tenemos concepción de la cosa, sino de nuestra propia incapacidad".

Thomas Hobbes
Leviatán, p.63.

En esta sección el objetivo principal es proporcionar un marco teórico acerca de la no lineabilidad de las funciones, objetos fractales y teoría del caos.

El estudio de los sistemas no lineales tiene diferentes campos de estudio entre los que destacaremos a los atractores, fractales, autómatas celulares, redes neuronales, algoritmos genéticos y lógica difusa. En su mayoría estos campos están relacionados, los atractores son usados en las redes neuronales, los fractales son usados en la compresión de datos, las redes neuronales y la lógica difusa son combinadas cuando las entradas del sistema no son claras.

Para empezar el desarrollo de este capítulo es necesario definir algunos conceptos que serán utilizados durante el planteamiento, posteriormente se describirán los fractales clásicos y sus propiedades especificas, posteriormente se definirán algunos conceptos relacionados con la aritmética de la geometría fractal de la cual derivaremos el algoritmo propuesto en el siguiente capítulo.

3.1 Antecedentes

En esta sección definiremos conceptos importantes que serán utilizados durante el desarrollo así como un breve resumen de lo referente al espacio euclidiano.

El **espacio n-dimensional** Euclidiano R^n, contiene subconjuntos tales como $R^1 = R$, el cual corresponde a los números reales de la recta, R^2 que corresponde al plano (Euclidiano) y así sucesivamente. Los puntos en R^n son denotados como coordenadas con la forma $x = (x_1,...,x_n), y = (y_1,...,y_n)$, etc.

Donde la suma es definida como $x + y = (x_1 + y_1,...,x_n + y_n)$ y el producto escalar $\lambda x = (\lambda x_1,...,\lambda x_n)$, donde λ es un escalar. La distancia euclidiana o métrica entre dos puntos en R^n es definida como: $| x - y |= \sqrt{\sum_{i=1}^{n} | x_i - y_i |^2}$

La región cerrada con centro en *x* y radio *r* es definida por $B_r(x) = \{y :| y - x |\leq r\}$.
Así mismo definimos a la región abierta como $B_r^0(x) = \{y :| y - x |< r\}$.

Lo que implica que la región cerrada incluye el límite y la abierta no. Como ejemplo de estas definiciones se encuentra en *R* a los intervalos cerrados y abiertos respectivamente.

Generalizando en R^n se le llama cubo coordenado de lado *2r* y centro $x = (x_1,...,x_n)$, al conjunto que cumple $\{y = (y_1,...,y_n) :| y_i - x_i |\leq r; \forall i = 1,...,n\}$. Un intervalo y un cuadrado cumplen con esta definición en *R* y R^2 respectivamente.

3.1.1 Funciones

Se le llama cuerpo paralelo δ (A_δ) de un conjunto A, al conjunto de puntos con distancia δ de A; esto es $A_\delta = \{x :| x - y |\leq \delta$ para alguna $y \in A\}$.

Una función *f: X→Y* es llamada una función Hölder de exponente α si:

$$| f(x) - f(y) | \leq c | x - y |^{\alpha}; (x, y \in X)$$

Para alguna constante *c*. La función es llamada de Lipschitz si α es igual a 1 y bi-Lipschitz si: $c_1 | x - y | \leq | f(x) - f(y) | \leq c_2 | x - y |; (x, y \in X)$ [1]

Se define al límite inferior de una función como $\underline{\lim}_{x \to 0} f(x) \equiv \lim_{x \to 0}(\inf\{f(x) : 0 < x < r\})$ y al límite superior como $\overline{\lim}_{x \to 0} f(x) \equiv \lim_{x \to 0}(\sup\{f(x) : 0 < x < r\})$.

Estos límites siempre existen (ya sea ∞ o tenga un valor) para cualquier función *f*. Si ambos límites existen y son iguales entonces el límite existe.

3.1.2 Mapa Unidimensional

Se le denomina así a la imagen de la función *f: S→ S, S ⊂ R*.

Donde el dominio *S* sería *S = [0, 1]* o *S = [-1,1]* en la mayor parte de los casos. Este resultado puede ser también denotado de la siguiente manera:

$$x_{i+1} = f(x_t), \qquad t = 0, 1, 2, \ldots, n \qquad x_0 \in S$$

Empezando con el valor inicial $x_0 \in S$ se obtiene, por iteración, la secuencia:

$$f(x_0), f(f(x_0)), f(f(f(x_0))), \ldots, f^n(x_0)$$

A dicha secuencia se le denomina orbita (o incluso trayectoria) generada por x_0. Estas trayectorias pueden ser periódicas, eventualmente periódicas, asintóticas, etc. El estudio de sistemas lineales se ocupa de analizar el comportamiento de la órbita cuando $t \to \infty$.

Durante este proceso existen ciertos valores (puntos) que determinan o dan información específica de la trayectoria que siguen. Tales puntos son conocidos, dependiendo de la

[1] Falconer, Kenneth J., Fractal Geometry: mathematical foundations and applications, John Wiley &Sons, 1990, Pág. 8.

característica particular del mapa, como punto fijo, punto periódico, punto eventualmente periódico y punto hiperbólico.

Definiremos como ***punto fijo*** al punto $x^* \in S$ del mapa *f* sí $f(x^*) = x^*$.

El ***punto periódico*** con período *n* es definido como el punto $x^* \in S$ tal que $f^{(n)}(x^*) = x^*$. En donde $f^{(n)}$ denota la *n-ésima* iteración de *f*, llamando a *n* el período de x^*. Al conjunto de todas las iteraciones de un punto periódico forman una órbita periódica.

Un punto x^* es ***eventualmente periódico*** de periodo *n* cuando x^* no es periódico pero existe una *m > 0* tal que se cumple $f^{(n+i)}(x^*) = f^{i}(x^*)$ para toda *i > m*. Es decir, $f^{(t)}$ es periódica para *i > m*.

A partir del punto periódico x^* con periodo *n*, definiremos a x^* como hiperbólico si $|(f^{(n)})'(x^*)| \neq 1$, al número $(f^{(n)})'(x^*)$ se le llama multiplicador del punto periódico.

Teorema[2]

Sea x^* un punto hiperbólico fijo con $|f'(x^*)| < 1$. Entonces hay un intervalo abierto *U* alrededor de x^* tal que si $x \in U$, entonces:

$$\lim_{n\to\infty} f^{(n)}(x) = x^*.$$

Un ejemplo de mapa unidimensional es el *mapa Bernoulli*. El cual se define como sigue: Sea $f:[0,1) \to [0,1)$ el mapa es definido como $f(x) = 2x \bmod 1 \equiv frac(2x)$. Que puede ser definido como la ecuación en diferencia siguiente:

$$x_{i+1} = \begin{cases} 2x_t & 0 \le x_t < 1/2 \\ (2x_t - 1) & 1/2 \le x_t < 1 \end{cases}$$

[2] Willi-Hans Steeb, The nonlinear workbook, World Scientific Publishing Co. Pte. Ltd, 1999, Pág. 2.

3.1.3 Mapa Bidimensional

Se le denomina así a la imagen resultante de las funciones $f_1: S \rightarrow S$ y $f_2: S \rightarrow S$, en S^2. Donde el dominio S^2 corresponde a un conjunto de puntos iniciales ubicados en el plano. Esto puede ser denotado de la siguiente manera:

$$x_{1t+1} = f_1(x_{1t}, x_{2t})$$
$$x_{2t+1} = f_2(x_{1t}, x_{2t})$$

Donde $t = 0, 1, \ldots, n$ los cuales corresponden a puntos (x_{1t}, x_{2t}) para $t = 0, 1, 2, \ldots, n$ en el plano R^2. A esto se le llama *phase portrait*.

3.1.4 Punto Fijo y Estabilidad

Dado un mapa bidimensional definido como $x_{1t+1} = f_1(x_{1t}, x_{2t})$, $x_{2t+1} = f_2(x_{1t}, x_{2t})$

Se definen como puntos fijos (x_1^*, x_2^*) aquellos valores que resuelven las siguientes expresiones:

$$f_1(x_1^*, x_2^*) = x_1^*, \quad f_2(x_1^*, x_2^*) = x_2^*$$

Un ejemplo de mapa bidimensional es el *mapa Hénon* el cual está dado por $f(x, y) = (y + 1 - ax^2, bx)$, donde *a* y *b* son los parámetros de la bifurcación con $b \neq 0$. Este mapa también puede representarse de la siguiente manera utilizando ecuaciones en diferencias:

$$x_{t+1} = 1 + y_t - ax_t^2$$
$$y_{t+1} = bx_t$$

El mapa es invertible si $b \neq 0$. Una visualización del comportamiento del mapa en que las líneas verticales son mapeadas a líneas en el eje horizontal, mientras que para $a > 0$ el mapeo de las líneas horizontales se mapean a parábolas abiertas cuya abertura es a la izquierda.

Para *a > 0* y *1 > b > 0* el mapa tiene dos puntos fijos los cuales son:

$$x^* = \frac{(b-1) \pm \sqrt{(1-b)^2 + 4a}}{2a}, y^* = bx^*$$

Estos puntos fijos son reales cuando $a > a_0 = \frac{(1-b)^2}{4}$

3.2 Fractales

El termino fractal engloba a un cierto tipo de objetos matemáticos, los cuales tienen características algebraicas y morfológicas únicas. Entre las que destacan la autosimilitud, el poseer una dimensión fraccionaria en la mayoría de los casos, entre otras. Cabe destacar que existen diferentes definiciones de fractales, las cuales se basan en la forma en la que se calcula la dimensión del objeto fractal o hacen referencia a alguna característica en particular.

A continuación daremos las definiciones de los neologismos dados por Benoît Mandelbrot algunos de los cuales serán utilizados durante el desarrollo de la presente sección:

Fractal: *adj. Sens intuitif. Se dit d'une figure géométrique ou d'un objet naturel qui combine les caractéristiques que voici. A) Ses parties ont la même forme ou structure que le tout, à ceci près qu'elles sont à une échelle différente et peuvent être légèrement déformées. B) Sa forme est, extrêmement irrégulière, soit extrêmement interrompue ou fragmentée, et le reste, que soit l'échelle d'examen. C) Il contient des «éléments distinctifs» dont les échelles sont très variées et couvrent une très large gamme*. [3]

Dimensión fractal: n. f. *Sens générique. Nombre qui quantifie le degré d'irrégularité et de fragmentation d'un ensemble géométrique ou d'un objet naturel, et qui se réduit, dans le cas des objets de la géométrie usuelle d'Euclide, à leur dimensions usuelles. Sens spécifique.*

[3] Les objets fractals forme, hasard et dimension, Benoit Mandelbrot, Flammarion. Pág. 154

"Dimension fractale" a souvent été appliqué à la dimension de Hausdorff et Besicovitch, mais cet usage est désormais très fortement déconseillé.[4]

Conjunto fractal: n. m. *Remplace le terme fractale lorsqu'il faut préciser qu'il s'agit d'un ensemble mathématique*.

Objeto fractal: n. m. *Remplace le terme fractale lorsqu'il faut préciser qu'il s'agit d'un objet naturel. Objet naturel qu'il est raisonnable et utile de représenter mathématiquement par une fractale*.

Polvo: n. f. *Collection entièrement discontinue de points, c'est-à-dire, objet de dimension topologique égale à 0*.

En la segunda parte de la definición brindada por Mandelbrot se establece que la dimensión fractal es el valor obtenido a través del método de *Hausdorff-Besicovitch* implicando que la dimensión es cuantificable y cabe resaltar aquí que dicho valor depende del procedimiento de cálculo y no esta acotada sólo al mencionado, lo que implicaría que la dimensión de un objeto/conjunto fractal cambie según la definición o método utilizado; a continuación se mostrarán algunas definiciones para el cálculo de la misma.

Ya que no es posible dar una definición única de fractal y de dimensión fractal, debido a la multiplicidad de las mismas, si podemos definir que características debe poseer un conjunto para ser considerado como un fractal las cuales son:

De forma general definiremos a ***F*** como un conjunto u objeto llamado fractal si:

- ***F*** tiene una estructura finamente detallada sobre escalas arbitrariamente pequeñas.
- ***F*** es muy irregular para ser descrita en el lenguaje de la geometría tradicional, tanto de forma local como global.
- A menudo ***F*** tiene algún grado de autosimilitud ya sea esta aproximada o estadística.
- Usualmente la dimensión fractal de ***F*** (definida de alguna forma) es más grande que su dimensión topológica.

[4] Ibídem. 155

- En la mayoría de los casos de interés ***F*** es definido de una sola forma, incluso de forma recursiva.

3.2.1 Conceptos de Dimensión

Si se utiliza la geometría elemental se observa a simple vista que algunos conjuntos geométricos tienen asociada una dimensión por ejemplo: los puntos tienen dimensión cero, las curvas dimensión uno, las superficies dimensión 2 y los sólidos dimensión 3. Sin embargo, si se deja de utilizar dicha geometría entonces existen conjuntos que no pertenecen claramente a uno de estos grupos.

A causa de este hecho y de estos conjuntos en especial, se crea el concepto de dimensión que de forma genérica se puede dividir en dos la ***dimensión topológica*** y la ***dimensión fractal***, la primera en correspondencia del hecho de utilizar la geometría elemental y la segunda para englobar a los demás conjuntos.

A la *dimensión topológica* también se le conoce como dimensión inductiva y parte de la explicación de Hermann Weyl: "*We say that space is 3-dimensional because the walls of a prison are 2-dimensional*"[5]. La idea básica de forma muy simple, consiste en encerrar al conjunto de puntos dado, con un conjunto de dimensión conocida, la dimensión del conjunto que lo envuelve debe ser la menor posible para a partir de esta deducir cual es la dimensión de dicho conjunto.

Bajo este concepto se tiene que la dimensión inductiva de un espacio métrico viene dado por un valor entero cuyo valor esta en $ind(x) = \{-1, 0, 1, 2, 3, \ldots, \infty\}$, de donde se tiene que $ind(\varnothing) = -1$ y así sucesivamente.

Se dice que dos espacios topológicos tienen la misma dimensión si existe una correspondencia uno a uno continua entre los puntos de cada uno, un ejemplo ilustrativo de esto son las llamadas curvas de Peano y curvas de Hilbert.

[5] Citado por Gerald A., Edgar en Measure, topology, and fractal geometry, Springer-Verlag, 1990, Pág. 79

3.2.2 Dimensión Topológica

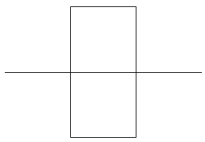
Esta definición parte de la idea básica de que una figura u objeto geométrico puede descomponerse en *n* copias a una escala *r* de sí misma (cabe decir que este es el método contrario al razonamiento de que al duplicar el lado de un cuadrado se obtiene un objeto geométrico cuatro veces mayor y cuando se trata de un cubo de ocho) de donde se deriva la siguiente relación $n = \left(\frac{1}{r}\right)^d$ aplicando logaritmos se obtiene: $d = \frac{\log n}{\log(1/r)} = \frac{\log n}{-\log r}$

Si se toma esta como definición de la dimensión de cualquier figura, que pueda ser descompuesta en *n* copias a escala *r* de sí misma, obtenemos una manera cómoda de asignar una dimensión a algunos conjuntos.

La única condición para aplicar este método es que el objeto pueda ser descompuesto en copias de sí mismo a escala. Con lo cual su utilización está limitada.

3.2.2.1 Curva de Peano

La curva de Peano se aplica genéricamente a una familia de curvas "*patológicas*" las cuales fueron decisivas en el concepto de dimensión topológica entre los años de 1890 a 1925. Giuseppe Peano en 1890 publico un artículo titulado "*Sur une courbe qui remplit toute une aire plane*", que fue el preludio de la llamada curva de Peano que dio a conocer en 1891, cuya característica principal es la de llenar el plano de tal forma que la curva pasa por cualquier punto de la superficie implicando que tiene como dimensión topológica 2.



El algoritmo para construir dicha curva parte de un segmento unitario, a partir del cual se generara la primera secuencia con 9 segmentos cuya longitud es de 1/3 de la inicial.

La manera en que se colocan dichos segmentos se muestra en la figura adyacente, está a su vez es conocida como generador. Las secuencias siguientes se generan a partir de la repetición de dicho procedimiento sobre cada uno de los segmentos. Dicho procedimiento se repetirá de forma indefinida e iterativa.

El objeto así generado es estrictamente autosemejante ya que existen *n* conjuntos semejantes al de la primera secuencia cada uno de ellos a una escala de 1/3 del anterior este dependerá del grado de la iteración como se ve a continuación.

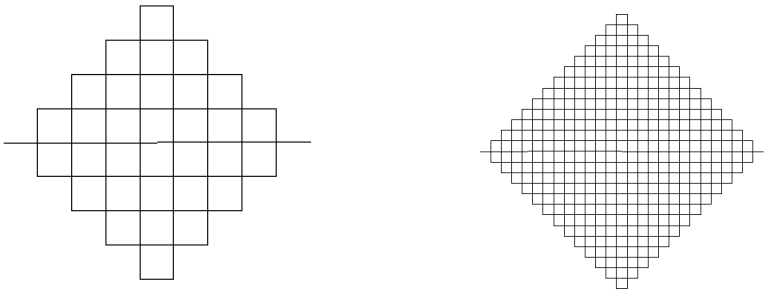


Figura 8. Curva de Peano

La dimensión de Hausdorff-Besicovitch (la cual se describirá en la siguiente sección) para esta curva está representada por la siguiente expresión: $D = \frac{\log S}{\log L}$

Donde *S* es la longitud total y *L* es la escala en dicha iteración por lo que para la curva de Peano se tiene que:

$$D = \frac{\log 9}{\log(1/3)} = \frac{\log 9}{\log 3} = 2$$

Lo que implica que su dimensión fractal es 2 la misma dimensión topológica de una superficie plana.

3.2.2.2 Curva de Hilbert

La curva de Hilbert[6] fue descrita por Hilbert en 1891 después de que Peano describiese la anterior. Para construir la curva de Hilbert se parte de un cuadrado unitario dividido en cuatro partes iguales y se unen los centros como se indica en el diagrama. En la secuencia

[6] Basado en: Proyecto Universitario de Enseñanza de las Matemáticas Asistida por Computadora. "La curva final" [en línea], 15 marzo 2008, http://interactiva.matem.unam.mx/fractales/html/curvafinal.html [Consulta: 15 de marzo 2008].

siguiente se realizan cuatro copias de la primera a escala ½ y se colocan en los cuadrantes uniendo cada uno de los segmentos. En la tercera iteración se realizan cuatro copias de la anterior a escala de ½ uniendo cada uno de los segmentos, dicho procedimiento se repite indefinidamente.

Se observa aquí que el objeto resultante no es estrictamente autosemejante. Conforme se aumente el número de secuencias la curva Hilbert irá rellenando el plano por tanto la dimensión topológica es 2.

En cada paso en la sucesión se tiene que dibujar 4 copias de la curva anterior a una escala ½, dos de estas copias van arriba, una abajo a la izquierda rotada 90 grados en dirección de las manecillas del reloj y la cuarta abajo a la derecha, rotada al revés. Estas cuatro partes hay que unirlas para obtener la siguiente curva en la sucesión.

La curva al final pasa por cada punto del cuadrado completo, pero por algunos puntos pasa más que una vez. La secuencia es mostrada a continuación:

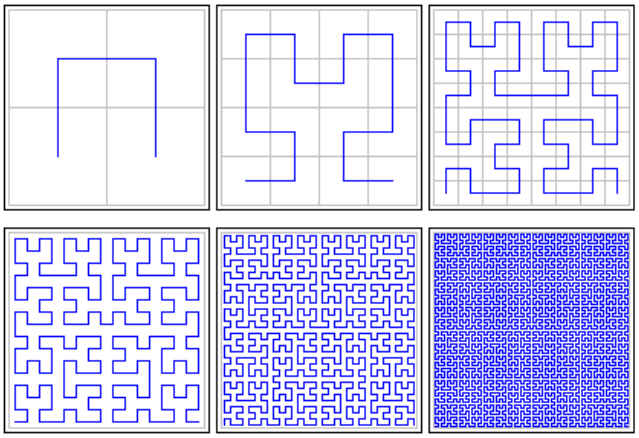


Figura 9. Curva de Hilbert

3.2.3 Dimensión Hausdorff

La dimensión Hausdorff tiene la ventaja de ser definida para cualquier conjunto y está basada en medidas las cuales son relativamente fáciles de usar. Sin embargo en la mayoría

de los casos es muy difícil de calcular con base en métodos computacionales. Para su cálculo es necesario definir una medida la cual es conocida como la *medida Hausdorff*.

Esta medida se basa en la idea de recubrir al conjunto de elementos que conforman el objeto fractal con un objeto de dimensión conocida. De manera formal se expresa de la siguiente manera:

Sea *U* un subconjunto no vacío *n*-dimensional en el espacio Euclidiano R^n, el diámetro de *U* está definido como $|U| = sup\ \{|x - y|: x, y \in U\}$.

Se le llama una cubierta de radio δ del conjunto ***F*** al conjunto contable definido por $F \subset \bigcup_{i=1}^{\infty} U_i$ .A partir de esto supongamos que ***F*** es un subconjunto de R^n y ***s*** es un número positivo.

Para cualquier δ>0 definimos: $H_{\delta}^{s}(F) = \inf\left\{\sum |U_i|^s\right\}$ donde $\{U_i\}$ es una cubierta δ de ***F.***[7]

Por lo que se define al límite $H^s(F) = \lim_{\delta \to 0} H_{\delta}^{s}(F)$ como la medida de Hausdorff.

La *dimensión Hausdorff* también llamada *dimensión Hausdorff – Besicovitch* de ***F*** es:

$$\dim_H F = \inf\{s : H^s(F) = 0\} = \sup\{s : H^s(F) = \infty\}.$$

Dicha expresión resume el procedimiento de cálculo de la dimensión, ***s*** indica el grado de recursividad o de escalamiento del objeto fractal, $\boldsymbol{\delta}$ indica el radio de la cubierta de ***F***, cabe destacar que la dimensión calculada será la correspondiente al nivel de recursividad ***s***, también llamado grado de escalamiento.

Una definición equivalente a esta es cambiar el concepto de cubierta $\boldsymbol{\delta}$ por cuerpos esféricos lo que queda expresado de la siguiente manera:

$B_{\delta}^{s}(F) = \inf\left\{\sum |B_i|^s\right\}$ donde $\{B_i\}$ es una cubierta $\boldsymbol{\delta}$ de ***F*** de forma esférica.[8]

[7] Falconer, Kenneth J., Fractal Geometry: mathematical foundations and applications, John Wiley &Sons, 1990, Pág. 25
[8] Ibídem. Pág. 32

Cabe destacar que el valor de la dimensión obtenido por cualquiera de estas definiciones puede no ser el mismo ya que aunque se basan casi en el mismo procedimiento de cálculo, se tiene como el único cambio notorio la definición de la cubierta δ ya que en la primera es una cubierta con un radio determinado y en la segunda dicha cubierta tiene la característica de ser esférica.

3.2.4 Calculo de Dimensión Hausdorff

3.2.4.1 Conjunto de Cantor

Sea *F* el conjunto de Cantor el cual se construye de la siguiente manera:

A partir del intervalo unidad $E_0 = [0, 1] \subset R$. Se divide dicho intervalo en tres segmentos iguales y solo se toman en cuenta los intervalos cerrados de los extremos es decir E_{11} = [0, 1/3] y E_{12} = [2/3, 1].

El proceso anterior se repite sobre los intervalos obtenidos. Cada uno de estos se divide en tres y se prescinde del intervalo central con lo cual solo se consideraran los intervalos E_{21} = [0, 1/9], E_{22} = [2/9, 1/3], E_{23} = [2/3, 7/9] y E_{24} = [8/9, 1], así sucesivamente.

Si consideramos $s = \log 2 / \log 3 = 0.6309$ entonces la $\dim_H F = s$ y $\frac{1}{2} \leq H^S(F) \leq 1$

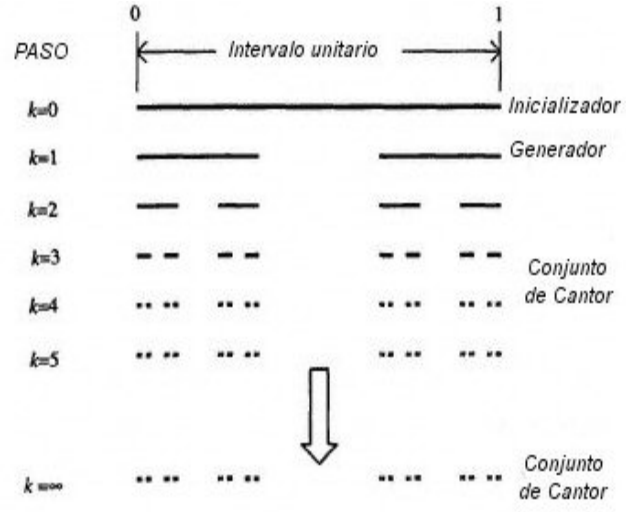


Figura 10. Conjunto de Cantor.

Otra forma de verlo es que el conjunto de Cantor F divide el segmento en una parte izquierda $F_I = F \cup [0,1/3]$ y en una derecha $F_D = F \cup [2/3,1]$.

Claramente ambas partes son geométricamente similares a F pero con una escala de 1/3 y $F = F_I \cup F_D$ el cual es un conjunto disjunto.

De esta forma para cualquier *s*

$$H^S(F) = H^S(F_I) + H^S(F_D)$$

$$H^S(F) = \frac{1}{3}^S H^S(F) + \frac{1}{3}^S H^S(F)$$

Asumiendo para el valor crítico $s = \dim_H F$ tenemos que $0 < H^s(F) < \infty$ y dividiendo por $H^s(F)$ se tiene $1 = 2\left(\frac{1}{3}\right)^S$ o $s = \log 2 / \log 3$ puede descomponerse en dos copias de sí mismo a escala 1/3 o en cuatro copias a escala 1/9 lo que da como resultado:

$$\dim(E) = \frac{\log 2}{\log 3} = \frac{\log 4}{\log 9} = 0.63092975$$

3.2.4.2 Curva de Koch

Esta curva fue definida en 1904 por Helge von Koch en el documento titulado "*Sur une courbe continue sans tangente, obtenue par une construction géométrique élémentaire*" la cual es también conocida como el copo de nieve de Koch o la estrella Koch, su construcción parte del proceso siguiente:

- Se divide en tres partes iguales al segmento unitario [0, 1]
- Se dibuja un triángulo equilátero tomando como base el segmento central obtenido del paso anterior.
- Se suprime la base del triángulo formado en el paso anterior quedando solo con 4 segmentos.
- Se repite el procedimiento anterior sobre cada uno de los segmentos resultantes.

Gráficamente es representado como:

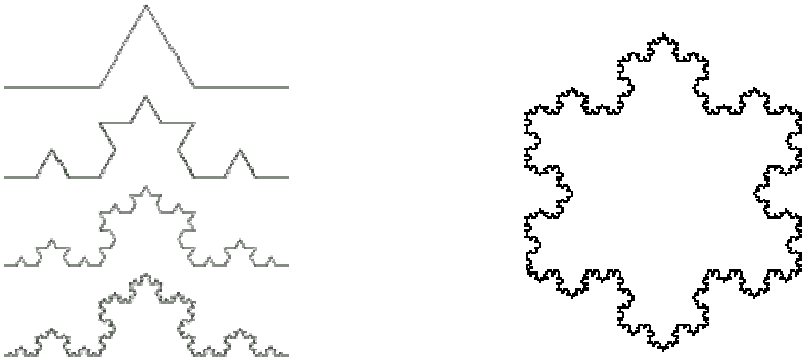


Figura 11. Curva de Koch

Si la figura generadora es un triángulo equilátero se obtiene la llamada estrella de David en la primera iteración.

Entre las características de esta curva se tienen:

- La curva tiene un perímetro infinito ya que este se incrementa de acuerdo al nivel de recursividad de la misma.
- En cada nivel de recursividad se tiene que la longitud de los segmentos que conforman a dicha figura es un tercio menor a la del estado anterior.
- La dimensión fractal es definida como: $\log 4 / \log 3 \approx 1.26$ la cual es más grande que la de una curva pero más pequeña que la de un área.
- La curva es continua sobre cualquier nivel de iteración pero no es diferenciable.

Esta curva puede ser expresada en términos del *sistema de Lindenmayer*[9] como sigue:

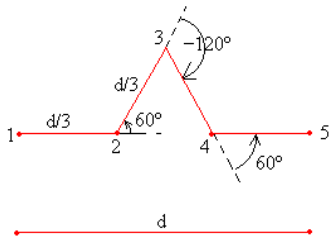


Figura 12. Función Lindenmayer de la curva de Koch

[9] Dicho sistema es semejante a una gramática de las utilizadas en lenguajes formales, puesto que sus elementos son un alfabeto (V) que contiene los símbolos posibles a sustituir, un conjunto de símbolos constantes (S), un generador o semilla (ω) y unas reglas de producción (P). Formalmente se representa como: G = {V, S, ω, P} y también es conocido como system-L.

Alfabeto	F	Donde:
Constantes	+, -	F significa dibujar
Axiomas	F++F++F	"+" girar 60° a la derecha
Regla de producción	F→F-F++F-F	"–" girar 60° a la izquierda

El área de la estrella de Koch está definida por la expresión $\frac{2\sqrt{3}s^2}{5}$ en donde "s" es la longitud de uno de los lados del triángulo original.

La curva de Koch es un caso especial de la curva de César donde $a = \frac{1}{2} + \frac{i}{\sqrt{12}}$ el cual a su vez es un caso especial de la curva Rham.

Al repetir dicho proceso de forma ininterrumpida hasta la k-ésima iteración se tiene una figura cuya longitud es de $(4/3)^k$. La curva así descrita define el limite al que converge la sucesión cuando *k* tiende a infinito, dando como consecuencia una curva de longitud infinita puesto que $(4/3)^k$ tiende a infinito junto con *k*. El área bajo la curva, por otra parte viene dada por la serie:

$$1 + \frac{4}{9} + \left(\frac{4}{9}\right)^2 + \left(\frac{4}{9}\right)^3 + ... + \left(\frac{4}{9}\right)^k$$

La cual converge a 9/3 asumiendo que el área de la primera iteración es 1.

3.2.5 Dimensión por Conteo de Cajas.

Esta definición es la más conocida y la más fácil de calcular de forma empírica. Es también conocida por términos como entropía de Kolmogorov, dimensión métrica, densidad logarítmica o dimensión de la información. Cuya forma de cálculo viene dada por:

$$\dim_B F = \lim_{\delta \to \infty} \frac{\log N_\delta(F)}{-\log \delta}$$

Donde:

$N_\delta(F)$: Es el menor número de conjuntos de diámetro a lo mucho de δ que cubren a ***F***.

El cálculo de la dimensión, utilizando esta definición, es de forma empírica por lo que aquí se listaran algunos caminos para calcularla:[10]

- El menor número de cuerpos esféricos de radio δ que cubren a ***F***.
- El menor número de cubos de lado δ que cubren a ***F***.
- El número de cubos de la malla δ que interceptan a ***F***.
- El menor número de conjuntos de diámetro a lo máximo de δ que cubren a ***F***.
- El mayor número de cuerpos esféricos disjuntos de radio δ con centro en ***F***.

Como se ve el concepto intuitivo de dimensión es multiforme: la dimensión de Hausdorff-Besicovitch, la dimensión d'homothétie y la dimensión topológica no representan más que un aspecto en particular. Y que pueden tomar valores diferentes según sea el caso.

3.2.6 Dimensión Fractal

Aunque el concepto de dimensión de Hausdorff es esencial en la geometría fractal, su definición es relativamente compleja y en la práctica se utilizan otras definiciones de dimensión que resultan, además, de gran valor a la hora de determinar empíricamente la dimensión de series de datos obtenidas del mundo real, y que suelen coincidir con la dimensión de Hausdorff en los casos más interesantes. La más extendida dentro de esta categoría es la denominada dimensión fractal. En este apartado restringiremos nuestro estudio a la dimensión de conjuntos compactos.

Sea $A \in H(R^n)$ un conjunto compacto y no vacío de R^n. Sea N(A, ε) el menor número de cuerpos esféricos cerrados de diámetro $\varepsilon > 0$ necesarias para cubrir A. Si existe:

$$D = \lim_{\varepsilon \to 0} \{\log N(A, \varepsilon) / \log(1/\varepsilon)\}$$

Entonces *D* se denomina dimensión fractal de A. Se escribiría también $D = D(A)$ para indicar que A tiene dimensión fractal D.

[10] Falconer, Kenneth J., Fractal Geommetry: mathematical foundations and applications, John Wiley &Sons, 1990. Pág. 41

Como ejemplo de lo anterior se toma la curva de Koch en la que se necesita una bola de diámetro 1 para cubrir todo el conjunto, 4 bolas de diámetro 1/3, 16 bolas de diámetro 1/9, 64 bolas de diámetro 1/27, etc. En general, son necesarias *4n* bolas de diámetro $(1/3)^n$ para cubrir la curva de Koch. Su dimensión fractal será por tanto.

$$D = \lim_{n\to\infty} \frac{\log 4^n}{\log 3^n} = \frac{\log 4}{\log 3} = 1.261859507....$$

Lo que indica que la curva está más cerca de ser una curva que un área (nótese que para que el diámetro tienda a cero, *n* ha de tender a infinito).

Como se ha mencionado anteriormente la dimensión fractal y la dimensión de Hausdorff coinciden. Más concretamente puede demostrarse el siguiente teorema.

Sea *n* un entero positivo y sea *A* un subconjunto de R^n. Si *D(A)* denota la dimensión fractal de *A* y $D_H(A)$ la dimensión de Hausdorff de *A*, entonces:

$$0 \le D_H(A) \le D(A) \le n$$

Esto queda de manifiesto en el siguiente teorema en el que se sustituye la variable continua ε por una variable discreta.

Teorema de recuento por cajas: Sea $A \in H(R^n)$ un conjunto compacto y no vacío de R^n. Cubramos R^n mediante cajas cuadradas cerradas con lados de longitud $(1/2)^m$. Sea $N_m(A)$ el número de cajas con lado de longitud $(1/2)^m$ que intersectan con A. Si

$$D = \lim_{m\to\infty} \frac{\log N_m(A)}{\log 2^m}$$

Entonces *A* tiene dimensión fractal *D*. Que en otras palabras especifica el procedimiento de colocar una malla sobre el conjunto a medir y contar el número de cajas en las que hay algún punto del conjunto.

3.3 El Inicio del Caos.

El termino caos describe el comportamiento de varios sistemas dinámicos los cuales presentan un comportamiento aperiódico (las oscilaciones regulares no se presentan durante un período infinito) resultado de un modelo totalmente determinista y que presenta gran sensibilidad a las condiciones iniciales.

Esta sensibilidad puede implicar ya sea una divergencia exponencial de trayectorias inicialmente muy próximas en el espacio, conocida como alejamiento o su opuesta el acercamiento (plegamiento), que conlleva como consecuencia que dos trayectorias lejanas eventualmente se acerquen. Dicho comportamiento parece ser aleatorio aunque el sistema sea determinístico a partir de las condiciones iniciales.

Las propiedades necesarias para que un sistema dinámico sea considerado como caótico son:

- Ser sensible a las condiciones iniciales
- Debe ser topológicamente mixto
- Sus orbitas periódicas deberían ser densas

La sensibilidad a las condiciones iniciales significa que cualquier punto arbitrariamente cercano puede generar una trayectoria diferente conforme pasa el tiempo, es decir, una pequeña perturbación de la trayectoria actual podría significar un comportamiento futuro diferente. A esta sensibilidad se le conoce como efecto mariposa a causa del título del artículo de Edward Lorenz publicado en 1972, "*Predictability: Does the Flap of a Butterfly's Wings in Brazil set off a Tornado in Texas?*" en el cual se plantea que un pequeño cambio en las condiciones iniciales puede desencadenar una serie de eventos a gran escala.

El ser topológicamente mixto significa que el sistema evolucionara a través del tiempo hasta que cualquier región dado o conjunto abierto de su espacio de fases eventualmente se sobrelaparía con cualquier otra región. Aquí el término mezclar corresponde a la intuición estándar: la mezcla de tonalidades de colores o fluidos es un ejemplo de un sistema caótico.

Una consideración importante es que un sistema lineal nunca es caótico ya que para que un sistema dinámico presente un comportamiento caótico tiene que ser no lineal, sin embargo, un sistema dinámico discreto puede presentar un comportamiento caótico en un espacio unidimensional o bidimensional (como el mapa logístico).

3.3.1 Atractores

Algunos sistemas dinámicos son caóticos pero en muchos casos el comportamiento caótico es encontrado solamente en un subconjunto de la fase. Los casos de mayor interés surgen cuando el comportamiento caótico tiene lugar sobre un atractor, dado que un gran conjunto de condiciones iniciales darán a lugar a órbitas que convergen a esta caótica región.

Una forma fácil de visualizar un atractor es comenzar con la región de influencia del atractor y simplemente graficar la órbita subsiguiente. A causa de la condición de la transitividad topológica esto comúnmente producirá una figura del atractor entero final.

Los atractores en general son muy simples (puntos o líneas curvas), pero en ocasiones surgen atractores con movimiento caótico que son conocidos como atractores extraños que pueden ser muy complejos. Un ejemplo de este tipo de atractores es el modelo tridimensional del clima de Lorenz. Cuya característica principal es que la trayectoria parece formar las alas de una mariposa.

Otros atractores como el obtenido por el mapa Rössler tienen un punto de bifurcación semejante al del mapa logístico, que será detallado más adelante.

Este tipo de atractores no son específicos de los sistemas dinámicos continuos como es el caso de Lorenz sino también de los discretos como en el mapa Hénon. También existen los llamados conjuntos de Julia que tienen un comportamiento semejante cerca de los puntos fijos. Ambos tienen una estructura que es típicamente fractal.

Cabe destacar que para el caso de un mapa de un sistema dinámico discreto como el mapa logístico se puede observar que las trayectorias nunca se interceptan entre sí, puesto que si lo hicieran implicarían un comportamiento periódico. Como la región en la que se ubica el

atractor es finita, se tiene, al seguir una trayectoria cualquiera, una curva de longitud infinita encerrada en un área finita o dicho de otra forma posee estructura fractal.

Uno de los ejemplos más representativos del caos (y mejor conocido en la literatura) dentro de la naturaleza es la vida. Regida por lo que llamamos leyes de la naturaleza, los materiales son compuestos químicos y aunque son conocidos y definidos, el resultado es variable y sorpresivo. Si nosotros tratáramos de representar este hecho de forma matemática entonces obtendríamos una regla fija, la cual transformaría una entrada en una salida, la cual es la entrada en el siguiente estado y así sucesivamente, a este tipo de ecuación o modelo se le llama retroalimentación el cual consiste en reinsertar el resultado de forma consecutiva.

Este problema es típicamente un sistema dinámico el cual puede ser representado de manera muy simple por la ecuación *f(p) = p+z*.

Donde *p* es la población original y *z* es el crecimiento de la misma.
El valor de *z* es el número de hijos dependiente de la población inicial adulta que puede ser expresada de la siguiente manera: *f(p) = p + k * p * (1-p)*
Si volvemos a esta ecuación iterativa obtenemos lo siguiente:

$$f(p_0) = p_0 + kp_0(1 - p_0) = p_1$$
$$f(p_0) = p_1 + kp_1(1 - p_1) = p_2$$
$$\ldots$$
$$f(p_n) = p_n + kp_n(1 - p_n) = p_{n+1}$$

A esta ecuación se le denomina ecuación logística de la cual hablaremos a continuación.

3.3.2 El Mapa Logístico

El mapa logístico es un mapeo polinomial utilizado a menudo para ejemplificar como un sistema determinístico con un comportamiento definido puede derivar en un comportamiento caótico al variar las condiciones iniciales. Dicho mapeo fue utilizado por el biólogo Robert May para representar de forma algebraica el comportamiento del

crecimiento de una población a través del tiempo, dicho modelo es representado de la siguiente forma: $x_{n+1} = rx_n(1 - x_n)$

Al variar el parámetro *r*, la ecuación presenta uno de los siguientes comportamientos:

- Entre 0 y 1 la población muere independientemente de la población inicial.
- Entre 1 y 2 la población rápidamente se estabilizará al valor (r-1)/r independientemente de la población inicial.
- Entre 2 y 3 la población también se estabilizará eventualmente al mismo valor pero primero oscilara durante un tiempo. La velocidad de convergencia es lineal, excepto para r = 3 cuando dramáticamente disminuye, menos que la lineal.
- Entre 3 y 1+√6 (aprox. 3.45) la población podría oscilar entre dos valores siempre. Estos valores dependen de *r*.
- Entre 3.45 y 3.54 la población oscilaría entre 4 valores
- Al incrementar el valor a más de 3.54 la población probablemente oscile entre 8 valores, después a 16, 32 y así sucesivamente. La distancia entre cada uno de los puntos del intervalo sobre el cual oscila disminuye rápidamente entre el par de intervalos sucesivos entre los que se bifurcan a una constante llamada de Feigenbaum. Este comportamiento es un ejemplo de un período de doble cascada.
- Para la mayoría de los valores superiores a 3.57 presentan un comportamiento caótico pero para varios valores aislados de *r* parecen mostrar un comportamiento no caótico estos valores son a menudo llamados islas de estabilidad. Un ejemplo de estas es el valor 1+√8 (aproximadamente 3.83) en la que hay un rango de parámetros *r* para el cual muestra una oscilación entre 3 valores.
- Para un valor mayor a 4 los valores resultantes abandonan el intervalo [0, 1] y divergen para la mayoría de los valores iniciales.

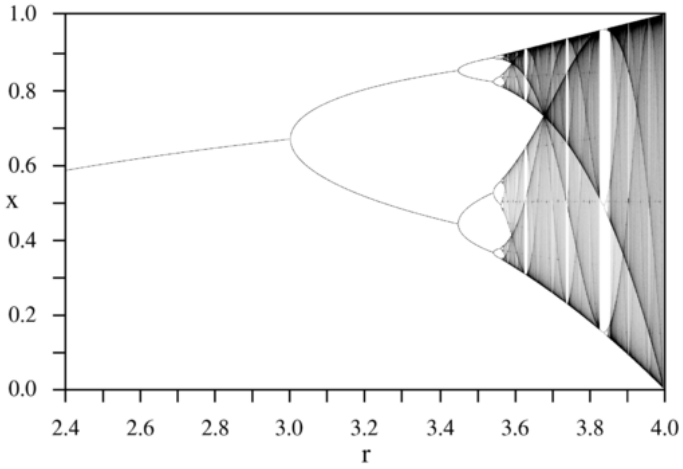


Figura 13. Mapa logístico

3.3.3 Constante de Feigenbaum

La ecuación logística se ha convertido en la manera usual de introducir las características del caos. Se trata de una ecuación en diferencias que fue formulada por Verhulst en el siglo pasado para explicar el crecimiento de una población perteneciente a la misma especie y que se reproduce en un entorno cerrado sin ningún tipo de influencia externa. Pese a su aparente sencillez, constituye un buen ejemplo para mostrar el comportamiento de los sistemas caóticos.

La ecuación se puede escribir como: $x_{n+1} = rx_n(1-x_n)$ donde el parámetro *r* es una constante denominada parámetro de crecimiento (generalmente entre 0 y 4) y la variable x_n puede verse como la fracción máxima de población que el ambiente puede soportar en el instante t_n.

Considerando que la población limite está dada por $x_\infty = \lim_{n\to\infty} x_n$ y que esta existe. Se observa que la dependencia entre el parámetro de crecimiento *r* con respecto a x_n, es la siguiente para valores de *r < 3* el sistema converge a un punto fijo estable, que es cero cuando *r < 1*. Cuando *r > 3*, el punto fijo se hace inestable y el valor de x_∞ oscila entre dos valores a este hecho se le conoce como duplicación de periodo.

Si se aumenta *r* ligeramente, por ejemplo *r = 3.44* el número de puntos sobre los que oscila x_∞ es de 4. Si se sigue aumentando el valor de *r*, aparece una nueva duplicación de periodo para *r = 3.544*, obteniendo un período de 8. Y así sucesivamente hasta llegar a obtener una sucesión de infinitos valores para x_∞ correspondiente al caos.

El comportamiento de la ecuación logística en función de *r* puede observarse visualmente a través de un diagrama de bifurcación. En el eje horizontal se representa un cierto intervalo de valores de *r* y entonces se dibujan los valores de *x* generados por la iteración en el eje vertical.

La duplicación del período es un signo ineludible del comportamiento caótico de un sistema. Un hecho que demuestra que hay un cierto orden en el caos es el descubrimiento realizado por Feigenbaum a mediados de los setenta de la constante que lleva su nombre.

Una vez obtenido el diagrama de bifurcación de la ecuación logística, se puede calcular el incremento del parámetro entre dos bifurcaciones contiguas $\Delta i = r_i - r_{i-1}$ y dividiendo por el incremento en el siguiente intervalo:

$$\frac{\Delta i}{\Delta i + 1} = \frac{r_i - r_{i-1}}{r_{i+1} - r_i}$$

Feigenbaum encontró que la fracción anterior convergía hacia un valor determinado al ir haciendo *i* mayor, de modo que en el límite se obtenía:

$$\delta = \lim_{i \to \infty} \frac{\Delta i}{\Delta i + 1} = 4.669201609102990671853203820466 2...$$

Feigenbaum calculo el límite anterior para otras ecuaciones en diferencias y obtuvo el mismo valor. El comportamiento caótico descrito anteriormente no sólo surge bajo sistemas discretos.

El llamado diagrama de Feigenbaum describe la conexión entre dos conceptos los cuales son parecidos pero distintos orden y caos, diferenciándose entre ellos por los valores de un parámetro. Implicando los dos lados opuestos de la misma moneda. Todos los sistemas no

lineales despliegan típicamente esta transición por lo que generalmente se habla de un escenario de Feigenbaum.

Si graficamos la población anterior (p_n) y la actual (p_{n+1}) de la ecuación logística de la siguiente manera <*p, f(p, k)*> se obtiene una gráfica que es denominada atractor en lugar de graficar de la forma normal (*k, p*).

3.3.4 El Mapa Hénon

Douglas Hofstadter describe al mapa Hénon como un sistema dinámico discreto representativo de los sistemas que exhiben un comportamiento caótico. Este define la secuencia de puntos (x, y) en el plano generado a partir de las formulas recursivas:

$$x_{n+1} = y_n - Ax_n^2 + 1$$
$$y_{n+1} = Bx_n$$

El mapa depende de dos parámetros *A* y *B*, los cuales para el mapa canónico de Hénon tiene los valores de *A = 1.4* y *B = 0.3*. Para estos valores el mapa es caótico para otros valores de *A* y *B* el mapa podría ser caótico, intermitente o convergente a una órbita periódico. Dicho comportamiento puede ser obtenido a partir de su diagrama de órbita

Figura 14. Mapa Hénon

Este mapa es un modelo simplificado de la sección de Poincaré del modelo de Lorenz. Para el mapa canónico un punto inicial del plano se aproximaría a un conjunto de puntos conocidos como el atractor de Hénon o diverge al infinito. El atractor Hénon es un fractal, en una dirección y un conjunto de Cantor en la otra.

Los valores numéricos estimados para la dimensión de correlación es 1.42 ± 0.02 y para la de Hausdorff de 1.261 ± 0.003 para el atractor del mapa canónico.

En 1968 Michel Hénon propone al Instituto de Astrofísica en París, tomar simples mapas cuadráticos para modelar el comportamiento de los satélites, asteroides o de las partículas cargadas en un acelerador de partículas.

Durante el período de 1954 - 1963 los matemáticos Kohnogorov, Arnold y Moser desarrollaron una teoría centrada alrededor del también llamado teorema KAM[11]. En el que estudian el comportamiento de sistemas dinámicos estables, cuando son afectados por pequeñas perturbaciones debidas a fuerzas externas que pueden provocar inestabilidad o un comportamiento caótico.

3.3.5 El mapa bidimensional Baker

El mapa Baker es el sistema dinámico plano más simple con un atractor fractal, también es conocido como la transformación Baker porque en sí mismo es el proceso repetitivo de estrechar un pedazo de masa y encimarla reiteradamente. Algebraicamente esto puede ser representado como sigue:

Sea $E = [0,1] \times [0,1]$ que representa el cuadrado unitario y $0 < \lambda < ½$ definimos a la transformación de Baker como:

$$f(x,y) = \begin{cases} (2x, \lambda y) & (0 \le x \le 1/2) \\ (2x-1, \lambda y + 1/2) & (1/2 < x < 1) \end{cases}$$

Esta transformación puede ser vista como el estrechamiento de ***E*** en un rectángulo $2 \times \lambda$, cortando este en dos rectángulos de $1 \times \lambda$, colocando uno de ellos encima del otro con un segmento en medio de $½ \times \lambda$. Entonces $E_k = f^k(E)$ es una secuencia decreciente de conjuntos con E_k comprimiendo 2^k segmentos horizontales de peso λ^{-k} separados por intervalos de al menos $(½ - \lambda)\lambda^{1-k}$. Ya que $f(E_k) = E_{k+1}$, el límite del conjunto en su totalidad es: $F = \bigcap_{k=0}^{\infty} E_k$ satisface $f(F) = F$. Si $(x, y) \in E$ entonces $f^k(x, y) \in E_k$, así $f^k(x, y)$ está

[11] Dicho teorema es descrito de forma básica por Jürgen Pöschell en: "A Lecture on the Classical KAM Theorem, Versión 1.2, Dec 2000, Proc. Symp. Pure Math, 69 (2001) 707-732".

situado a una distancia λ^{-k} de ***F***. De esta manera todos los puntos de ***E*** son atraídos a ***F*** en cada iteración de ***f***.

Con lo que se muestra que ***f*** tiene dependencia sensitiva sobre las condiciones iniciales y que los puntos periódicos de ***f*** son densos en ***F***, así que ***F*** es un atractor caótico para ***f***. Ciertamente ***F*** es un fractal ya que es el producto *[0,1] x* F_1, donde F_1 es un conjunto de Cantor uniforme obtenido por el repetido remplazo de intervalos ***I*** para un par de sub-intervalos de longitud $\lambda|I|$.

La transformación Baker es un buen ejemplo para ilustrar como el proceso de estrechamiento y división resulta en un atractor fractal. Otra forma de ver a dicha transformación es como el mapeo caótico del cuadrado unidad en sí mismo equivalente a la operación de amasado realizada por los panaderos, la masa se corta a la mitad y las mitades se apilan una sobre otra y finalmente se comprimen.

El mapa de Baker es estudiado por su acción sobre el espacio de funciones definidos sobre el cuadrado unitario, él define un operador sobre el espacio de funciones, conocido como el operador de transferencia del mapa. El mapa de Baker es un modelo exacto de caos determinístico, en el que las eigenfunciones y eigenvalores del operador de intercambio pueden ser explícitamente determinados.

Hay dos definiciones alternativas del mapa de Baker que se usan a menudo. Una definición despliega o rota una de las mitades antes de incorporarla de nuevo (similar al mapeo de horseshoe también conocido como mapeo de herradura) y la que no realiza esto.

El mapa Baker actúa sobre el cuadrado unidad como:

$$S(x,y)=\begin{cases}(2x,y/2) & 0\le x<\dfrac{1}{2}\\ (2-2x,1-y/2) & \dfrac{1}{2}\le x<1\end{cases}$$

Cuando la sección superior no es plegada, el mapa podría ser escrito como:

$$S(x,y)=\left\{2x-\lfloor 2x\rfloor\ ,\ \frac{y+\lfloor 2x\rfloor}{2}\right.$$

3.3.6 Atractor Lorenz

En 1963 Edward Lorenz publico su trabajo titulado "*Deterministic nonperiodic flow*" en el que detalla el comportamiento de un modelo matemático simplificado, que representa su trabajo con relación a la atmósfera. En el que mostró que un modelo simple determínistico podría generar un comportamiento impredecible (mayormente llamado caos). El modelo de Lorenz contiene tres variables x, y, z.

Dicho modelo es un conjunto de ecuaciones diferenciales con parámetros a, b, c. Cuya estructura es tridimensional y refleja un comportamiento caótico. Dichas ecuaciones son:

$$x' = a(x - y)$$
$$y' = x(b - z) - y$$
$$z' = xy - cz$$

Como se ve en la definición de las ecuaciones este posee dos términos no lineales el *xz* y *xy* los cuales presentan un movimiento periódico o caótico dependiendo del valor de los parámetros de control *a*, *b* y *c*.

Cuyas soluciones son complicadas pero cuya interpretación fue muy interesante ya que permitía una descripción matemática racional de la impredecibilidad del clima en la meteorología. Ya que en el modelo se observa que el sistema oscila en la región positiva de x por un par de veces y posteriormente cambia a la región negativa de x por un par de oscilaciones, regresando nuevamente a la parte positiva unas cuantas oscilaciones y así sucesivamente. Sin embargo, el sistema nunca repite su comportamiento, es decir, nunca se cruza una repetición oscilatoria en el transcurso del tiempo por lo que su comportamiento es aperiódico.

Este modelo propuesto no fue capaz de explicar la complejidad de la termodinámica de la atmósfera, pero a partir de esto se establecieron dos puntos:

- La imposibilidad de obtener una predicción del clima, Lorenz lo estableció diciendo que "el aleteo de una mariposa puede influir en el clima". A lo que llamo el efecto mariposa.
- La esperanza de poder explicar comportamientos complejos a través de modelos matemáticos simples aumenta.

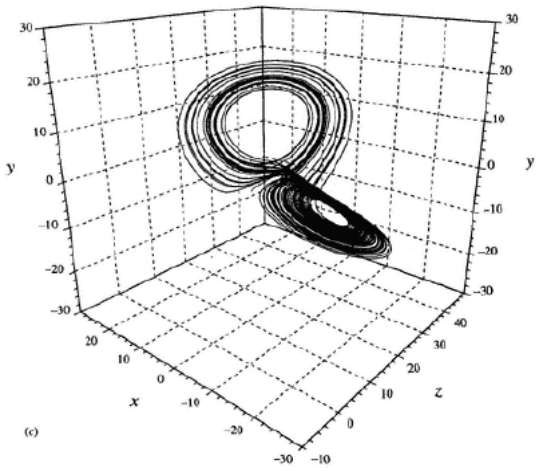


Figura 15. Atractor de Lorenz

3.3.7 Conjuntos de Julia

Los conjuntos de Julia son el resultado de la iteración de un sistema dinámico que utiliza parámetros en el plano complejo *C*. A partir de los trabajos realizados por los matemáticos franceses Gaston Julia y Pierre Fatou acerca de estos sistemas se generó la teoría de sistemas dinámicos complejos a comienzos de este siglo.

Ambos matemáticos trabajaron sobre el plano complejo con resultados complementarios ya que Julia encontró en este puntos sobre los cuales un conjunto de puntos orbitaba a su alrededor a los cuales llamo atractores. Por su parte Fatou encontró puntos cuyo comportamiento era el contrario puntos que tendían alejar a los cercanos según la órbita generada.

A partir de este hecho se puede definir a un conjunto de Julia y a un conjunto de Fatou de forma complementaria de las siguientes maneras:

Los conjuntos de Julia $f : C \to C$ son definidos por polinomios de grado $n \geq 2$ con coeficientes cuya forma es $f(z) = a_0 + a_1 z + ... + a_n z^n$. Si *f(w)=w* donde *w* es un punto fijo de *f*, por lo que al conjunto formado por $w, f(w),..., f^p(w)$ se le conoce como una órbita de

período *p*. Se define a la derivada en el plano complejo como $f'(z) = \lim_{w\to 0}(f(z+w) - f(z))/w$.

Si mezclamos este concepto con el de punto fijo se tiene $(f^p)'(w) = \lambda$ donde el punto *w* es llamado de acuerdo al valor de λ:

Super atractivo	λ = 0
Atractivo	0 ≤ |λ| < 1
Indiferente	|λ| = 1
Repelente	|λ| > 1

Tabla 5. Propiedades del punto fijo.

De tal forma que el conjunto de Julia *J(f)* de *f* podría ser definido como la cerradura del conjunto de los puntos de repulsión periódicos *f*, denotado como *J(f)*. Al complemento de este conjunto se le llama conjunto de Fatou o conjunto estable *F(f)*.

Como ejemplo de éste se tiene que el conjunto de Julia de $f(z) = z^2$ es el círculo $|z| = 1$ con las iteraciones $f^k(z) \to 0$ si *z* esta dentro de *J* y $|f^k(z)| \to \infty$ si *z* esta fuera de *J*.

Pero si *f* es afectada con una pequeña variación de la siguiente forma $f(z) = z^2 + c$ entonces la región que contiene a los puntos para los cuales $f^k(z)$ converge hacia el punto fijo *w* cercano a cero es fractal.

De forma simple se puede definir al conjunto de Julia como: "*el conjunto de todos los puntos resultantes de la iteración de un polinomio de variable compleja que no escapen al infinito es decir que convergen en una frontera definida la cual es característica de cada conjunto y que además es fractal*".

Donde se entenderá como la órbita de un punto *x* en un sistema dinámico *f* a la sucesión de puntos $\{f^n(x)_{n=0}^{\infty}\}$.

El factor de perturbación *c* define al conjunto de puntos que conforman al conjunto de Julia de forma característica. Una implicación adicional de este es que permite clasificar a los conjuntos de Julia en dos categorías la de los conjuntos conectados y la de los desconectados.

Cabe resaltar que la órbita divergirá hacia el infinito si en algún momento uno de sus puntos tiene módulo mayor o igual a 2.

El conjunto Fatou *F(f)* de *f* es el complemento del conjunto de Julia compuesto de aquellos puntos que tienen un comportamiento estable.

3.3.8 El Conjunto de Mandelbrot

Ya que los conjuntos de Julia son definidos por polinomios de la forma $f(z) = z^2 + c$. Y que a partir del parámetro c puede ser un conjunto conectado o desconectado.

Se define al conjunto de Mandelbrot *M* como el conjunto de parámetros *c* para los cuales el conjunto de Julia de f_c esta conectado, lo cual se puede expresar como:

$$M = \{c \in C : J(f_c)\} \text{ esta conectado.}$$

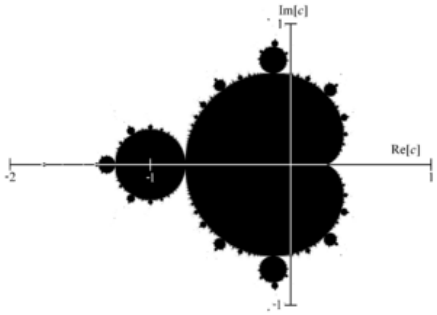


Figura 16. Conjunto de Mandelbrot

El conjunto de Mandelbrot gráficamente tiene la forma mostrada en la figura adyacente y corresponde al subconjunto del plano complejo para los cuales el parámetro *c* generará un conjunto de Julia conexo.

El conjunto de Mandelbrot es compacto y sus límites están acotados a una circunferencia de radio 2 aproximadamente.

Si $|c|>\frac{1}{4}(5+2\sqrt{6})$ entonces $J(f_C)$ está totalmente desconectado y el conjunto resultante es la unión de dos ramas de $f_c^{-1}(z)$ que presentan una reducción, que no altera sus características principales. Ahora bien si el módulo de |*c*| ≥ 2 para alguna *n* en la secuencia entonces escapará al infinito.

El área del conjunto de Mandelbrot ha sido estimada en 1.50659177 ± 0.00000008, además de que el conjunto de Mandelbrot a su vez es un conjunto conectado.

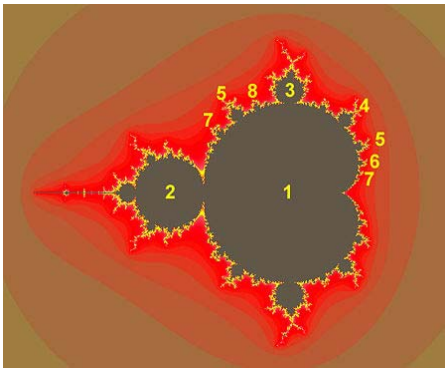


Figura 17. Conjunto de Mandelbrot

El conjunto de Mandelbrot está formado por un cardioide principal el cual comprende la región de todos los puntos *c* para los cuales sólo se tiene un punto fijo de atracción.

Dichos parámetros tienen en general la forma siguiente:

$$c=\frac{1-(\mu-1)^2}{4}$$

Para alguna μ en el disco unitario.

El bulbo ubicado a la izquierda del cardiode principal contiene a los parámetros para los cuales existe un ciclo de atracción de período 2 y están acotados a una circunferencia de aproximadamente ¼ alrededor de -1.

A partir de la elección del parámetro *c* dentro del conjunto de Mandelbrot se puede deducir la forma que tendrá el conjunto de Julia puesto que está definida con base en un subconjunto del mismo conjunto de Mandelbrot de ahí que también sirva como un mapa de predictibilidad de la forma del conjunto de Julia generado.

3.4 Principios de diseño para cifrado basado en caos.

Con base en los algoritmos de cifrado basados en caos se pueden derivar algunas reglas de diseño que están fundamentadas en la fortaleza y debilidades de los sistemas que han sido propuestos. Dichas reglas engloban varios rubros tales como la estructura del sistema, el espacio de llaves, la no lineabilidad y dinámica de los mapas caóticos en los que se basan. A continuación presentaremos estas de forma general.

3.4.1 Estructura del sistema

La estructura de los algoritmos de cifrado basado en caos tienen dos requerimientos principales:

- A fin de poder descifrar el texto la operación de cifrado debe ser invertible.
- Para la mayoría de las implementaciones de sistemas de cifrado basados en caos (de tiempo discreto) los mapas caóticos deben ser seleccionados para mantener propiedades importantes como son la mezcla o densidad uniforme de la señal generada cuando esta es digitalizada.

En criptografía clásica existen dos tipos los cifrados por bloque y por flujo. Los cifrados por bloque mapean bloques de texto en claro a bloques de texto cifrado dicho modo de operación es llamado *Electronic Codebook Mode* (ECB), desde el punto de vista de la dinámica no lineal estos pueden ser considerados como mapas no lineales estáticos. El proceso de cifrado por flujo utilizaría sistemas dinámicos.

Los algoritmos de cifrado basados en caos usan un bloque de texto en claro como condición inicial y/o parámetros del mapa caótico, dado que los mapas caóticos no son invertibles es necesario realizar una discretización de los mismos. El ejemplo más simple de este hecho es el mapa Baker unidimensional *B*: $I = [0,1) \rightarrow I$

$$B(x) = 2 \cdot x - \lfloor 2 \cdot x \rfloor$$

La discretización de este a $I = [0,2N)$ mantiene el mapa invertible como sigue:

$$B_d(x) = \begin{cases} 2x & 0 \le x < N \\ 2x - (2N-1) & N \le x < 2N \end{cases}$$

En sistemas basados en caos se combinan operaciones discretas a fin de hacer invertibles los mapas, al utilizar funciones con la operación binaria XOR o suma modular de la siguiente forma: $c(n) = \mathrm{mod}(p(n) + k(n))$.

Una ventaja del también llamado sistema inverso es que un generador de llave de flujo basado en caos puede ser usado el cual no necesita ser invertible y que puede ser diseñado a partir de las características de una distribución uniforme.

"Usar un mapa caótico adecuado que conserve importantes propiedades durante la discretización (para cifrados en bloque) o una combinación equilibrada de funciones junto con un generador de llaves de flujo (para cifrados por flujo)".

3.4.2 Espacio de llaves

Para evitar una búsqueda exitosa sobre el espacio de llaves utilizado por un algoritmo de cifrado basado en caos:

"El espacio de llaves a ser utilizado para algoritmos de cifrado basado en caos debe ser grande".

Para el caso específico de cifrado de imágenes usualmente se utilizan los valores RGB de cada uno de los pixeles como un solo valor en lugar de utilizar cada uno de los bytes. Esto permite utilizar un mayor número de bits para cada parámetro.

Otras consideraciones a tomar en cuenta con respecto al espacio de llaves son:

- *"No usar una condición inicial del sistema inverso como parte de la llave".*
- *"Usar una transformación compleja de la llave de entrada".*

- *"Usar el mismo tamaño para las subllaves (ya sean definidas o calculadas) como para los parámetros del sistema".*

3.4.3 No lineabilidad y dinámica

En muchos casos los sistemas de cifrados basados en caos utilizan mapas caóticos como son el mapa logístico y los mapas Markov. Una razón para su utilización es que sus propiedades estadísticas pueden ser calculadas o bien diseñadas. Por lo que se recomienda para dichos algoritmos las siguientes características:

- *"Usar un sistema dinámico".*
- *"Usar no lineabilidad compleja".*
- *"Modificar la no lineabilidad dependiendo de la llave y del mensaje en turno".*
- *"Aplicar varias rondas de la operación para cifrados por bloque".*

4. Función Hash Caótica Lúthien - Tinúviel

"... y allí, en el bosque de Neldoreth, nació Lúthien, y las blancas flores de niphrendil se adelantaron para saludarla como estrellas de la tierra…"

"Las hojas eran largas, la hierba era verde, las umbelas de los abetos altas y hermosas y en el claro se vio una luz de estrellas en la sombra centelleante. Tinúviel bailaba allí, a la música de una flauta invisible, con una luz de estrellas en los cabellos y en las vestiduras brillantes..."

J. R. R. Tolkien

Por definición una función hash calcula *hash value* diferentes si el mensaje o bloque de datos sobre el que actúa cambia, para que dicho resultado sea un indicador de la integridad del mismo. La función hash debe ser irreversible y generar un mismo resultado para un mismo mensaje.

Empero al utilizar elementos propios de la teoría del caos para el diseño de una función hash se hace necesario recalcar esta característica, ya que se hace uso de la discretización de una función dinámica en la que interviene el grado de precisión de los parámetros, las condiciones iniciales y los elementos que forman la órbita (puesto que está limitada al grado de precisión soportada en la arquitectura)

Debido a la naturaleza de las funciones que se propondrán se hace necesario incluir las reglas de diseño de una función criptográfica basada en caos, a fin de lograr que las funciones sean irreversibles, no lineales, con comportamiento determinístico y con un grado de seguridad aceptable.

En esta sección se propondrá la definición de una función hash basada en Whirlpool [9] la cual contendrá ciertas características de la teoría del caos específicamente del mapa logístico bidimensional de Baker y de atractor de Lorenz, a fin de que cumpla con lo anteriormente expuesto.

Esta tendrá entre sus propiedades la *no lineabilidad* y al menos las siguientes fases en su diseño:

- Preprocesamiento (expansión) del mensaje.
- Procesamiento (compresión).
- Finalización (transformación de salida).

Pudiendo agregar algunas otras durante la definición las cuales serán indicadas en su momento y ubicadas dentro del esquema básico.

4.1 Antecedentes

La utilización de mapas caóticos en criptografía se ha enfocado principalmente en operaciones de cifrado de imágenes -como se vio en el capítulo anterior- empero esto no ha limitado su utilización con respecto a las funciones hash como veremos a continuación.

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 ***CHA-1*** [10] tiene como principales características:

- **Longitud de bloque** 160 bits.
- **Longitud de *hash value*** 160 bits.
- **Longitud máxima del mensaje** 2^{80} bits.

Puesto que una función hash puede ser vista como un generador de números pseudo aleatorios 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; entre los mapas caóticos más prometedores criptográficamente hablando está el mapa logístico.

Como vimos anteriormente el mapa logístico tiene entre sus propiedades ser simple, rápido, sensible a las condiciones iniciales e impredecibilidad, las cuales podemos transpolar 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 $n \in N$.

4.1.1 Chaos Hash Algorithm-1 (CHA-1)

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

- *Primera fase*: Utiliza la función ***R*** para producir una salida de 160 bits que será la entrada X_0 del mapa logístico en la fase 3.
- *Segunda fase*: La función X_0 es utilizada para producir el valor ***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 ***R*** y X_0 son usados como parámetros de entrada del mapa logístico. *CHA-1* toma los primeros 64 bits del resultado de la función ***R*** para ser agregados al valor de ***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.

De forma gráfica podemos definir a la función *CHA-1* como:

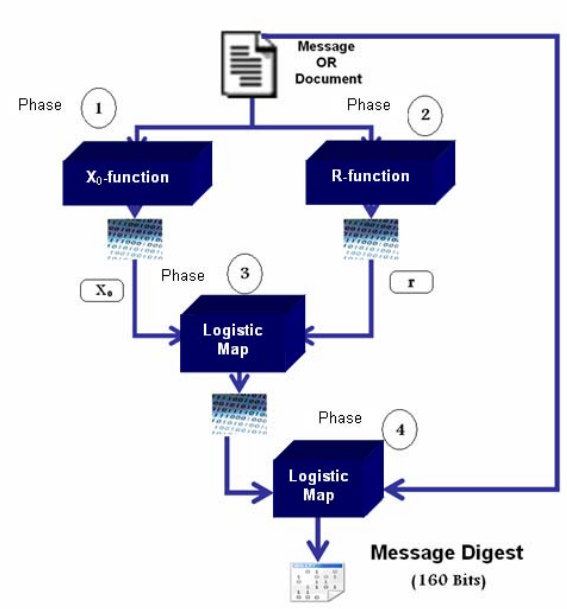


Figura 18. Definición de CHA-1

4.1.2 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 que es el tamaño de bloque de la función hash, dicho bloque de entrada a su vez es dividido en 5 de 32 bits para formar el buffer *ABCDE* que será procesado por la función $\boldsymbol{X_0}$.

Produciendo una salida de 160 bits a la que se le aplica la operación *XOR* con el siguiente bloque de mensaje representado como *A´, B´, C´, D´, E´*. El resultado será utilizado como entrada a la siguiente iteración de la función $\boldsymbol{X_0}$. Este proceso continúa hasta que la función termine de procesar el mensaje y 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.

Gráficamente la definición corresponde a:

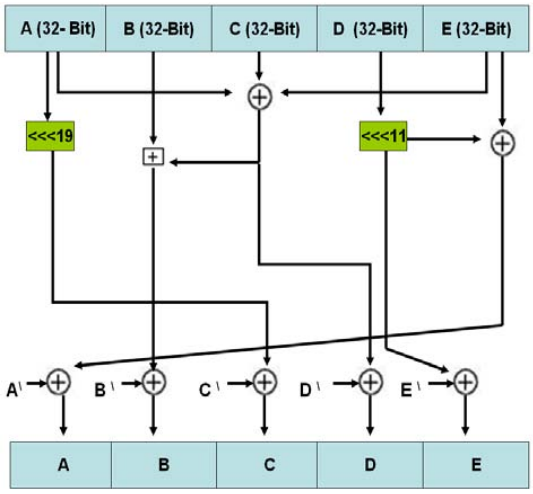


Figura 19. Función X_0 de *CHA-1*

4.1.3 Función R

La definición de la función ***R*** utilizada en *CHA-1* es muy similar a la función X_0 ya que procesa bloques de 160 bits del mensaje de entrada cada vez y produce una salida de la misma longitud. Dicha salida será sumada *XOR* con los siguientes 160 bits de la entrada original. El resultado de la operación previa es tomado como entrada de la función ***R*** en la siguiente iteración. Después de que todos los bloques del mensaje sean procesados el resultado será un bloque de 160 bits, este valor será utilizado como el valor incremental ***r*** en el mapa logístico definido en la fase 3. Este proceso es repetido hasta terminar el mensaje. La función ***R*** es una función simple que usa tres operaciones no conmutativas similares a la función X_0.

Los primeros 64 bits del bloque de salida de la función ***R*** son convertidos a un número real de 160 bits entre 0 y 1. Que usaremos en la fase 3 como el valor incremental ***r*** del mapa logístico.

Gráficamente se puede definir a la función ***R*** como:

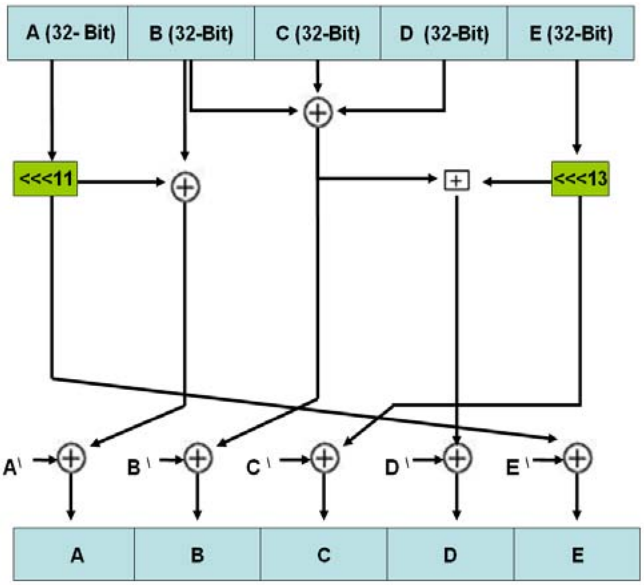


Figura 20. Función *R* de *CHA-1*

4.1.4 Finalización

Después de aplicar el mapa logístico en la fase 3 tenemos un valor de 160 bits como salida, este valor será usado como valor inicial para el mapa logístico de la fase 4. En la fase 4 tomamos 64 bits del mensaje original (cada vez) para ser usados como el valor incremental ***r***. Este proceso es repetido con valores de 64 bits hasta que el mensaje sea procesado 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.

Gráficamente el proceso de generación del valor hash de un mensaje utilizando *CHA-1* queda como sigue:

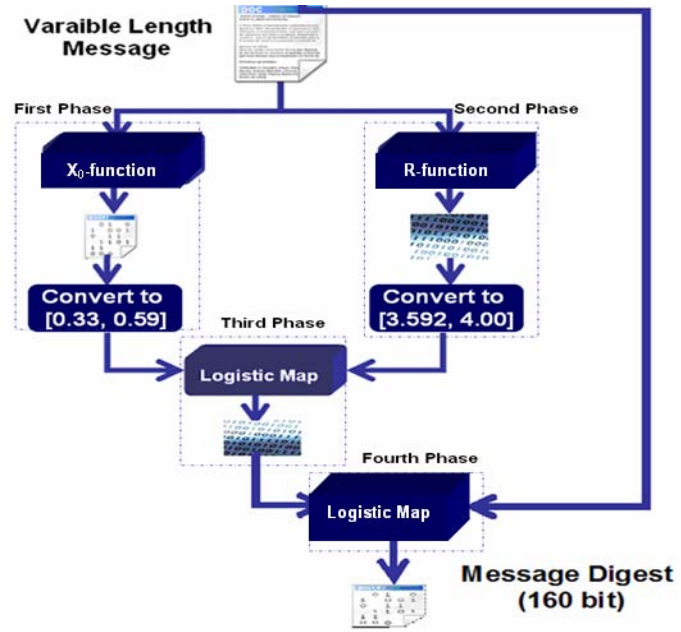


Figura 21. Función *CHA-1*

4.2 Planteamiento de la función Lúthien - Tinúviel Chaos Hash

El algoritmo que aquí se propone seguirá lo indicado por R. Merkle y Damgård [4] (pp. 32) para la fase de expansión del preprocesamiento. Durante el desarrollo de este capítulo se establecerán las características particulares de cada una de las propuestas a fin de diferenciarlas entre ellas y cuando alguna de estas sea común se indicará.

De manera general el algoritmo propuesto tendrá como principales características:

- **Longitud de bloque** 512 bits
- **Longitud de *hash value*** 512 bits.
- **Número de rondas** 12
- **Nombre de la función hash** Lúthien-Tinúviel Chaos Hash (***LT-CHA***)

4.3 Expansión del Mensaje.

Esta fase es obligatoria sin importar que la longitud del mensaje de entrada ya sea un múltiplo entero de la longitud del bloque establecido como 512. El procedimiento a seguir es agregar bits al final del mensaje hasta que la longitud final de este sea un múltiplo entero del tamaño del bloque de la función hash para el caso de que este sea de 512 se seguirá el procedimiento siguiente:

- Se agrega un 1-bit al final del mensaje M.
- Se le agrega al resultado anterior ***r*** 0-bits ($0 \leq r \leq 511$) de tal forma que la longitud de la cadena resultante sea un múltiplo impar de 256. Es decir: $Mlen + 1 + r \equiv 256 \bmod 512$.
- Al final se concatena la representación de *Mlen* de 256 bits con los bits más significativos primero.

Se divide el mensaje resultante en bloques de 512 bits. Con lo que se tendrían *t* bloques de 512 bits ($Mlen + 1 + r + 256 = 512t$) $M_0, M_1, ..., M_{t-1}$.

Como se observa la definición de esta fase es igual a la de la función hash Whirlpool [9] mostrada anteriormente.

4.4 Definición de Parámetros.

Uno de los parámetros importantes dentro del diseño de una función hash es la definición del conjunto de constantes que fungen como sal durante el procesamiento del mensaje. La forma de establecer este conjunto puede ser de dos formas:

- Con base en unas constantes fijas que serán utilizadas de manera repetitiva durante el procesamiento.
- En la definición de un procedimiento de generación en la que las constantes son variables entre cada iteración.

Para nuestro diseño optaremos por la segunda y se definirán un procedimiento de generación basado en dos mapas logísticos.

Así mismo se definen funciones genéricas que serán utilizadas para el cálculo del *hash value* o para la generación de constantes.

El bloque de datos para ***LT-CHA*** (Lúthien-Tinúviel Chaos Hash) consiste de 512 bits lo cual corresponde a un bloque de 64 bytes a semejanza al *state* definido por Whirlpool [9] los cuales pueden ser representados de forma gráfica para facilitar la definición de las funciones que lo utilicen de la siguiente manera:

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

Figura 22. Definición del bloque LT-CHA

4.4.1 Permutación del Bloque LT-CHA

Se definen dos permutaciones una basada en el mapeo bidimensional Baker [11], la cual sólo es usada en la generación de constantes y otra sobre el bloque ***LT-CHA***, como siguen:

El mapa Baker esta descrito con las siguientes expresiones:

$B(x, y) = (2x, y/2)$ donde $0 \leq x < ½$

$B(x, y) = (2x - 1, y/2 + 1/2)$ donde $½ \leq x \leq 1$

El mapa Baker es 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).

Una manera gráfica de ver dicho mapeo es la siguiente:

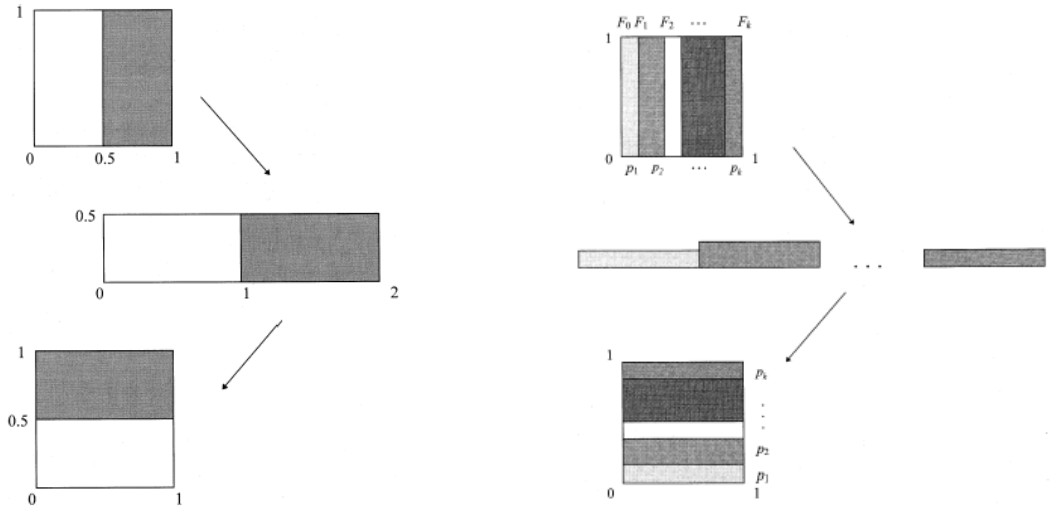


Figura 23. El mapa Baker y su generalización

Basados en la generalización del mapa Baker definiremos a la permutación que utilizaremos para generar las constantes de ronda como:

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

Figura 24. Permutación del mapa Baker.

La cual corresponde a una de las permutaciones del mapa Baker para una imagen de 8 pixeles cuando los enteros *n* no son divisores de 8 para el caso de nuestra definición utilizaremos 3 y 5.

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

$$\begin{vmatrix} P_{0,0} & P_{0,1} & P_{0,2} & P_{0,3} & P_{0,4} & P_{0,5} & P_{0,6} & P_{0,7} \\ P_{1.0} & P_{1,1} & P_{1,2} & P_{1,3} & P_{1,4} & P_{1,5} & P_{1,6} & P_{1,7} \\ P_{2,0} & P_{2,1} & P_{2,2} & P_{2,3} & P_{2,4} & P_{2,5} & P_{2,6} & P_{2,7} \\ P_{3,0} & P_{3,1} & P_{3,2} & P_{3,3} & P_{3,4} & P_{3,5} & P_{3,6} & P_{3,7} \\ P_{4,0} & P_{4,1} & P_{4,2} & P_{4,3} & P_{4,4} & P_{4,5} & P_{4,6} & P_{4,7} \\ P_{5,0} & P_{5,1} & P_{5,2} & P_{5,3} & P_{5,4} & P_{5,5} & P_{5,6} & P_{5,7} \\ P_{6,0} & P_{6,1} & P_{6,2} & P_{6,3} & P_{6,4} & P_{6,5} & P_{6,6} & P_{6,7} \\ P_{7,0} & P_{7,0} & P_{7,2} & P_{7,3} & P_{7,4} & P_{7,5} & P_{7,6} & P_{7,7} \end{vmatrix} \rightarrow \begin{vmatrix} P_{0,7} & P_{1,5} & P_{0,5} & P_{2,6} & P_{1,6} & P_{0,6} & P_{2,7} & P_{1,7} \\ P_{3,7} & P_{4,5} & P_{3,5} & P_{2,5} & P_{4,6} & P_{3,6} & P_{5,7} & P_{4,7} \\ P_{6,7} & P_{7,5} & P_{6,5} & P_{5,5} & P_{7,6} & P_{6,6} & P_{5,6} & P_{7,7} \\ P_{0,4} & P_{1,0} & P_{0,0} & P_{1.1} & P_{0,1} & P_{1,2} & P_{0,2} & P_{0,3} \\ P_{1,4} & P_{3,0} & P_{2,0} & P_{2,1} & P_{2,2} & P_{2,3} & P_{1,3} & P_{2,4} \\ P_{3,4} & P_{4,1} & P_{3,1} & P_{4,2} & P_{3,2} & P_{4,3} & P_{3,3} & P_{4,4} \\ P_{5,4} & P_{5,0} & P_{4,0} & P_{5,1} & P_{5,2} & P_{6,3} & P_{5,3} & P_{6,4} \\ P_{7,4} & P_{7,0} & P_{6,0} & P_{7,1} & P_{6,1} & P_{7,2} & P_{6,2} & P_{7,3} \end{vmatrix}$$

Figura 25. Permutación para la generación de constantes de LT-CHA.

A esta permutación la denominaremos $\Pi(P_i)$.

Definiremos una permutación adicional la cual será aplicada sólo al bloque ***LT-CHA*** como parte de la obtención del hash value durante cada una de las rondas. Dicha permutación se define a continuación:

$$\begin{vmatrix} P_{0,0} & P_{0,1} & P_{0,2} & P_{0,3} & P_{0,4} & P_{0,5} & P_{0,6} & P_{0,7} \\ P_{1.0} & P_{1,1} & P_{1,2} & P_{1,3} & P_{1,4} & P_{1,5} & P_{1,6} & P_{1,7} \\ P_{2,0} & P_{2,1} & P_{2,2} & P_{2,3} & P_{2,4} & P_{2,5} & P_{2,6} & P_{2,7} \\ P_{3,0} & P_{3,1} & P_{3,2} & P_{3,3} & P_{3,4} & P_{3,5} & P_{3,6} & P_{3,7} \\ P_{4,0} & P_{4,1} & P_{4,2} & P_{4,3} & P_{4,4} & P_{4,5} & P_{4,6} & P_{4,7} \\ P_{5,0} & P_{5,1} & P_{5,2} & P_{5,3} & P_{5,4} & P_{5,5} & P_{5,6} & P_{5,7} \\ P_{6,0} & P_{6,1} & P_{6,2} & P_{6,3} & P_{6,4} & P_{6,5} & P_{6,6} & P_{6,7} \\ P_{7,0} & P_{7,0} & P_{7,2} & P_{7,3} & P_{7,4} & P_{7,5} & P_{7,6} & P_{7,7} \end{vmatrix} \rightarrow \begin{vmatrix} P_{2,0} & P_{1,1} & P_{0,2} & P_{7,3} & P_{6,4} & P_{5,5} & P_{4,6} & P_{3,7} \\ P_{3,0} & P_{2,1} & P_{1,2} & P_{0,3} & P_{7,4} & P_{6,5} & P_{5,6} & P_{4,7} \\ P_{4,0} & P_{3,1} & P_{2,2} & P_{1,3} & P_{0,4} & P_{7,5} & P_{6,6} & P_{5,7} \\ P_{5,0} & P_{4,1} & P_{3,2} & P_{2,3} & P_{1,4} & P_{0,5} & P_{7,6} & P_{6,7} \\ P_{6,0} & P_{5,1} & P_{4,2} & P_{3,3} & P_{2,4} & P_{1,5} & P_{0,6} & P_{7,7} \\ P_{7,0} & P_{6,1} & P_{5,2} & P_{4,3} & P_{3,4} & P_{2,5} & P_{1,6} & P_{0,7} \\ P_{0,0} & P_{7,1} & P_{6,2} & P_{5,3} & P_{4,4} & P_{3,5} & P_{2,6} & P_{1,7} \\ P_{1,0} & P_{0,1} & P_{7,2} & P_{6,3} & P_{5,4} & P_{4,5} & P_{3,6} & P_{2,7} \end{vmatrix}$$

Figura 26. Permutación definida para el bloque LT-CHA.

4.4.2 Sustitución de Bytes del Bloque LT-CHA

En esta sección cada byte del bloque ***LT-CHA*** es sustituido por otro, cuyo valor es el indicado en la siguiente tabla, teniendo como entrada un byte formado por los nibbles (grupos de 4 bits) *XY*:

		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

Tabla 6. S-box definida para la función LT-CHA

El procedimiento para calcular dicha tabla es el siguiente:

- Se calcula el inverso multiplicativo en GF(2^8) del byte correspondiente, teniendo como polinomio irreducible a $x^8 + x^5 + x^3 + x + 1$, con excepción del cero el cual no se define.

  El algoritmo para calcular la inversión en F_{2^m} utilizado está basado en el algoritmo extendido de Euclides el cual se define como sigue [12] (pp. 58):

  Entrada: *Un polinomio binario a ≠ 0 de grado m-1, un polinomio binario f el cual es irreducible en* F_{2^m} .

  Salida: *Un polinomio* a^{-1} *que cumple* $a^{-1} \bmod f$.

  Procedimiento:

  - $u \leftarrow a, v \leftarrow f$
  - $g_1 \leftarrow 1, g_2 \leftarrow 0$
  - *Mientras* $u \neq 1$ *haz*
    - $j \leftarrow \deg(u) - \deg(v)$
    - *Si* $j < 0$ *entonces* $u \leftrightarrow v, g_1 \leftrightarrow g_2, j = -j$
    - $u \leftarrow u + z^j v$
    - $g_1 \leftarrow g_1 + z^j g_2$
  - *Regresa* (g_1)

- Se efectúa la siguiente operación matricial que consiste en el producto del inverso multiplicativo representado como un byte con la matriz *8x8* siguiente, a dicho resultado se le agrega 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 difuminar más los bytes resultantes.

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

Figura 27. Operación matricial para definir la función de substitución de bytes

- A esta operación la denominaremos $\Delta(P_i)$.
- En la tabla siguiente se muestra el valor del inverso multiplicativo definido por el irreducible $x^8 + x^5 + x^3 + x + 1$ en GF(2^8) de cada byte, cabe destacar que el valor del inverso de cero no está definido y que el inverso multiplicativo de 1 es si mismo:

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

Tabla 7. Inversos multiplicativos para el irreducible $x^8 + x^5 + x^3 + x + 1$ en GF(2^8).

4.4.2.1 Código para la Obtención de Inversos Multiplicativos

A continuación se proporcionará la implementación en código C de la forma en que se obtuvieron los inversos multiplicativos.

```
#include <stdio.h>
#include <string.h>
#define NAMEFILE_OUTPUT      "LT_CHA_BOX_5.txt"
static unsigned char BOX[8] = {0xad,0x5b,0xb6,0x6d,0xda,0xb5,0x6b,0xd6};
static unsigned char POL    = 0x96;
/* Con esta función calculamos la multiplicación con la matriz que se
 * definió anteriormente dicha operación corresponde a la multiplicación y
 * suma de la constante 0x96 en GF(2^8)
 */
unsigned char calculaValor(unsigned char input) {
      unsigned char result = 0;
      unsigned char temp = 0;
      unsigned char high = 0xf0, low = 0x0f;
      int i = 0, j = 0, k = 0;
      for(i=0; i<8; i++) {
            j = 0;
            // Realizamos la multiplicación columna vector
            temp = input & BOX[i];
            /* Ya realizado esto lo dividimos en dos nibbles.
             * Utilizando la parte alta de la palabra
             */
            k = temp & high;
            k = k >> 4;
            switch(k) {
                  // Número impar de unos
                  case 1:     case 2:
                  case 4:     case 7:
                  case 8:     case 11:
                  case 13:    case 14:
                        j++;
                        break;
            }
            k = temp & low;   // Utilizando la parte baja de la función
```

```
            switch(k) {
                  // Número impar de unos
                  case 1:     case 2:
                  case 4:     case 7:
                  case 8:     case 11:
                  case 13:    case 14:
                        j++;
                        break;
            }
            if(j == 1)  // Número impar dejamos
                  result = result ^ (1<<i);
      }
      result ^= POL;
      return result;
}
// Esta función define cual es el grado del Polinomio en GF(2^8)
int degreePolinomio(int in) {
      int i = 0;
      for(i=8; i>=0; i--) {
            if(in & (1<<i))
                  return i;
      }
      return 0;
}
int main(int argc, char **argv) {
      int iArray[256];  // Arreglo en el que se almacenarán
      int f = 0x012B;   // Polinomio de LT-CHA x^8+x^5+x^3+x+1 (299 en decimal)
      int i = 0;
      int u = 0, v = 0, g1 = 1, g2 = 0, j = 0, tmp = 0, a = 0;
      int k1 = 0, k2 = 0;
      char *name = NULL;
      unsigned char valor_final;
      FILE *pf = NULL, *pf2 = NULL;
      //
      if(argc > 1)
            name = argv[1];
      else
            name = NAMEFILE_OUTPUT;
```

```
pf = fopen(name, "w");
pf2 = fopen("Inverse.txt", "w");
//
for(i=0; i<256; i++) {
      iArray[i] = 0;          // Inicializamos a cero
}
/* Aquí está la implementación del algoritmo descrito para la
 * obtención del inverso multiplicativo.
 */
for(i=0; i<256; i++) {
      g1 = i;
      if(i>1) {
            u = i;
            v = f;
            g1 = 1;
            g2 = 0;
            while(u != 1) {
                  j = degreePolinomio( u ) - degreePolinomio( v );
                  if( j < 0) {
                        tmp = u;
                        u = v;
                        v = tmp;
                        //
                        tmp = g1;
                        g1 = g2;
                        g2 = tmp;
                        j = -1 * j;
                  }
                  k1 = v<<j;
                  k2 = g2<<j;
                  u = u ^ (v << j);
                  g1 = g1 ^ (g2 << j);
            }
      }
      /* Calculamos el valor que corresponde en la tabla el producto
       * con la matriz y la suma de la constante 0x96 en GF(28)
       */
      valor_final = calculaValor((unsigned char)g1);
```

```
			iArray[valor_final]++;
			// Escribimos al archivo los resultados si este último es
			válido
			if(pf != NULL) {
				// Si terminamos una vuelta damos un \n
				if(i % 16 == 0)
					fprintf(pf, "\n");
				//
				if(i%2 == 0)
					fprintf(pf, "\\u%02X", valor_final);
				else
					fprintf(pf, "%02X", valor_final);
			}
			if(pf2 != NULL) {
				if(i % 16 == 0)
					fprintf(pf2, "\n");
				fprintf(pf2, "%x, ", g1);
			}
		}
		// Con este procedimiento validamos que no existan colisiones en la
		tabla
		if(pf != NULL) {
			fprintf(pf, "\n\n Tabla de validación en colisiones \n\n ");
			for(i=0; i<256; i++) {
				if(i%16 == 0)
					fprintf(pf, "\n");
				fprintf(pf, "%d,", iArray[i]);
			}
			fclose(pf);
		}
		if(pf2 != NULL)
			fclose(pf2);
		printf("\n Terminamos con el Inverse Galois.");
		return 0;
}
```

4.4.3 Difusión Lineal del Bloque LT-CHA

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

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

Figura 28. Matriz de difusión lineal de la función LT-CHA.

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

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

Figura 29. 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:

$$S'_{i,0} = S_{i,0} \oplus (S_{i,1} \cdot 05x) \oplus (S_{i,2} \cdot 09x) \oplus (S_{i,3} \cdot 04x) \oplus (S_{i,4} \cdot 08x) \oplus (S_{i,5} \cdot 02x) \oplus (S_{i,6} \cdot 03x) \oplus S_{i,7}$$

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 $x^8 + x^5 + x^3 + x + 1$.

En la implementación del algoritmo *LT-CHA* se utilizarán tablas en las cuales se definen las operaciones previamente indicadas en GF(2^8). Tomando al byte de entrada como un escalar del mismo y esto se realiza para cada una de los demás tomando en cuenta que solo basta aplicar corrimientos adecuados de los valores para definir las demás operaciones. Las tablas generadas se mostraran a detalle en la siguiente sección identificadas como las variables estáticas C0, C1, C2, C3, C4, C5, C6 y C7.

En la que para cada valor definido en la tabla de sustitución se aplica la operación matricial definida por el vector renglón, cabe destacar que con esto se tiene el siguiente resultado; para cada byte que definamos en GF(2^8) definimos un bloque de 8 bytes que representa la multiplicación de dicho byte por el vector renglón de la matriz A.

4.4.3.1 Código para la Obtención de la Tabla de Sustitución

Aquí se muestra el código con el que se obtienen las tablas que permiten definir tanto las operaciones de sustitución, la multiplicación con la matriz de difusión y la combinación de los renglones.

Como entrada se tiene la Sbox generada por el código anterior y con la cual se obtendrán los valores necesarios para la generación de los valores.

```
package cesaro.luthien.chaos.hash;
public class LuthienChaosHash
{
	// Definimos nuestra caja S-BOX para LT-CHA
	private static final String sbox =
		"\u96FD\u53EF\u041E\u2AE3\uAFE5\uA23C\uB8D9\u2C11" +
		"\u0AEB\u5F79\u8CCA\uB3F3\u81FA\u3118\uCBF0\u25CE" +
```

```
        "\uA8FE\u58CF\u02FF\u614B\u9BD6\uC84D\u74EC\u54D0" +
        "\u1DA3\uA0BD\u3593\uD106\u484A\uD577\u3FB0\uBA5D" +
        "\uF98F\uD2E1\uF1B9\u3AD4\uDC68\u5282\u6D36\u08C2" +
        "\u10EA\uB67A\uC970\u7B91\uE7E6\uABBE\uF785\uC56A" +
        "\u237D\u0CEE\u8DA1\u73AD\u47DD\u647F\u45B5\uAEB1" +
        "\u89F4\u88ED\u37BF\u661F\u3242\uF509\u8020\u0317" +
        "\u2116\u1A62\uC4A6\u2DA4\u552F\u01DA\uC01C\uB77C" +
        "\uC390\u99E8\u8440\u9C67\u6B4E\uC65A\uA96C\uBC2E" +
        "\uA500\uD830\u8641\uE0F8\u49DF\u95FB\u60B2\u6524" +
        "\u5E8B\uDED7\u7844\uF228\u2676\u6F50\u4F6E\u985B" +
        "\u4CE4\u133E\uDBBB\uAA39\u1B57\u0D87\u14AC\u0BD3" +
        "\u0E05\u4383\u9F97\u12F6\u0FC7\u0763\u8A9A\u753B" +
        "\u693D\uA759\uE9C1\u2B38\u4633\u7219\u9E71\u227E" +
        "\uB415\uFCE2\u2792\u29CC\u9D34\uCD8E\u5C51\u5694";
//El irreducible es 0x12BL
private static long[][] C = new long[8][256];
static {
      for (int x = 0; x < 256; x++)
      {
            char c = sbox.charAt(x/2);
            long v1 = ((x & 1) == 0) ? c >>> 8 : c & 0xff;
            long v2 = v1 << 1;            // v1 * 2
            // Calculamos el valor multiplicado por 0x02
            if (v2 >= 0x100L)       // Si es mayor a 256
                  v2 ^= 0x12BL;     // x^8 + x^5 + x^3 + x + 1
            // Calculamos por 0x03 a partir del de 0x02 y del v1
            long v3 = v2 ^ v1;
            // Calculamos por 0x04 a partir del de 0x02
            long v4 = v2 << 1;      // v1 * 4 = v2 * 2
            if (v4 >= 0x100L)
                  v4 ^= 0x12BL;
            // Calculamos por 0x05 a partir del 0x04 y del v1
            long v5 = v4 ^ v1;
            // Calculamos por 0x08 a partir del 0x04
            long v8 = v4 << 1;      // v1 * 8 = v2 * 4 = v4 * 2
            if (v8 >= 0x100L)
                  v8 ^= 0x12BL;
            // Calculamos el valor multiplicado por 0x08 y del v1
```

```
                long v9 = v8 ^ v1;
                /* Construye la tabla
                 * C[0][x] = S[x].[1, 1, 3, 2, 8, 4, 9, 5]: */
                C[0][x] =
                      (v1 << 56) | (v1 << 48) | (v3 << 40) |
                      (v2 << 32) | (v8 << 24) | (v4 << 16) |
                      (v9 <<  8) | (v5      );
                /* Se construyen las tablas restantes
                 * C[t][x] = C[0][x] rotr t
                 */
                for (int t = 1; t < 8; t++) {
                      C[t][x] = (C[t - 1][x] >>> 8) |
                                  ((C[t - 1][x] << 56));
                }
          }
          for (int t = 0; t < 8; t++) {
                System.out.println("static const u64 C" +t+ "[256] = {");
                for (int i = 0; i < 64; i++) {
                      System.out.print("   ");
                      for (int j = 0; j < 4; j++) {
                            String v = Long.toHexString(C[t][4*i + j]);
                            while (v.length() < 16) {
                                  v = "0" + v;
                            }
                            System.out.print(" LL(0x" + v + "),");
                      }
                      System.out.println();
                }
                System.out.println("};");
                System.out.println();
          }
          System.out.println();
    }
    public static void main(String[] args) {
          System.out.println("Creamos las tablas para C++");
    }
}
```

4.4.4 Generación de Constantes a partir del Atractor de Lorenz

Con las funciones anteriormente descritas se realizan las operaciones que operan sobre el bloque *LT-CHA*, empero nos falta definir las constantes de ronda que serán utilizadas.

Como su nombre lo indica su valor no cambia durante el desarrollo del algoritmo y la forma de calcularlas - a partir de la definición del atractor de Lorenz - es la siguiente:

Algoritmo que define las constantes a utilizar del atractor de Lorenz.

Entradas:

- *El punto inicial del atractor de Lorenz: P_0 = <x, y, z> = <1, 1, 1>*
- *Definición de las constantes A = 10, B = 28, C = 8/3.*
- *El incremento de las variables Δ = 0.01*

Salida:

- *Se obtienen 104 constantes de 64 bits agrupadas en 13 grupos de 8.*

Procedimiento:

- *Mientras # constantes < 104 haz*
  - *Para $i = 0$ hasta $i < 10$ haz*

$$x' = a(x - y)$$
$$y' = x(b - z) - y$$
$$z' = xy - cz$$
$$x = x + \Delta x'$$
$$y = y + \Delta y'$$
$$z = z + \Delta z'$$

  - *Suprimimos la parte entera de **x**, **y** e **z**. Y a la parte fraccionaria la multiplicamos por la constante 1000000000L*.

$$x = 1000000000 * \lfloor x \rfloor$$
$$y = 1000000000 * \lfloor y \rfloor$$
$$z = 1000000000 * \lfloor z \rfloor$$

  - *Se pasa 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 son agrupados en bloques de 64 bits siguiendo el siguiente orden* $x_0 \mid y_0 \mid z_0 \mid x_1 \mid y_1 \mid z_1 \mid ... \mid x_n \mid y_n \mid z_n$.

- *Regresa 13 grupos de 8 constantes de 64 bits.*

Con el algoritmo anterior las constantes obtenidas son las siguientes:

```
static const u64 KR[R + 1][8] = {
      {
            LL(0x3b82d8000cd5d290), LL(0x03259bcc31cc9cc0),
            LL(0x15b3f340113b3860), LL(0x1cf35d0030441e40),
            LL(0x2d997580357560c0), LL(0x2fa6f340211c1540),
            LL(0x17f8c0e00db5c6c0), LL(0x3804ae002160aa80),
      },
      {
            LL(0x0ae174701a316800), LL(0x06e449e81b76e1a0),
            LL(0x3089a5c02f39b1c0), LL(0x1f29d44035041380),
            LL(0x24c0150027022380), LL(0x214ab5001cf07f20),
            LL(0x158a620006d50618), LL(0x10ca76000f548190),
      },
      {
            LL(0x095d83f033adf600), LL(0x133cc680289751c0),
            LL(0x1e8597c00b7439c0), LL(0x37dcbfc0120f9fa0),
            LL(0x0b15ed400b5bf030), LL(0x04df67a02d522ec0),
            LL(0x0691bc581bb4da80), LL(0x08003a30069ea0f8),
      },
      {
            LL(0x0b8c13a00a7ccd60), LL(0x22dcb440106abd00),
            LL(0x1bfb5de008232a30), LL(0x236d424006f780b0),
            LL(0x1f5c12c019a300c0), LL(0x2fa6fd80276fd980),
            LL(0x26148f800fb4e990), LL(0x0499ed381020d0a0),
      },
      {
            LL(0x2fe5ad4007e6b3a0), LL(0x1c06ad201e7f55e0),
            LL(0x3a9189800069bbf5), LL(0x155369c033e1d300),
            LL(0x1339c90009d204a0), LL(0x2c9c41802d244f00),
            LL(0x12f54be0314ce040), LL(0x1012b6c0391dd840),
      },
```

```
{
      LL(0x0ef793d0309ac680), LL(0x180872a000d46656),
      LL(0x274894c033df4700), LL(0x2de1e60009cfd210),
      LL(0x39ac67001dcf5840), LL(0x01ae633030085340),
      LL(0x26a2fec01b533520), LL(0x140ebb0015c138c0),
},
{
      LL(0x3353ae4030746140), LL(0x2d2247802ea704c0),
      LL(0x34a60cc014394f40), LL(0x107d32a012f03d60),
      LL(0x1d4415202dde33c0), LL(0x2d16a70029a8a980),
      LL(0x39c255000e047920), LL(0x34a01ac0346180c0),
},
{
      LL(0x05ad14e01ced8720), LL(0x136228e0382030c0),
      LL(0x2920c6c02bdad300), LL(0x0eb43ef00102a550),
      LL(0x2c496d001c15fa40), LL(0x13736ec005f9a578),
      LL(0x0afbff6027e2e000), LL(0x2530ff801e0942e0),
},
{
      LL(0x1cc4678011aa7ce0), LL(0x111edcc0041904b8),
      LL(0x087cf7001fe8b400), LL(0x369e1ec03609d9c0),
      LL(0x1d2c2aa0102e6800), LL(0x05167f900bb7fc90),
      LL(0x1db7ea8001d5ab9c), LL(0x38e1b74020c7f900),
},
{
      LL(0x274fb2800328d314), LL(0x0b1af31011a5c9a0),
      LL(0x2c643ec0050d4970), LL(0x1b5bbc201a32b380),
      LL(0x396b01802e45aa80), LL(0x37edca0026deafc0),
      LL(0x1ccec0201485fea0), LL(0x2415e0401aebde00),
},
{
      LL(0x27b6980036e4df40), LL(0x2061784028d5ca40),
      LL(0x11255a401f81fee0), LL(0x197f2ca02d10b340),
      LL(0x14b05be0370fc900), LL(0x03a5e9b031558500),
      LL(0x10f9c8c02f4d76c0), LL(0x0b69e88000ff230d),
},
{
      LL(0x2036920027f98780), LL(0x01a8edbc23f20c80),
```

```
            LL(0x020fcc4816057760), LL(0x1328f4c03b146340),
            LL(0x2d2fe48029f7adc0), LL(0x28c6c5c02cfe7000),
            LL(0x2e9ac400067ea880), LL(0x2fb2eb8019b71ca0),
      },
      {
            LL(0x317b1b802b77d5c0), LL(0x16ae72601f4aab60),
            LL(0x15d63a400479ef28), LL(0x0e946ac01bba18c0),
            LL(0x39506f402fb80f40), LL(0x2a7e3800089baf10),
            LL(0x30076c4012254e60), LL(0x033882f40653f680),
      },
};
```

Sin embargo, estas sólo representan el valor de incremento o sal que se le agrega al bloque *LT-CHA* correspondiente a la ronda.

Estos valores junto con el bloque *LT-CHA* anterior son entradas para generar las constantes de ronda a utilizar en esta iteración de la siguiente manera:

Algoritmo para el cálculo de las constantes de ronda.

Entradas:

- *El bloque LT-CHA inmediato anterior representará nuestra constante de ronda semilla a la cual denotaremos como K^{r-1}. La constante K^0 es igual a un bloque inicializado a cero al inicio de la función, con cada procesamiento de bloque dicho valor cambia y corresponde a la salida LT-CHA anterior.*
- *Las constantes de ronda generadas por el atractor Lorenz (`const u64 KR[R + 1][8]`).*

Salida:

- *Ocho constantes de 64 bits validas solo para esta ronda.*

Procedimiento:

- *Se obtiene el bloque LT-CHA anterior y se inicializa K^0 = LT-CHA*
- *Se suman en GF(2^8) las constantes obtenidas de Lorenz: $K^0 = K^0 \wedge KR^0$*
- *Para cada una de las rondas se realiza el siguiente procedimiento:*
- *Para r = 1 hasta r = R haz*
    - *L = Substitución de Bytes del bloque LT-CHA (K^{r-1})*

- *L = Permutación Baker del bloque LT-CHA (K^{r-1})*
- *L = Mezcla de renglones del bloque LT-CHA (K^{r-1})*
- *L = Agrega la constante de ronda correspondiente (L, KR^r)*

- *En cada iteración se regresa un bloque de 8 constantes de 64 bits valido solo para esta ronda L.*

4.4.4.1 Código para la Obtención de los Valores de Lorenz.

Aquí se muestra el código escrito en Java que se utiliza para la generación de las constantes de ronda, en este se implementa y muestra la utilización del atractor de Lorenz.

```
package cesaro.luthien.chaos.hash;
public class LuthienChaosHash {
	public static void main(String[] args) {
		int i = 0, j = 0, iTermina = 0;
		float AA = 10, BB = 28;
		float CC = (float)((float)8/(float)3);
		float x = 0, y = 0, z = 0;
		float x1 = 0, y1 = 0, z1 = 0;
		long x2 = 0, y2 = 0, z2 = 0;
		float DELTA = (float)0.01;
		String vx="", vy="", vz="", vtotal = "";
		// Primero con decimales sino funciona probamos con enteros
		x = 1;
		y = 1;
		z = 1;
		// Obteniendo las 104 constantes
		System.out.println("static const u64 KR[R + 1][8] = {");
		for (i = 0, iTermina = 0; iTermina < 104; i++) {
			// Recalculamos los valores y obtenemos el 10 valor
			for (j = 0; j < 10; j++) {
				x1 = (AA * (y - x));
				y1 = ((x * (BB - z)) - y);
				z1 = ((x * y) - (CC * z));
				//
				x = (x + (DELTA * x1));
				y = (y + (DELTA * y1));
```

```
                        z = (z + (DELTA * z1));
                }
                x2 = (long)((x - (int)x) * 1000000000L);
                y2 = (long)((y - (int)y) * 1000000000L);
                z2 = (long)((z - (int)z) * 1000000000L);
                // Acompletamos las constantes
                vx = Long.toHexString(x2 > 0 ? x2 : -1*x2);
                while(vx.length() < 8) {
                        vx = "0" + vx;
                }
                vy = Long.toHexString(y2 > 0 ? y2 : -1*y2);
                while(vy.length() < 8) {
                        vy = "0" + vy;
                }
                vz = Long.toHexString(z2 > 0 ? z2 : -1*z2);
                while(vz.length() < 8) {
                        vz = "0" + vz;
                }
                vtotal = vtotal + vx + vy + vz;
                while(vtotal.length() >= 16 && iTermina < 104) {
                        if(iTermina % 8 == 0)
                                System.out.print("    {\n\t\t");
                        iTermina++;
                        System.out.print("LL(0x"+vtotal.substring(0, 16) +
                        "), ");
                        vtotal = vtotal.substring(16, vtotal.length());
                        if(iTermina % 8 == 0)
                                System.out.println("\n    },");
                        else if(iTermina % 4 == 0)
                                System.out.print("\n\t\t");
                }
        }
        System.out.println("};");
        System.out.println("Creamos las tablas para C++.");
    }
}
```

Con este algoritmo se obtienen los valores que definen cada una de las constantes de ronda que se utilizarán para la implementación de *LT-CHA* en C. Dichos valores fueron los anteriormente mostrados.

4.5 Procesamiento

Con los planteamientos descritos anteriormente solo resta definir propiamente el algoritmo *LT-CHA* de la siguiente manera:

Para procesar cada *M(i)* (bloque i^{th} de 512 bits) es necesario realizar los siguientes pasos:

INICIO
Inicializamos a $H^0 = K^0 = 0L$
/* Calculamos el hash value. Procesamos los t bloques de la entrada original*/
PARA i = 1 HASTA *t* HAZ
INICIO
/* Asignamos al bloque LT-CHA representado por *S* los 64 bytes correspondientes al bloque *i*. Mas el valor del hash anterior.*/
S = M[i] ^ H[i-1]
/* Asignamos la llave inicial a partir de los valores que le corresponden. */
K[0] = H[i-1] ^ KR[0][0..7]
/* Calculamos el hash según el siguiente procedimiento. */
PARA r = 1 HASTA R HAZ
INICIA
/* Calculamos las llaves de ronda que se utilizarán */
L = Substitucion 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 = Agrega las constantes de ronda correspondientes (K^{r-1}, KR^r)
/* Obtuvimos la constante de ronda */
K^r = L
/* Con las constantes obtenidas procesamos el bloque LT-CHA */
S = Substitucion de Bytes del bloque LT-CHA (S)
S = *Permutación del bloque* LT-CHA (S)
S = Difusión Lineal del bloque LT-CHA (S)

```
            S = Agrega la constante de ronda correspondiente (S, K^r)
        FIN PARA
        /* Aplicamos la operación Miyhaguchi-Preneel. */
        H[i+1] = H[i] ⊕ S ⊕ M[i]
    FIN PARA
FIN PARA
```

4.6 Finalización

El *hash value* corresponde al valor H_t después de procesar en su totalidad el mensaje quedando de la siguiente manera:

$$LT - CHA(M) = H_t$$

5. Implementación de la función LT-CHA

"La llamo en su corazón Tinúviel, que significa Ruiseñor, hija del crepúsculo, en la lengua de los Elfos Grises, pues no conocía otro nombre para ella. Y la vio a lo lejos como las hojas en los vientos de otoño, y en invierno como una estrella sobre la colina, pero una cadena le aprisionaba los miembros…"

"Mientras Beren la miraba a los ojos entre las sombras de los cabellos vio brillar allí en un espejo la luz temblorosa de las estrellas. Tinúviel la belleza élfica, doncella inmortal de sabiduría élfica lo envolvió con una sombría cabellera y brazos de plata resplandeciente…"

J. R. R. Tolkien

En este capítulo se mostrará una implementación didáctica de la función que hemos denominado como ***Lúthien-Tinúviel Chaos Hash*** (cuyo nombre está inspirado en el personaje de ficción creado por J. R. R. Tolkien en su libro El Silmarillion).

En esta implementación se conjuntará la salida de los programas que fueron descritos en el capítulo anterior, con el objetivo de se comprenda como se relacionan dichas salidas en la codificación de la función ***Lúthien-Tinúviel Chaos Hash***.

Adicionalmente a los comentarios que se colocan en el código fuente se tendrá una breve descripción de a que parte de la función corresponde según lo descrito en el capítulo anterior.

A fin de que se entienda él PORQUE de la codificación de los programas anteriores se indicará de manera puntual a que salida corresponde dicha sección, en caso de que no haya sido indicada previamente en el capítulo anterior.

5.1 Definición de Constantes

En esta sección mostraremos la codificación específica de las constantes que se utilizarán para la definición del algoritmo Lúthien-Tinúviel Chaos Hash.

Esta sección del código fuente corresponde a las siguientes secciones:

- Sustitución de Bytes del Bloque *LT-CHA*
- Difusión Lineal del Bloque *LT-CHA*
- Generación de Constantes a partir del Atractor de Lorenz.
- Así como la implementación en código respectiva a cada sección.

La salida de dichos programas corresponde a la salida en código C que a continuación se muestra:

```
#include <stdio.h>
#include <string.h>

/** La función Lúthien-Tinúviel Chaos Hash
* ================================================================
* - Matriz de difusión por cir(1, 1, 3, 2, 8, 4, 9, 5).
* ================================================================
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "Tinuviel.h"

/** El número de rondas definidas para la función Lúthien-Tinúviel.  */
#define R 12
/** Las tablas que describen a la función Lúthien-Tinúviel están en formato de BIG-ENDIAN.  */
static const u64 C0[256] = {
LL(0x969691071c0e8a98), LL(0xfdfd2cd13989c474), LL(0x5353f5a6ce679d34), LL(0xefef1af5a9c1462e),
LL(0x04040c0820102414), LL(0x1e1e223cf078ee66), LL(0x2a2a7e547ba85182), LL(0xe3e30eedc9f12a12),
LL(0xafafda75ffea5045), LL(0xe5e504e1f9e91c0c), LL(0xa2a2cd6f97de357c), LL(0x3c3c4478cbf0f7cc),
LL(0xb8b8e35b47b6ff0e), LL(0xd9d940993219ebc0), LL(0x2c2c74584bb0679c), LL(0x1111332288449955),
LL(0x0a0a1e1450285a22), LL(0xebeb16fd89d1623a), LL(0x5f5fe1beae57f108), LL(0x79798bf2b5cfccb6),
LL(0x8c8cbf33cc6640ea), LL(0xcaca75bfaa55609f), LL(0xb3b3fe4d1f9aac29), LL(0xf3f33ecd49b1ba42),
```

LL(0x8181a829a45225d3), LL(0xfafa25df0195fb6f), LL(0x31315362a3c492f5), LL(0x18182830c060d878), LL(0xcbcb76bda251699a), LL(0xf0f03bcb51bda14d), LL(0x25256f4a039426b1), LL(0xcece79b78a45448b), LL(0xa8a8d37bc7f66f5e), LL(0xfefe29d72185df7b), LL(0x5858e8b0964bce13), LL(0xcfcf7ab582414d8e), LL(0x020206041008120a), LL(0xffff2ad52981d67e), LL(0x6161a3c275af14ce), LL(0x4b4bdd960e07454c), LL(0x9b9b861d743aefa1), LL(0xd6d651874a259cf3), LL(0xc8c873bbba5d7295), LL(0x4d4dd79a3e1f7352), LL(0x74749ce8ddfba98f), LL(0xecec1ff3b1cd5d21), LL(0x5454fca8f67ba22f), LL(0xd0d05b8b7a3daaed), LL(0x1d1d273ae874f569), LL(0xa3a3ce6d9fda3c79), LL(0xa0a0cb6b87d62776), LL(0xbdbdec516fa2d21f), LL(0x35355f6a83d4b6e1), LL(0x93939e0d341aa789), LL(0xd1d158897239a3e8), LL(0x06060a0c3018361e), LL(0x4848d890160b5e43), LL(0x4a4ade9406034c49), LL(0xd5d55481522987fc), LL(0x777799eec5f7b280), LL(0x3f3f417ed3fcecc3), LL(0xb0b0fb4b0796b726), LL(0xbabae55f57beed04), LL(0x5d5de7babe5fe302), LL(0xf9f920d91999e060), LL(0x8f8fba35d46a5be5), LL(0xd2d25d8f6a35b8e7), LL(0xe1e108e9d9f93818), LL(0xf1f138c959b9a848), LL(0xb9b9e0594fb2f60b), LL(0x3a3a4e74fbe8c1d2), LL(0xd4d457835a2d8ef9), LL(0xdcdc4f931a0dc6d1), LL(0x6868b8d03d8b55e3), LL(0x5252f6a4c6639431), LL(0x8282ad2fbc5e3edc), LL(0x6d6db7da159f78f2), LL(0x36365a6c9bd8adee), LL(0x0808181040204828), LL(0xc2c26dafea7528b7), LL(0x1010302080409050), LL(0xeaea15ff81d56b3f), LL(0xb6b6f147378e8138), LL(0x7a7a8ef4adc3d7b9), LL(0xc9c970b9b2597b90), LL(0x707090e0fdeb8d9b), LL(0x7b7b8df6a5c7debc), LL(0x919198092412b583), LL(0xe7e702e5e9e10e06), LL(0xe6e601e7e1e50703), LL(0xababd67ddffa7451), LL(0xbebee95777aec910), LL(0xf7f732c569a19e56), LL(0x8585a421844201c7), LL(0xc5c564a1d26917ac), LL(0x6a6abed42d8347e9), LL(0x23236546338c10af), LL(0x7d7d87fa95dfe8a2), LL(0x0c0c141860306c3c), LL(0xeeee19f7a1c54f2b), LL(0x8d8dbc31c46249ef), LL(0xa1a1c8698fd22e73), LL(0x737395e6e5e79694), LL(0xadaddc71efe2424f), LL(0x4747c98e6e372970), LL(0xdddd4c911209cfd4), LL(0x6464acc85dbb39df), LL(0x7f7f81fe85d7faa8), LL(0x4545cf8a7e3f3b7a), LL(0xb5b5f4412f829a37), LL(0xaeaed977f7ee5940), LL(0xb1b1f8490f92be23), LL(0x8989b039e4726dfb), LL(0xf4f437c371ad8559), LL(0x8888b33bec7664fe), LL(0xeded1cf1b9c95424), LL(0x3737596e93dca4eb), LL(0xbfbfea557faac015), LL(0x6666aacc4db32bd5), LL(0x1f1f213ef87ce763), LL(0x32325664bbc889fa), LL(0x4242c68446230461), LL(0xf5f534c179a98c5c), LL(0x09091b124824412d), LL(0x8080ab2bac562cd6), LL(0x202060402b800ba0), LL(0x03030506180c1b0f), LL(0x1717392eb85caf4b), LL(0x21216342238402a5), LL(0x16163a2cb058a64e), LL(0x1a1a2e34d068ca72), LL(0x6262a6c46da30fc1), LL(0xc4c467a3da6d1ea9), LL(0xa6a6c167b7ce1168), LL(0x2d2d775a43b46e99), LL(0xa4a4c763a7c60362), LL(0x5555ffaafe7fab2a), LL(0x2f2f715e53bc7c93), LL(0x0101030208040905), LL(0xdada459f2a15f0cf), LL(0xc0c06babfa7d3abd), LL(0x1c1c2438e070fc6c), LL(0xb7b7f2453f8a883d), LL(0x7c7c84f89ddbe1a7), LL(0xc3c36eade27121b2), LL(0x90909b0b2c16bc86), LL(0x999980196432fdab), LL(0xe8e813fb91dd7935), LL(0x8484a7238c4608c2), LL(0x4040c080562b166b), LL(0x9c9c8f134c26d0ba), LL(0x6767a9ce45b722d0), LL(0x6b6bbdd625874eec), LL(0x4e4ed29c2613685d), LL(0xc6c661a7ca650ca3), LL(0x5a5aeeb48643dc19), LL(0xa9a9d079cff2665b), LL(0x6c6cb4d81d9b71f7), LL(0xbcbcef5367a6db1a), LL(0x2e2e725c5bb87596), LL(0xa5a5c461afc20a67), LL(0x0000000000000000), LL(0xd8d8439b3a1de2c5), LL(0x30305060abc09bf0), LL(0x8686a1279c4e1ac8), LL(0x4141c3825e2f1f6e), LL(0xe0e00bebd1fd311d), LL(0xf8f823db119de965), LL(0x4949db921e0f5746), LL(0xdfdf4a950201ddde), LL(0x9595940104029197), LL(0xfbfb26dd0991f26a), LL(0x6060a0c07dab1dcb), LL(0xb2b2fd4f179ea52c), LL(0x6565afca55bf30da), LL(0x24246c480b902fb4), LL(0x5e5ee2bca653f80d), LL(0x8b8bb63df47a7ff1), LL(0xdede49970a05d4db), LL(0xd7d75285422195f6), LL(0x787888f0bdcbc5b3), LL(0x4444cc88763b327f), LL(0xf2f23dcf41b5b347), LL(0x282878506ba04388), LL(0x26266a4c1b983dbe), LL(0x76769aeccdf3bb85), LL(0x6f6fb1de05976af8), LL(0x5050f0a0d66b863b), LL(0x4f4fd19e2e176158), LL(0x6e6eb2dc0d9363fd), LL(0x9898831b6c36f4ae), LL(0x5b5bedb68e47d51c),

```
LL(0x4c4cd498361b7a57), LL(0xe4e407e3f1ed1509), LL(0x13133526984c8b5f), LL(0x3e3e427cdbf8e5c6),
LL(0xdbdb469d2211f9ca), LL(0xbbbbe65d5fbae401), LL(0xaaaad57fd7fe7d54), LL(0x39394b72e3e4dadd),
LL(0x1b1b2d36d86cc377), LL(0x5757f9aeee77b920), LL(0x0d0d171a68346539), LL(0x8787a225944a13cd),
LL(0x14143c28a050b444), LL(0xacacdf73e7e64b4a), LL(0x0b0b1d16582c5327), LL(0xd3d35e8d6231b1e2),
LL(0x0e0e121c70387e36), LL(0x05050f0a28142d11), LL(0x4343c5864e270d64), LL(0x8383ae2db45a37d9),
LL(0x9f9f8a15542acbb5), LL(0x97979205140a839d), LL(0x121236249048825a), LL(0xf6f631c761a59753),
LL(0x0f0f111e783c7733), LL(0xc7c762a5c26105a6), LL(0x0707090e381c3f1b), LL(0x6363a5c665a706c4),
LL(0x8a8ab53ffc7e76f4), LL(0x9a9a851f7c3ee6a4), LL(0x75759fead5ffa08a), LL(0x3b3b4d76f3ecc8d7),
LL(0x6969bbd2358f5ce6), LL(0x3d3d477ac3f4fec9), LL(0xa7a7c265bfca186d), LL(0x5959ebb29e4fc716),
LL(0xe9e910f999d97030), LL(0xc1c168a9f27933b8), LL(0x2b2b7d5673ac5887), LL(0x38384870ebe0d3d8),
LL(0x4646ca8c66332075), LL(0x33335566b3cc80ff), LL(0x727296e4ede39f91), LL(0x19192b32c864d17d),
LL(0x9e9e89175c2ec2b0), LL(0x717193e2f5ef849e), LL(0x222266443b8819aa), LL(0x7e7e82fc8dd3f3ad),
LL(0xb4b4f74327869332), LL(0x15153f2aa854bd41), LL(0xfcfc2fd3318dcd71), LL(0xe2e20defc1f52317),
LL(0x2727694e139c34bb), LL(0x92929d0f3c1eae8c), LL(0x29297b5263a44a8d), LL(0xcccc7fb39a4d5681),
LL(0x9d9d8c114422d9bf), LL(0x34345c688bd0bfe4), LL(0xcdcd7cb192495f84), LL(0x8e8eb937dc6e52e0),
LL(0x5c5ce4b8b65bea07), LL(0x5151f3a2de6f8f3e), LL(0x5656faace673b025), LL(0x949497030c069892),
};
```

Las definiciones de las S-box restantes se encuentran en el material incluido en el CD-ROM en el que se encuentra la implementación completa de la función hash Lúthien – Tinúviel definida en el capítulo anterior.

```
static const u64 C1[256] = { … };
static const u64 C2[256] = { … };
static const u64 C3[256] = { … };
static const u64 C4[256] = { … };
static const u64 C5[256] = { … };
static const u64 C6[256] = { … };
static const u64 C7[256] = { … };

/**  Las 13 constantes de ronda definidas para la función Lúthien-Tinúviel.  */
static const u64 KR[R + 1][8] = {
{   LL(0x3b82d8000cd5d290), LL(0x03259bcc31cc9cc0), LL(0x15b3f340113b3860), LL(0x1cf35d0030441e40),
    LL(0x2d997580357560c0), LL(0x2fa6f340211c1540), LL(0x17f8c0e00db5c6c0), LL(0x3804ae002160aa80),
},
{   LL(0x0ae174701a316800), LL(0x06e449e81b76e1a0), LL(0x3089a5c02f39b1c0), LL(0x1f29d44035041380),
    LL(0x24c0150027022380), LL(0x214ab5001cf07f20), LL(0x158a620006d50618), LL(0x10ca76000f548190),
},
{   LL(0x095d83f033adf600), LL(0x133cc680289751c0), LL(0x1e8597c00b7439c0), LL(0x37dcbfc0120f9fa0),
    LL(0x0b15ed400b5bf030), LL(0x04df67a02d522ec0), LL(0x0691bc581bb4da80), LL(0x08003a30069ea0f8),},
{   LL(0x0b8c13a00a7ccd60), LL(0x22dcb440106abd00), LL(0x1bfb5de008232a30), LL(0x236d424006f780b0),
```

```
    LL(0x1f5c12c019a300c0), LL(0x2fa6fd80276fd980), LL(0x26148f800fb4e990), LL(0x0499ed381020d0a0),
},
{   LL(0x2fe5ad4007e6b3a0), LL(0x1c06ad201e7f55e0), LL(0x3a9189800069bbf5), LL(0x155369c033e1d300),
    LL(0x1339c90009d204a0), LL(0x2c9c41802d244f00), LL(0x12f54be0314ce040), LL(0x1012b6c0391dd840),
},
{   LL(0x0ef793d0309ac680), LL(0x180872a000d46656), LL(0x274894c033df4700), LL(0x2de1e60009cfd210),
    LL(0x39ac67001dcf5840), LL(0x01ae633030085340), LL(0x26a2fec01b533520), LL(0x140ebb0015c138c0),
},
{   LL(0x3353ae4030746140), LL(0x2d2247802ea704c0), LL(0x34a60cc014394f40), LL(0x107d32a012f03d60),
    LL(0x1d4415202dde33c0), LL(0x2d16a70029a8a980), LL(0x39c255000e047920), LL(0x34a01ac0346180c0)
},
{   LL(0x05ad14e01ced8720), LL(0x136228e0382030c0), LL(0x2920c6c02bdad300), LL(0x0eb43ef00102a550),
    LL(0x2c496d001c15fa40), LL(0x13736ec005f9a578), LL(0x0afbff6027e2e000), LL(0x2530ff801e0942e0),
},
{   LL(0x1cc4678011aa7ce0), LL(0x111edcc0041904b8), LL(0x087cf7001fe8b400), LL(0x369e1ec03609d9c0),
    LL(0x1d2c2aa0102e6800), LL(0x05167f900bb7fc90), LL(0x1db7ea8001d5ab9c), LL(0x38e1b74020c7f900),
},
{   LL(0x274fb2800328d314), LL(0x0b1af31011a5c9a0), LL(0x2c643ec0050d4970), LL(0x1b5bbc201a32b380),
    LL(0x396b01802e45aa80), LL(0x37edca0026deafc0), LL(0x1ccec0201485fea0), LL(0x2415e0401aebde00),
},
{   LL(0x27b6980036e4df40), LL(0x2061784028d5ca40), LL(0x11255a401f81fee0), LL(0x197f2ca02d10b340),
    LL(0x14b05be0370fc900), LL(0x03a5e9b031558500), LL(0x10f9c8c02f4d76c0), LL(0x0b69e88000ff230d),
},
{   LL(0x2036920027f98780), LL(0x01a8edbc23f20c80), LL(0x020fcc4816057760), LL(0x1328f4c03b146340),
    LL(0x2d2fe48029f7adc0), LL(0x28c6c5c02cfe7000), LL(0x2e9ac400067ea880), LL(0x2fb2eb8019b71ca0),
},
{   LL(0x317b1b802b77d5c0), LL(0x16ae72601f4aab60), LL(0x15d63a400479ef28), LL(0x0e946ac01bba18c0),
    LL(0x39506f402fb80f40), LL(0x2a7e3800089baf10), LL(0x30076c4012254e60), LL(0x033882f40653f680),
},
};
```

5.2 Definición e Inicialización del Bloque LT-CHA

En esta sección se mostrará la implementación en código de la función *LT-CHA*. La cual será mostrada en sus diferentes secciones según corresponde:

Se define la estructura principal de la función *LT-CHA* así como las constantes que se utilizaran durante la implementación:

Se redefinen los tipos de datos de acuerdo al tipo de compilador que se utilizan para este caso se utiliza el compilador Visual Studio C++ 6.0 y gcc de GNU Linux.

```
typedef signed char s8;
typedef unsigned char u8;
#ifdef _MSC_VER
        typedef unsigned __int64 u64;
        typedef signed __int64 s64;
        #define LL(v)   (v##i64)
        #define ONE64   LL(0xffffffffffffffff)
#else  /* !_MSC_VER */
        typedef unsigned long long u64;
        typedef signed long long s64;
        #define LL(v)   (v##ULL)
        #define ONE64   LL(0xffffffffffffffff)
#endif /* ?_MSC_VER */
```

Definición de las principales constantes a utilizar durante la implementación y de la estructura a utilizar, el bloque *LT-CHA* está representado de forma dual en la estructura *LUTHIENstruct* de la siguiente forma:

- *buffer*: El bloque *LT-CHA* actual que contiene el fragmento del mensaje a procesar. Formado por 64 bloques de 8 bits.
- *hash*: El hash value parcial que representa el procesamiento del bloque *LT-CHA* anterior. Formado por 8 bloques de 64 bits.

```
#define DIGESTBYTES 64
#define DIGESTBITS  (8*DIGESTBYTES)          /* 512 */
#define WBLOCKBYTES 64
#define WBLOCKBITS  (8*WBLOCKBYTES)          /* 512 */
#define LENGTHBYTES 32
#define LENGTHBITS  (8*LENGTHBYTES)          /* 256 */

typedef struct LUTHIENstruct {
        u8  bitLength[LENGTHBYTES];     /* Número de bits del mensaje (256-bit) */
        u8  buffer[WBLOCKBYTES];        /* buffer de datos para obtener el hash */
        int bufferBits;                 /* Número actual de bits sobre el buffer */
        int bufferPos;                  /* current (possibly incomplete) byte slot on the buffer */
        u64 hash[DIGESTBYTES/8];        /* El bloque LT-CHA procesado */
} LUTHIENstruct;
```

Inicialización de los elementos principales que conforman la estructura de control de la función hash.

```
/** Inicializa el estado de la función Lúthien-Tinúviel. */
void initLuthienTinuviel(struct LUTHIENstruct * const structpointer) {
	int i;

	memset(structpointer->bitLength, 0, 32);
	structpointer->bufferBits = structpointer->bufferPos = 0;
	structpointer->buffer[0] = 0;		/* Esto es necesario para limpiar el buffer[bufferPos] */
	for (i = 0; i < 8; i++) {
		structpointer->hash[i] = 0L;		/* Valor inicial */
	}
}
```

5.3 Procesamiento del Bloque LT-CHA

En esta función se define el procesamiento del bloque *LT-CHA* y por consecuencia a la función hash. Como primer instancia se inicializa el buffer con los datos del fragmento del mensaje.

Se inicializa el nuevo conjunto de constantes a ser utilizadas durante cada una de las rondas de la función hash a partir del hash value anterior. Posteriormente en cada una de las rondas se utilizan las constantes calculadas así como las constantes obtenidas a partir del atractor de Lorenz.

Se calcula el nuevo valor hash después de procesarlo por cada una de las rondas y al final de la operación se aplica la función de compresión de Miyaguchi-Preneel.

En esta sección se codifica la permutación del bloque *LT-CHA* que fue definida previamente siguiente el mapa bidimensional Baker modificado. Esta operación se observa en la utilización de la definición de las constantes, tanto para la generación de las llaves como en el procesamiento del bloque *LT-CHA* respectivo.

```
/* La definición funcional de la función hash Lúthien-Tinúviel  */
static void processBuffer(struct LUTHIENstruct * const structpointer) {
        int i, r;
        u64 block[8];                   /* mu(buffer) */
        u64 bloqueLCH[8];               /* el bloque de cifrado */
        u64 L[8];
        u64 K[8];                       /* la llave de ronda */
        u8 *buffer = structpointer->buffer;
        /* Mapea el buffer al bloque LCH */
        for (i = 0; i < 8; i++, buffer += 8) {
                block[i] =
                        (((u64)buffer[0]          ) << 56) ^
                        (((u64)buffer[1] & 0xffL) << 48) ^
                        (((u64)buffer[2] & 0xffL) << 40) ^
                        (((u64)buffer[3] & 0xffL) << 32) ^
                        (((u64)buffer[4] & 0xffL) << 24) ^
                        (((u64)buffer[5] & 0xffL) << 16) ^
                        (((u64)buffer[6] & 0xffL) <<  8 ) ^
                        (((u64)buffer[7] & 0xffL)  );
        }
        /* Aplicamos la K^0 al bloque inicial antes de continuar. Realizamos la siguiente operación:
         * bloqueLCH = bloque ^ hashValueAnterior ^ Llave de ronda                      */
        bloqueLCH[0] = block[0] ^ (K[0] = structpointer->hash[0]);
        bloqueLCH[1] = block[1] ^ (K[1] = structpointer->hash[1]);
        bloqueLCH[2] = block[2] ^ (K[2] = structpointer->hash[2]);
        bloqueLCH[3] = block[3] ^ (K[3] = structpointer->hash[3]);
        bloqueLCH[4] = block[4] ^ (K[4] = structpointer->hash[4]);
        bloqueLCH[5] = block[5] ^ (K[5] = structpointer->hash[5]);
        bloqueLCH[6] = block[6] ^ (K[6] = structpointer->hash[6]);
        bloqueLCH[7] = block[7] ^ (K[7] = structpointer->hash[7]);

        K[0] ^= KR[0][0];
        K[1] ^= KR[0][1];
        K[2] ^= KR[0][2];
        K[3] ^= KR[0][3];
        K[4] ^= KR[0][4];
        K[5] ^= KR[0][5];
        K[6] ^= KR[0][6];
        K[7] ^= KR[0][7];

        for (r = 1; r <= R; r++) {              /*  Itera sobre todas las rondas  */
                /*  Calcula K^r de K^{r-1}:  */
```

```
L[0] =
        C7[(int)(K[0] >> 56)        ] ^
        C5[(int)(K[1] >> 48) & 0xff] ^
        C5[(int)(K[0] >> 40) & 0xff] ^
        C6[(int)(K[2] >> 32) & 0xff] ^
        C6[(int)(K[1] >> 24) & 0xff] ^
        C6[(int)(K[0] >> 16) & 0xff] ^
        C7[(int)(K[2] >>  8) & 0xff] ^
        C7[(int)(K[1]        ) & 0xff] ^
        KR[r][0]; // Aplicamos la llave de ronda respectiva
L[1] =
        C7[(int)(K[3] >> 56)        ] ^
        C5[(int)(K[4] >> 48) & 0xff] ^
        C5[(int)(K[3] >> 40) & 0xff] ^
        C5[(int)(K[2] >> 32) & 0xff] ^
        C6[(int)(K[4] >> 24) & 0xff] ^
        C6[(int)(K[3] >> 16) & 0xff] ^
        C7[(int)(K[5] >>  8) & 0xff] ^
        C7[(int)(K[4]        ) & 0xff] ^
        KR[r][1]; // Aplicamos la llave de ronda respectiva
L[2] =
        C7[(int)(K[6] >> 56)        ] ^
        C5[(int)(K[7] >> 48) & 0xff] ^
        C5[(int)(K[6] >> 40) & 0xff] ^
        C5[(int)(K[5] >> 32) & 0xff] ^
        C6[(int)(K[7] >> 24) & 0xff] ^
        C6[(int)(K[6] >> 16) & 0xff] ^
        C6[(int)(K[5] >>  8) & 0xff] ^
        C7[(int)(K[7]        ) & 0xff] ^
        KR[r][2]; // Aplicamos la llave de ronda respectiva
L[3] =
        C4[(int)(K[0] >> 56)        ] ^
        C0[(int)(K[1] >> 48) & 0xff] ^
        C0[(int)(K[0] >> 40) & 0xff] ^
        C1[(int)(K[1] >> 32) & 0xff] ^
        C1[(int)(K[0] >> 24) & 0xff] ^
        C2[(int)(K[1] >> 16) & 0xff] ^
        C2[(int)(K[0] >>  8) & 0xff] ^
        C3[(int)(K[0]        ) & 0xff] ^
        KR[r][3]; // Aplicamos la llave de ronda respectiva
```

```
L[4] =
        C4[(int)(K[1] >> 56)        ] ^
        C0[(int)(K[3] >> 48) & 0xff] ^
        C0[(int)(K[2] >> 40) & 0xff] ^
        C1[(int)(K[2] >> 32) & 0xff] ^
        C2[(int)(K[2] >> 24) & 0xff] ^
        C3[(int)(K[2] >> 16) & 0xff] ^
        C3[(int)(K[1] >>  8) & 0xff] ^
        C4[(int)(K[2]        ) & 0xff] ^
        KR[r][4]; // Aplicamos la llave de ronda respectiva
L[5] =
        C4[(int)(K[3] >> 56)        ] ^
        C1[(int)(K[4] >> 48) & 0xff] ^
        C1[(int)(K[3] >> 40) & 0xff] ^
        C2[(int)(K[4] >> 32) & 0xff] ^
        C2[(int)(K[3] >> 24) & 0xff] ^
        C3[(int)(K[4] >> 16) & 0xff] ^
        C3[(int)(K[3] >>  8) & 0xff] ^
        C4[(int)(K[4]        ) & 0xff] ^
        KR[r][5]; // Aplicamos la llave de ronda respectiva
L[6] =
        C4[(int)(K[5] >> 56)        ] ^
        C0[(int)(K[5] >> 48) & 0xff] ^
        C0[(int)(K[4] >> 40) & 0xff] ^
        C1[(int)(K[5] >> 32) & 0xff] ^
        C2[(int)(K[5] >> 24) & 0xff] ^
        C3[(int)(K[6] >> 16) & 0xff] ^
        C3[(int)(K[5] >>  8) & 0xff] ^
        C4[(int)(K[6]        ) & 0xff] ^
        KR[r][6]; // Aplicamos la llave de ronda respectiva
L[7] =
        C4[(int)(K[7] >> 56)        ] ^
        C0[(int)(K[7] >> 48) & 0xff] ^
        C0[(int)(K[6] >> 40) & 0xff] ^
        C1[(int)(K[7] >> 32) & 0xff] ^
        C1[(int)(K[6] >> 24) & 0xff] ^
        C2[(int)(K[7] >> 16) & 0xff] ^
        C2[(int)(K[6] >>  8) & 0xff] ^
        C3[(int)(K[7]        ) & 0xff] ^
        KR[r][7]; // Aplicamos la llave de ronda respectiva
```

```
K[0] = L[0];
K[1] = L[1];
K[2] = L[2];
K[3] = L[3];
K[4] = L[4];
K[5] = L[5];
K[6] = L[6];
K[7] = L[7];
/*  Aplica la r-th ronda  */
L[0] =
        C0[(int)(bloqueLCH[2] >> 56)        ] ^
        C1[(int)(bloqueLCH[1] >> 48) & 0xff] ^
        C2[(int)(bloqueLCH[0] >> 40) & 0xff] ^
        C3[(int)(bloqueLCH[7] >> 32) & 0xff] ^
        C4[(int)(bloqueLCH[6] >> 24) & 0xff] ^
        C5[(int)(bloqueLCH[5] >> 16) & 0xff] ^
        C6[(int)(bloqueLCH[4] >>  8) & 0xff] ^
        C7[(int)(bloqueLCH[3]      ) & 0xff] ^
        K[0];
L[1] =
        C0[(int)(bloqueLCH[3] >> 56)        ] ^
        C1[(int)(bloqueLCH[2] >> 48) & 0xff] ^
        C2[(int)(bloqueLCH[1] >> 40) & 0xff] ^
        C3[(int)(bloqueLCH[0] >> 32) & 0xff] ^
        C4[(int)(bloqueLCH[7] >> 24) & 0xff] ^
        C5[(int)(bloqueLCH[6] >> 16) & 0xff] ^
        C6[(int)(bloqueLCH[5] >>  8) & 0xff] ^
        C7[(int)(bloqueLCH[4]      ) & 0xff] ^
        K[1];
L[2] =
        C0[(int)(bloqueLCH[4] >> 56)        ] ^
        C1[(int)(bloqueLCH[3] >> 48) & 0xff] ^
        C2[(int)(bloqueLCH[2] >> 40) & 0xff] ^
        C3[(int)(bloqueLCH[1] >> 32) & 0xff] ^
        C4[(int)(bloqueLCH[0] >> 24) & 0xff] ^
        C5[(int)(bloqueLCH[7] >> 16) & 0xff] ^
        C6[(int)(bloqueLCH[6] >>  8) & 0xff] ^
        C7[(int)(bloqueLCH[5]      ) & 0xff] ^
        K[2];
L[3] =
        C0[(int)(bloqueLCH[5] >> 56)        ] ^
        C1[(int)(bloqueLCH[4] >> 48) & 0xff] ^
```

```
            C2[(int)(bloqueLCH[3] >> 40) & 0xff] ^
            C3[(int)(bloqueLCH[2] >> 32) & 0xff] ^
            C4[(int)(bloqueLCH[1] >> 24) & 0xff] ^
            C5[(int)(bloqueLCH[0] >> 16) & 0xff] ^
            C6[(int)(bloqueLCH[7] >>  8) & 0xff] ^
            C7[(int)(bloqueLCH[6]      ) & 0xff] ^
            K[3];
L[4] =
            C0[(int)(bloqueLCH[6] >> 56)       ] ^
            C1[(int)(bloqueLCH[5] >> 48) & 0xff] ^
            C2[(int)(bloqueLCH[4] >> 40) & 0xff] ^
            C3[(int)(bloqueLCH[3] >> 32) & 0xff] ^
            C4[(int)(bloqueLCH[2] >> 24) & 0xff] ^
            C5[(int)(bloqueLCH[1] >> 16) & 0xff] ^
            C6[(int)(bloqueLCH[0] >>  8) & 0xff] ^
            C7[(int)(bloqueLCH[7]      ) & 0xff] ^
            K[4];
L[5] =
            C0[(int)(bloqueLCH[7] >> 56)       ] ^
            C1[(int)(bloqueLCH[6] >> 48) & 0xff] ^
            C2[(int)(bloqueLCH[5] >> 40) & 0xff] ^
            C3[(int)(bloqueLCH[4] >> 32) & 0xff] ^
            C4[(int)(bloqueLCH[3] >> 24) & 0xff] ^
            C5[(int)(bloqueLCH[2] >> 16) & 0xff] ^
            C6[(int)(bloqueLCH[1] >>  8) & 0xff] ^
            C7[(int)(bloqueLCH[0]      ) & 0xff] ^
            K[5];
L[6] =
            C0[(int)(bloqueLCH[0] >> 56)       ] ^
            C1[(int)(bloqueLCH[7] >> 48) & 0xff] ^
            C2[(int)(bloqueLCH[6] >> 40) & 0xff] ^
            C3[(int)(bloqueLCH[5] >> 32) & 0xff] ^
            C4[(int)(bloqueLCH[4] >> 24) & 0xff] ^
            C5[(int)(bloqueLCH[3] >> 16) & 0xff] ^
            C6[(int)(bloqueLCH[2] >>  8) & 0xff] ^
            C7[(int)(bloqueLCH[1]      ) & 0xff] ^
            K[6];
L[7] =
            C0[(int)(bloqueLCH[1] >> 56)       ] ^
            C1[(int)(bloqueLCH[0] >> 48) & 0xff] ^
            C2[(int)(bloqueLCH[7] >> 40) & 0xff] ^
            C3[(int)(bloqueLCH[6] >> 32) & 0xff] ^
```

```
                    C4[(int)(bloqueLCH[5] >> 24) & 0xff] ^
                    C5[(int)(bloqueLCH[4] >> 16) & 0xff] ^
                    C6[(int)(bloqueLCH[3] >>  8) & 0xff] ^
                    C7[(int)(bloqueLCH[2]      ) & 0xff] ^
                    K[7];
            bloqueLCH[0] = L[0];
            bloqueLCH[1] = L[1];
            bloqueLCH[2] = L[2];
            bloqueLCH[3] = L[3];
            bloqueLCH[4] = L[4];
            bloqueLCH[5] = L[5];
            bloqueLCH[6] = L[6];
            bloqueLCH[7] = L[7];
        }
        /*  Se aplica la función de compresión Miyaguchi-Preneel  */
        structpointer->hash[0] ^= bloqueLCH[0] ^ block[0];
        structpointer->hash[1] ^= bloqueLCH[1] ^ block[1];
        structpointer->hash[2] ^= bloqueLCH[2] ^ block[2];
        structpointer->hash[3] ^= bloqueLCH[3] ^ block[3];
        structpointer->hash[4] ^= bloqueLCH[4] ^ block[4];
        structpointer->hash[5] ^= bloqueLCH[5] ^ block[5];
        structpointer->hash[6] ^= bloqueLCH[6] ^ block[6];
        structpointer->hash[7] ^= bloqueLCH[7] ^ block[7];
}
```

5.4 Finalización de la Función Hash

Esta función es un wrapper de la función *LT-CHA* ya que el objetivo principal de esta es la de dividir el mensaje de entrada en bloque con la longitud válida para ser procesados por la función *LT-CHA*.

```
/** Obtiene el hash value del bloque LT-CHA.
 *  Este método usa el invariante: bufferBits < DIGESTBITS
 */
void finalizeLuthienTinuviel(struct LUTHIENstruct * const structpointer, unsigned char * const result) {
        int i;
        u8 *buffer      = structpointer->buffer;
        u8 *bitLength   = structpointer->bitLength;
        int bufferBits  = structpointer->bufferBits;
```

```
        int bufferPos       = structpointer->bufferPos;
        u8 *digest          = result;

        buffer[bufferPos] |= 0x80U >> (bufferBits & 7);          /*   agregamos un '1'-bit  */
        bufferPos++;                                            /* Los bits restantes se colocan a cero. */
        /*   Rellenamos con cero para completar (N*WBLOCKBITS - LENGTHBITS) bits  */
        if (bufferPos > WBLOCKBYTES - LENGTHBYTES)  {
                if (bufferPos < WBLOCKBYTES) {
                        memset(&buffer[bufferPos], 0, WBLOCKBYTES - bufferPos);
                }
                processBuffer(structpointer);           /*  Procesa el bloque de datos.  */
                bufferPos = 0;                          /*  Reinicia buffer  */
        }
        if (bufferPos < WBLOCKBYTES - LENGTHBYTES) {
                memset(&buffer[bufferPos], 0, (WBLOCKBYTES - LENGTHBYTES) - bufferPos);
        }
        bufferPos = WBLOCKBYTES - LENGTHBYTES;
        /*  Agrega la longitud en bits del mensaje del hash value.  */
        memcpy(&buffer[WBLOCKBYTES - LENGTHBYTES], bitLength, LENGTHBYTES);
        processBuffer(structpointer);                   /*  Procesa el bloque de datos.  */
        /*  Regresa el mensaje completamente procesado.  */
        for (i = 0; i < DIGESTBYTES/8; i++)  {
                digest[0] = (u8)(structpointer->hash[i] >> 56);
                digest[1] = (u8)(structpointer->hash[i] >> 48);
                digest[2] = (u8)(structpointer->hash[i] >> 40);
                digest[3] = (u8)(structpointer->hash[i] >> 32);
                digest[4] = (u8)(structpointer->hash[i] >> 24);
                digest[5] = (u8)(structpointer->hash[i] >> 16);
                digest[6] = (u8)(structpointer->hash[i] >>  8);
                digest[7] = (u8)(structpointer->hash[i]      );
                digest += 8;
        }
        structpointer->bufferBits   = bufferBits;
        structpointer->bufferPos    = bufferPos;
}
```

5.5 Agregar y Procesar un Nuevo Bloque.

Se agrega un nuevo bloque de bits para que sea procesado por la función *LT-CHA*, también tiene como tarea la de realizar el preprocesamiento del mensaje antes descrito y procesar todos los bloque menos uno del mensaje a fin de generar el hash value final.

```
/* Agrega datos de entrada para el algoritmo hash.
 * @param    source         Datos en claro para el hash.
 * @param    sourceBits     Cuantos bits de texto en claro a procesar.
 * Este mantiene los valores invariantes: bufferBits < DIGESTBITS
 */
void addLuthienTinuviel(const unsigned char * const source, unsigned long sourceBits,
                        struct LUTHIENstruct * const structpointer) {
        /*              sourcePos
                        |
                        +-------+-------+-------
                            |||||||||||||||||||| source
                        +-------+-------+-------
         +-------+-------+-------+-------+-------+-------
         |||||||||||||||||||||                   buffer
         +-------+-------+-------+-------+-------+-------
                    |
                    bufferPos
         */
        /* Índice del byte (u8) más a la izquierda que contiene datos (1 a 8 bits). */
        int sourcePos    = 0;
        int sourceGap    = (8 - ((int)sourceBits & 7)) & 7;      /* Espacio sobre el fuente [sourcePos]. */
        int bufferRem    = structpointer->bufferBits & 7;        /* bits ocupados en buffer[bufferPos]. */
        int i;
        u32 b, carry;
        u8 *buffer           = structpointer->buffer;
        u8 *bitLength        = structpointer->bitLength;
        int bufferBits       = structpointer->bufferBits;
        int bufferPos        = structpointer->bufferPos;
        // coincide la longitud de los datos agregados
        u64 value = sourceBits;
        for (i = 31, carry = 0; i >= 0 && (carry != 0 || value != LL(0)); i--)  {
                carry += bitLength[i] + ((u32)value & 0xff);
                bitLength[i] = (u8)carry;
                carry >>= 8;
```

```
            value >>= 8;
    }
    /*  Procesa datos en bloques de 8 bits  */
    while (sourceBits > 8)  {
            // N.B. al menos source[sourcePos] y source[sourcePos+1] contiene datos.
            /*  Se toma un byte de la fuente.  */
            b = ((source[sourcePos] << sourceGap) & 0xff) |
                    ((source[sourcePos + 1] & 0xff) >> (8 - sourceGap));
            /*  Procesa este byte  */
            buffer[bufferPos++] |= (u8)(b >> bufferRem);
            bufferBits += 8 - bufferRem;          /* bufferBits = 8*bufferPos; */
            if (bufferBits == DIGESTBITS)  {
                    processBuffer(structpointer);         /*  Procesa el bloque de datos  */
                    bufferBits = bufferPos = 0;           /*  Reinicia el buffer  */
            }
            buffer[bufferPos] = b << (8 - bufferRem);
            bufferBits += bufferRem;
            /*  Procesa los datos restantes  */
            sourceBits -= 8;
            sourcePos++;
    }
    /*  Ahora 0 <= sourceBits <= 8;  */
    if (sourceBits > 0)  {
            b = (source[sourcePos] << sourceGap) & 0xff; /* bits are left-justified on b. */
            /*  Procesa los bits restantes  */
            buffer[bufferPos] |= b >> bufferRem;
    }
    else  {
            b = 0;
    }
    if (bufferRem + sourceBits < 8)  {
            /* all remaining data fits on buffer[bufferPos], and there still remains some space.  */
            bufferBits += sourceBits;
    }
    else  {
            /*  buffer[bufferPos] esta lleno  */
            bufferPos++;
            bufferBits += 8 - bufferRem; /* bufferBits = 8*bufferPos; */
            sourceBits -= 8 - bufferRem;
            /*   Ahora 0 <= sourceBits < 8;  */
            if (bufferBits == DIGESTBITS)  {
                    processBuffer(structpointer);         /*  Procesa el bloque de datos  */
```

```
                    bufferBits = bufferPos = 0;          /*  Reinicia el buffer.  */
                }
                buffer[bufferPos] = b << (8 - bufferRem);
                bufferBits += (int)sourceBits;
        }
        structpointer->bufferBits   = bufferBits;
        structpointer->bufferPos    = bufferPos;
}
```

5.6 Finalización

El *hash value* corresponderá a H_t después de procesar en su totalidad el mensaje mostrando dicha representación en formato hexadecimal de la siguiente manera:

$$LT - CHA(M) = H_t$$

```
static void display(const u8 array[ ], int length)
{
        int i;
        for (i = 0; i < length; i++)  {
                if (i%32 == 0)  {
                        printf("\n");
                }
                if (i%8 == 0)  {
                        printf(" ");
                }
                printf("%02X", array[i]);
        }
}
```

6. Análisis de Resultados

"... Lúthien les salió al encuentro andando lentamente... ...abrazó a Beren, y lo besó, pidiéndole que la esperara más allá del Mar Occidental…"

"Por causa de sus fatigas y sus dolores, podría abandonar a Mandos e ir a Valinor, para morar allí hasta el fin de los días. Allí no la seguiría Beren…
… pero la otra elección posible era la que sigue: regresar a la Tierra Media y llevar consigo a Beren para morar allí otra vez, mas sin ninguna seguridad de vida o alegría. Ella se volvería entonces mortal y estaría sometida a una segunda muerte, lo mismo que él, y antes de no mucho abandonaría el mundo para siempre y su belleza no sería más que un recuerdo en el canto..."

J. R. R. Tolkien

En la definición final de la función ***Lúthien-Tinúviel Chaos Hash*** se utilizan dos permutaciones las cuales fueron previamente mostradas, una que solo actúa sobre el bloque *LT-CHA* que sirve para calcular las constantes de ronda a utilizar y otra que actúa exclusivamente sobre el bloque *LT-CHA* del mensaje entrante.

En la primera definición de la función ***Lúthien-Tinúviel Chaos Hash*** sólo se utilizaba una permutación la cual correspondía al mapa Baker, sin embargo, al implementar la función se encontraron ciertos valores que presentaban un alto grado de correlación y por ende de colisión, debido a que si bien el mapeo realiza una distribución de valores este tiene la característica de permutar columnas con renglones, por lo que la distribución no es uniforme y provoca resultados como los que se mostrarán en esta sección.

6.1 Análisis Estadístico

Un análisis estadístico es realizado a la función ***Lúthien-Tinúviel Chaos Hash*** para obtener sus características principales así como ver sus diferencias con respecto a Whirlpool función en la cual se basa su diseño.

6.1.1 Espacio muestral con longitud de bloque variable

Para la realización de este análisis se procede en primera instancia a generar el espacio muestral que definiremos de la siguiente manera: "*El conjunto formado por los hash values de las cadenas de 0-bits cuya longitud* ***L*** *esta en el intervalo* $0 \leq L < 2048$".

Lúthien-Tinúviel Chaos Hash:

Longitud	# 1-bits Bloque del LT-CHA State								
	Total	0	1	2	3	4	5	6	7
0	253	35	31	30	30	35	34	30	28
1	270	33	36	34	35	32	34	33	33
2	254	27	36	32	29	34	31	31	34
3	248	23	33	28	35	26	36	32	35
…									
512	244	35	27	31	28	23	34	35	31
513	256	35	30	32	30	27	28	38	36
514	271	31	27	35	37	43	30	34	34
…									
1024	270	32	36	36	36	28	31	40	31
1025	239	31	28	30	32	31	33	22	32
1026	251	25	32	29	31	33	39	36	26
1027	251	34	36	31	26	28	30	30	36
…									
2044	257	28	30	24	36	40	31	35	33
2045	253	27	33	28	26	31	34	34	40
2046	245	20	33	37	29	36	32	24	34
2047	265	36	32	43	32	32	33	33	24

Tabla 8. Estadísticas del espacio muestral de longitud de bloque variable en LT-CHA. y ***Whirlpool***:

Longitud	# 1-bits Bloque del Whirlpool State								
	Total	0	1	2	3	4	5	6	7
0	256	31	31	25	38	35	33	35	28
1	257	31	28	29	31	35	31	37	35
2	262	37	34	37	28	26	35	28	37
3	262	33	33	36	36	30	31	26	37
…									
512	279	36	31	39	31	38	32	39	33
513	270	37	32	33	33	38	34	29	34
514	256	26	31	37	33	38	30	36	25
…									
1024	284	45	41	33	29	36	33	35	32
1025	251	26	32	31	25	30	37	34	36
1026	246	33	25	30	29	36	32	33	28
1027	261	32	36	30	32	35	35	29	32
…									
2044	277	38	27	36	37	34	32	37	36
2045	262	32	34	38	32	35	31	32	28
2046	271	34	35	27	35	34	33	38	35
2047	259	28	33	40	32	29	34	34	29

Tabla 9. Estadísticas del espacio muestral de longitud de bloque variable en Whirlpool.

A partir de los datos anteriores se obtienen los siguientes resultados con respecto a las medidas estadísticas de la media y desviación estándar.

LT-CHA	32.00439	32.07129	31.9502	31.94971	31.9043	32.02148	31.94287	31.96875
Whirlpool	31.9458	31.97217	32.02637	31.97314	31.93359	31.98047	32.1333	32.14746
Media	32	32	32	32	32	32	32	32

Desviación estándar

LT-CHA	4.049436	4.136796	3.909556	4.060566	3.916065	4.026102	3.949159	3.932661
Whirlpool	3.995419	3.934486	4.006256	3.959112	3.959695	3.970424	4.120773	4.005701

Graficando cada una de las medidas obtenemos:

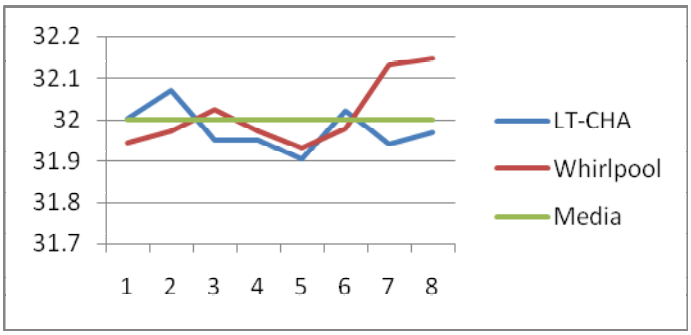


Figura 30. Media en bits de cada bloque que conforman el state del bloque de longitud variable.

Como se puede ver para este espacio muestral la función LT-CHA tiene una ligera tendencia (255.8129883) al equilibrio en cuanto a la cantidad de 1-bit y 0-bits que conforman el hash value en contraparte en la función Whirlpool sucede lo contrario (256.1123) el hash value tiende a tener más 1-bits, por lo que al menos para este espacio muestral LT-CHA tiene una mejor distribución cercana a la media, empero dicha distribución no es uniforme para el state definido sino que es relativamente diferente entre ambas funciones en los bloques iniciales y finales, en la grafica anterior se observa este comportamiento diferencial en los bloques 1, 2, 3, 7 y 8 en los que difiere siendo semejante en los restantes.

Graficando la medida de desviación estándar

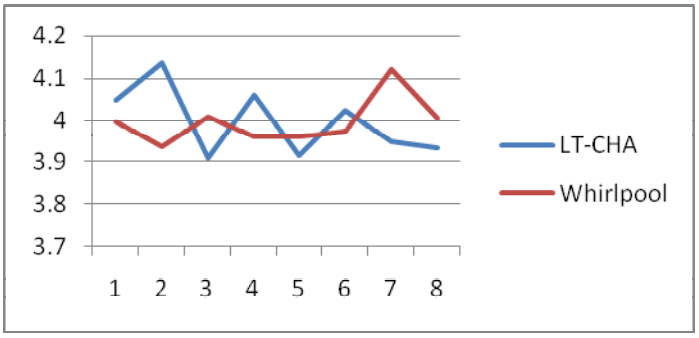


Figura 31. Desviación estándar de cada uno de los bloques que conforman el state.

Un comportamiento semejante es obtenido al calcular la deviación estándar del espacio muestral que definimos cabe destacar que éste tiene longitud de bloque variable por lo que

presentaremos el comportamiento de otro espacio muestral en el que como característica adicional la longitud del bloque es la misma y lo que cambia es el mensaje de entrada como tal.

6.1.2 Espacio muestral con longitud de bloque fija

Para la realización de este análisis se procede en primera instancia a generar el espacio muestral que definiremos de la siguiente manera: "*El conjunto S formado por todos los hash values de las cadenas de 1024-bits conteniendo un solo 1-bit*".

Lúthien-Tinúviel Chaos Hash:

Longitud	# 1-bits Bloque del LT-CHA State								
	Total	0	1	2	3	4	5	6	7
0	275	32	39	31	40	29	34	33	37
1	252	32	35	34	25	33	31	35	27
2	257	36	32	27	32	31	35	29	35
3	231	31	23	32	27	32	28	35	23
…									
512	263	36	36	32	36	33	28	29	33
513	261	34	31	29	33	37	37	28	32
514	241	33	29	26	35	34	21	32	31
515	253	32	29	35	35	26	33	32	31
…									
1020	255	32	31	33	27	31	31	37	33
1021	259	34	41	32	33	31	34	27	27
1022	238	34	24	29	28	26	38	32	27
1023	259	30	28	31	33	34	35	36	32

Tabla 10. Estadísticas del espacio muestral de longitud de bloque fija en LT-CHA.

y ***Whirlpool***:

Longitud	# 1-bits Bloque del Whirlpool State								
	Total	0	1	2	3	4	5	6	7
0	261	34	29	35	33	25	41	29	35
1	262	27	39	35	33	34	32	28	34
2	262	28	23	33	44	35	35	31	33

3	273	33	33	36	36	32	24	35	44
…									
512	259	39	30	28	32	26	38	34	32
513	244	28	32	27	31	35	26	28	37
514	255	31	35	36	30	32	32	33	26
515	259	33	30	31	34	37	35	32	27
…									
1020	266	37	29	36	34	30	31	33	36
1021	244	30	26	33	35	30	33	30	27
1022	254	31	30	28	34	31	31	32	37
1023	263	34	37	39	32	35	23	38	25

Tabla 11. Estadísticas del espacio muestral de longitud de bloque fija en Whirlpool.

A partir de los datos anteriores se obtienen los siguientes resultados con respecto a las medidas estadísticas de la media y desviación estándar.

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

Desviación estándar

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

Graficando cada una de las medidas obtenemos:

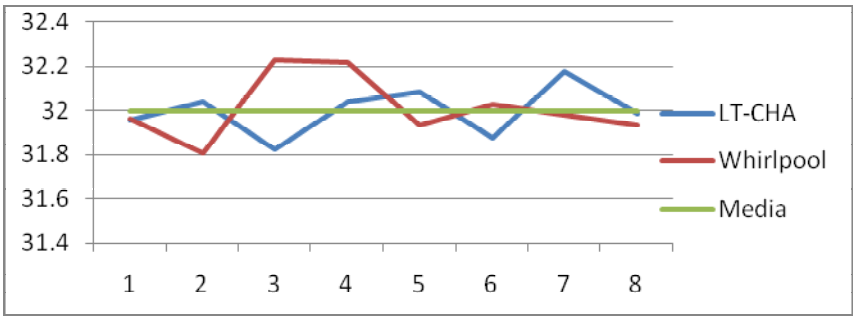


Figura 32. Media en bits de cada bloque que conforman el state del bloque de longitud fija.

Como se puede ver para este espacio muestral la función LT-CHA tiene una ligera tendencia (255.97363) al equilibrio en cuanto a la cantidad de 1-bit y 0-bits que conforman el hash value en contraparte en la función Whirlpool sucede lo contrario (256.1055) el hash value tiende a tener más 1-bits, por lo que al menos nuevamente para este espacio muestral LT-CHA tiene una mejor distribución cercana a la media, empero dicha distribución no es uniforme ya que los bloques 1, 2, 4, 5 y 8 del state de LT-CHA tiene una mayor aproximación a la media en contraparte Whirlpool presenta dicho comportamiento en los bloques 1, 5, 6, 7 y 8, en los bloques restantes en ambas funciones divergen de la media.

Graficando la medida de la desviación estándar

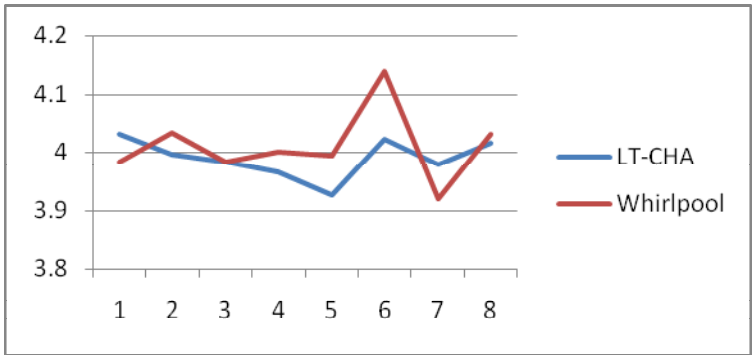


Figura 33. Desviación estándar de cada uno de los bloques que conforman el state.

Ambas funciones tienen un grafo semejante en cuanto a la desviación estándar variando dicho grafo en los bloque 1, 2 y 4 en los cuales los valores tienen una tendencia contraria una respecto de la otra.

6.2 Análisis de Rendimiento

Se realizan pruebas de rendimiento entre las funciones LT-CHA y Whirlpool para medir el rendimiento de las mismas cuando éstas son aplicadas a bloques de datos de diferentes longitudes.

A continuación listaremos 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 serán realizadas en dos equipos cuyas características son listadas a continuación:

- Equipo con 2 procesadores Quad Core de 64 bits a 3 GHz, velocidad de bus 1333MHz, 48 Gb en RAM y dos unidades de disco con capacidad de 450 GB.
- Equipo 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.

Para diferenciar los resultados obtenidos en cada uno de los equipos les daremos la notación de desktop al primero de estos y de laptop al segundo, esta notación será utilizada para la etiquetación de las tablas y gráficas obtenidas.

Al efectuar cada una de las pruebas se obtienen los siguientes resultados:

DATOS	LT-CHA (s)	Whirlpool (s)
10 KB	0.0000000000	0.0000000000
30 KB	0.0000000000	0.0000000000
90 KB	0.0000000000	0.0000000000
950 KB	0.0000000000	0.0000000000
1150 KB	0.0000000000	0.0099999998
2 Mb	0.0099999998	0.0000000000
4 MB	0.0099999998	0.0099999998
10 MB	0.0199999996	0.0199999996
20 MB	0.0599999987	0.0500000007
50 MB	0.1299999952	0.1000000015
100 MB	0.2800000012	0.2199999988
200 MB	0.5400000215	0.4399999976
500 MB	1.3600000143	1.0900000334

1 GB	2.7799999714	2.2400000095
2 GB	5.5700001717	4.4699997902
4 GB	11.1400003433	8.9399995804
8 GB	22.2700004578	17.8799991608
20 GB	55.6800003052	44.7299995422
40 GB	111.3600006104	89.4899978638

Tabla 12. Tiempo para procesar un bloque de longitud L entre Whirlpool y LT-CHA en desktop.

Graficando los valores de la tabla anterior obtenemos:

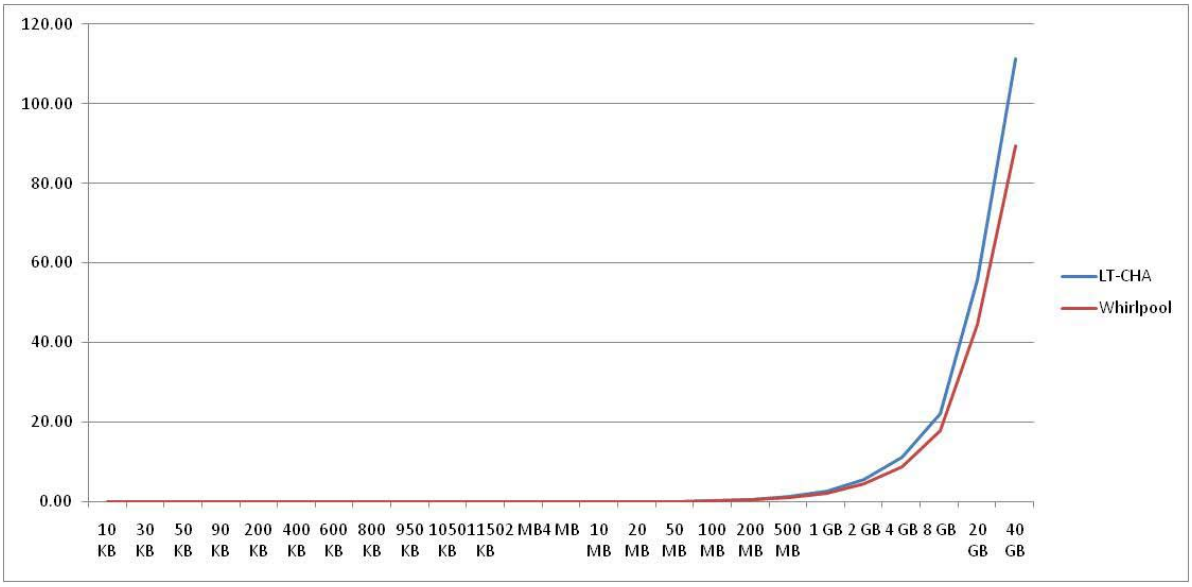


Figura 34. Rendimiento de Whirlpool y LT-CHA en desktop.

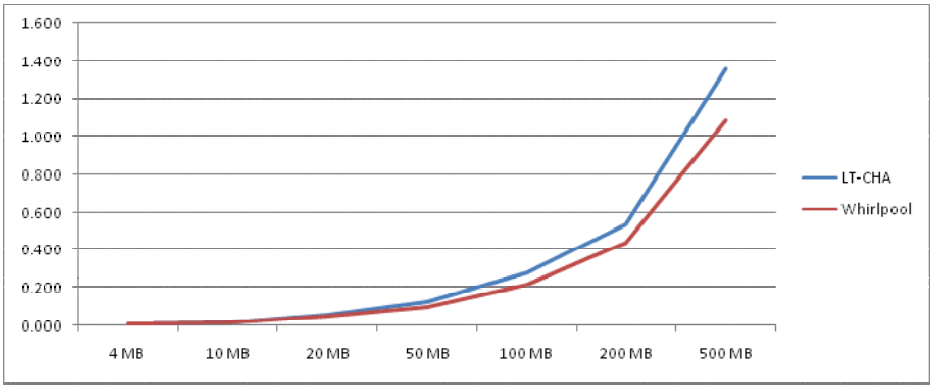


Figura 35. Rendimiento de Whirlpool y LT-CHA con bloque de datos pequeños en desktop.

En cuya gráfica se observa que para valores pequeños LT-CHA tiene el mismo rendimiento que Whirlpool al menos la diferencia no es perceptible, sin embargo, para bloques de datos mayores a 10 MB el rendimiento de Whirlpool tiende a ser mejor que LT-CHA y este se incrementa conforme aumenta la cantidad de bloques a procesar. Para el caso de extrapolación de 40Gb la diferencia ya es notoria en cuanto al tiempo necesario para procesar la cantidad de bloques obtenidos de los datos especificados.

Los resultados correspondientes a la segunda prueba son:

Veces	LT-CHA	Whirlpool
100000000	353.6899719	301.8099975586
	353.8099976	301.6400146484
	353.9200134	301.5500183105
	353.2200012	302.0799865723
	353.6499939	302.0000000000

Tabla 13. Rendimiento en la iteración de la función hash Whirlpool y LT-CHA en desktop.

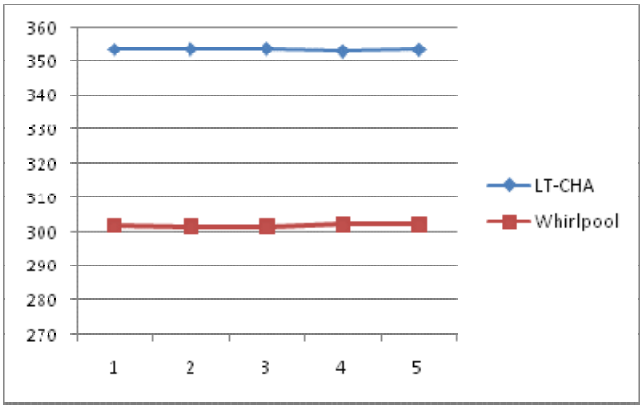


Figura 36. Comparación en eficiencia al iterar a la función Whirlpool y LT-CHA en desktop.

En la grafica anterior se observa que LT-CHA toma más tiempo que Whirlpool en procesar 100000000 veces la función hash con lo que en general LT-CHA es más lento que Whirlpool, tanto en el procesamiento de bloques de datos para calcular el hash value como para iterar la función hash individualmente aunque este resultado no se refleje de forma notable para el procesamiento de una pequeña cantidad de bloques.

Las mismas pruebas realizadas en el equipo denominado como laptop presentan el siguiente comportamiento:

DATOS	LT-CHA (s)	Whirlpool (s)
10 KB	0.0000000000	0.0000000000
30 KB	0.0000000000	0.0000000000
90 KB	0.0000000000	0.0000000000
950 KB	0.0099999998	0.0199999996
1050 KB	0.0199999996	0.0099999998
1150 KB	0.0099999998	0.0099999998
2 Mb	0.0299999993	0.0199999996
4 MB	0.0599999987	0.0500000007
10 MB	0.1400000006	0.1199999973
20 MB	0.2899999917	0.2300000042
50 MB	0.7200000286	0.5799999833
100 MB	1.4400000572	1.1699999571
200 MB	2.8900001049	2.3299999237
500 MB	7.2399997711	5.8499999046
1 GB	14.8299999237	11.9700002670
2 GB	29.5699996948	23.9300003052
4 GB	59.0999984741	47.9000015259
8 GB	118.2099990845	95.9499969482
20 GB	295.4500122070	239.4100036621
40 GB	590.9799804688	479.4299926758

Tabla 14. Tiempo para procesar un bloque de longitud L entre Whirlpool y LT-CHA en laptop.

Graficando los valores de la tabla anterior obtenemos:

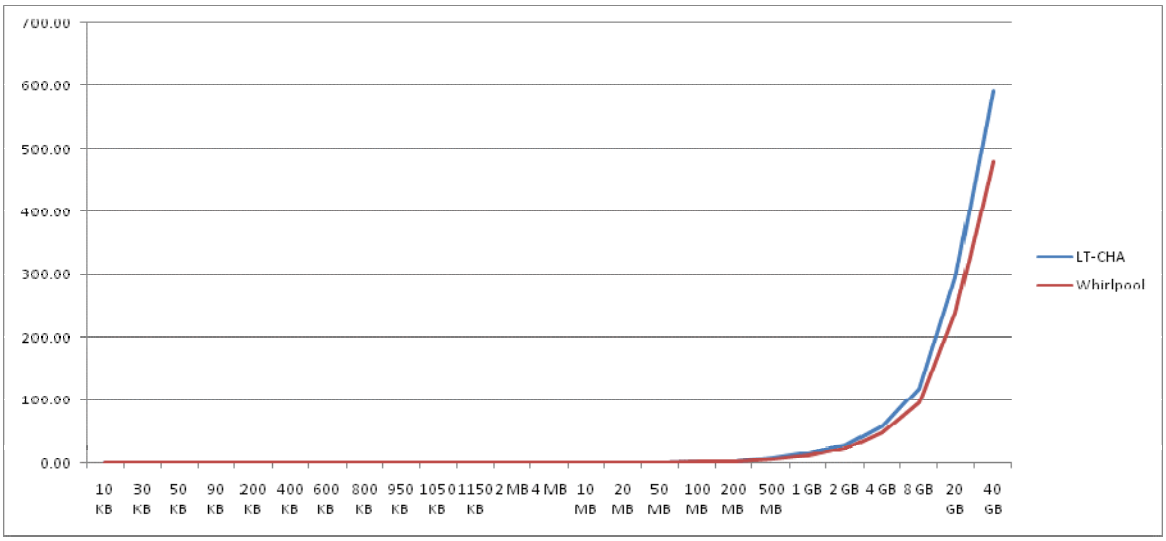


Figura 37. Rendimiento de Whirlpool y LT-CHA en laptop.

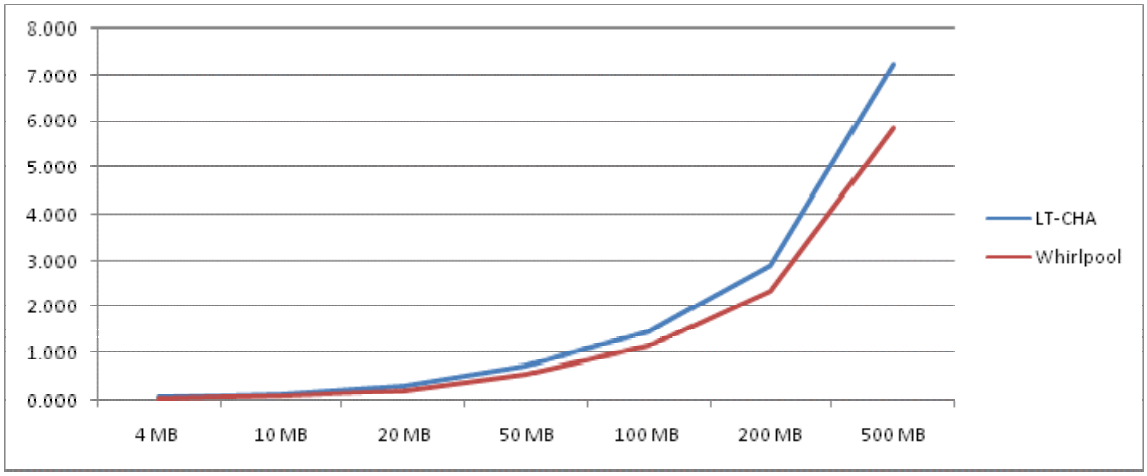


Figura 38. Rendimiento de Whirlpool y LT-CHA con bloque de datos pequeños en laptop.

En cuya gráfica se observa que al igual que en el análisis anterior para valores pequeños LT-CHA tiene el mismo rendimiento que Whirlpool al menos la diferencia no es perceptible, sin embargo, para bloques de datos mayores a 4 MB el rendimiento de Whirlpool tiende a ser mejor que LT-CHA y este se incrementa conforme aumenta la cantidad de bloques a procesar en un equipo con las características antes mencionadas. Para el caso de

extrapolación de 40Gb la diferencia ya es notoria en cuanto al tiempo necesario para procesar la cantidad de bloques obtenidos de los datos especificados.

Los resultados correspondientes a la segunda prueba son:

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

Tabla 15. Rendimiento en la iteración de la función hash Whirlpool y LT-CHA en laptop.

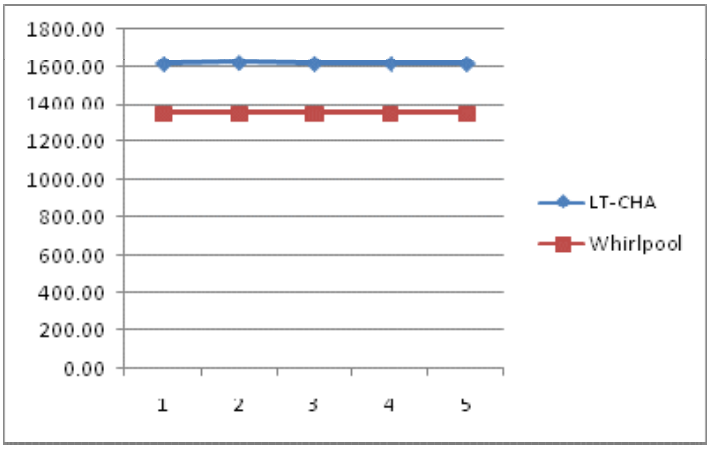


Figura 39. Comparación en eficiencia al iterar a la función Whirlpool y LT-CHA en laptop.

Al igual que en el análisis anterior se observa que LT-CHA toma más tiempo que Whirlpool en procesar 100000000 veces la función hash aunque este resultado no sea notorio al procesar una pequeña cantidad de bloques de datos.

6.3 Análisis de Seguridad

Para las funciones hash la no lineabilidad es la característica más importante para evitar ataques tales como del cumpleaños, primera y segunda preimagen; puesto que está permite dificultar tanto el criptoanálisis lineal como el diferencial, para esto, se debe

asegurar que el espacio de mensajes en el que la probabilidad de encontrar una colisión (del grado que sea) sea alta, no se reduzca a un conjunto compacto de mensajes sino que dicho conjunto este lo más disperso posible dentro del dominio de la función.

A continuación mostraremos la justificación del porque no utilizamos el mapa Baker tanto para procesar el bloque LT-CHA como para calcular las constantes de ronda para LT-CHA.

Al aplicar el mapa Baker tanto para la generación de constantes como para el procesamiento del bloque LT-CHA se encuentra que para el espacio muestral definido como:

"*El conjunto S formado por todos los hash values de las cadenas de 512-bits conteniendo un solo 1-bit*".

Se tiene un conjunto compacto de salidas con un alto grado de correlación, por lo que es posible obtener una colisión con una probabilidad alta debido a la poca variabilidad en el hash value resultante (presenta colisiones internas lo cual permitiría la aplicación del *meet in the middle* o multicolisiones de Joux) de mensajes en los cuales no existe una gran variabilidad.

Para mostrar dicho problema primero se listarán dos mensajes diferentes junto con el grado de correlación cuando se utiliza siempre el mapa Baker y posteriormente el resultado obtenido para el mismo conjunto de entradas utilizando la definición final de la permutación para el bloque LT-CHA.

6.3.1 Resultados de utilizar solamente el mapa Baker en LT-CHA.

Los tres mensajes que serán utilizados para mostrar el problema anteriormente mencionado son los siguientes:

S[8] = **0080000000000000 0000000000000000 0000000000000000 0000000000000000**
0000000000000000 0000000000000000 0000000000000000 0000000000000000

S[9] = **0040000000000000 0000000000000000 0000000000000000 0000000000000000**
0000000000000000 0000000000000000 0000000000000000 0000000000000000

Mostraremos el desarrollo de cada una de las rondas tanto del bloque LT-CHA como de la llave en turno a fin de observar internamente el comportamiento de la misma de los tres bloques simultáneamente. En primera instancia la llave **K0** y el primer bloque de ambos mensajes posteriormente se irán separando ambas funciones por cada ronda.

K0	S[8]	S[9]
3B 82 D8 00 0C D5 D2 90	00 80 00 00 00 00 00 00	00 40 00 00 00 00 00 00
03 25 9B CC 31 CC 9C C0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
15 B3 F3 40 11 3B 38 60	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
1C F3 5D 00 30 44 1E 40	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
2D 99 75 80 35 75 60 C0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
2F A6 F3 40 21 1C 15 40	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
17 F8 C0 E0 0D B5 C6 C0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
38 04 AE 00 21 60 AA 80	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00

Ronda 1

K1	LT-CHA[1]	LT-CHA[1]
CE 94 BD 7D EF D3 C1 85	C9 02 A6 6F 6B C1 CF 85	C9 02 A6 6F 6B C1 CF 85
89 4C 76 FC F1 F6 E3 D9	18 C1 7F 6A 67 EA ED DE	18 C1 7F 6A 67 EA ED DE
3B 3B 45 FF B4 AB 97 9A	AD 20 57 7B A6 A5 97 9D	AD 20 57 7B A6 A5 97 9D
A8 83 14 40 5A E9 8E FD	BA 07 06 4E 5A EE 18 E6	BA 07 06 4E 5A EE 18 E6
B9 9A 8A 94 FA 63 DB 08	AB 94 8A 93 6C 78 C9 8C	AB 94 8A 93 6C 78 C9 8C
E6 E1 F5 1D F8 28 58 E0	E6 E1 F5 1D F8 28 58 E0	E6 E1 F5 1D F8 28 58 E0
D7 23 B4 54 4A 69 F7 BA	C5 2D B4 53 DC 72 E5 3E	C5 2D B4 53 DC 72 E5 3E
FF 3F 93 AA 5E 63 B6 73	ED BB 81 A4 5E 64 20 68	ED BB 81 A4 5E 64 20 68

Ronda 2:

K2	LT-CHA[2]	LT-CHA[2]
A2 97 E2 3C D4 D0 A6 AC	2F F8 2E 4D E9 84 98 A5	2F F8 2E 4D E9 84 98 A5
AC E2 E0 D9 28 3B 74 76	28 4B FF 60 6C 12 F7 9B	28 4B FF 60 6C 12 F7 9B
AE 2F A1 1E 4D 1C 64 70	1C 45 B8 33 78 5D 67 65	1C 45 B8 33 78 5D 67 65
57 44 6E EC 9F 43 E4 5E	2A 4E 09 EE B0 74 BD 42	2A 4E 09 EE B0 74 BD 42
C0 05 4B A0 E9 92 CD C7	D7 31 90 AB C0 6F DB B0	D7 31 90 AB C0 6F DB B0
9F 24 13 0B 77 07 CD 49	B7 A8 87 67 D9 27 FA 5D	B7 A8 87 67 D9 27 FA 5D
30 18 75 C3 E9 0E 6C BC	2B 13 11 F1 9B 9E 24 FF	2B 13 11 F1 9B 9E 24 FF
57 1A 0C 45 BF 04 BA 7E	FB AF C0 94 2E E4 E2 0D	FB AF C0 94 2E E4 E2 0D

Ronda 12:

K12	LT-CHA[12]	LT-CHA[12]
14 C7 29 E2 1E 4F 97 11	82 15 DD BD 61 D9 A8 8C	82 15 DD BD 61 D9 A8 8C
C9 CD 61 15 BB 33 1D 82	F8 DE C9 6C ED 0B 28 C9	F8 DE C9 6C ED 0B 28 C9
6E D4 86 00 D4 25 BD 90	6E 68 44 8F 8E 6A 13 3C	6E 68 44 8F 8E 6A 13 3C

CA 27 F8 17 84 C1 77 01	C7 B6 64 C5 5D 1A 96 CC	C7 B6 64 C5 5D 1A 96 CC
68 7F 71 03 84 76 1B CF	33 07 D5 58 EB D7 85 4E	33 07 D5 58 EB D7 85 4E
80 84 26 F4 04 0E 80 EC	70 98 6A A8 7D 7A 01 33	70 98 6A A8 7D 7A 01 33
51 F5 E0 59 8A E0 96 81	2E 93 49 AF FB A1 72 84	2E 93 49 AF FB A1 72 84
D3 47 1E E1 35 93 30 AE	B2 78 F2 A2 DD FC 67 48	B2 78 F2 A2 DD FC 67 48

Hash del primer bloque procesado

LT-CHA(Bloque0)	**LT-CHA(Bloque0)**
82 95 DD BD 61 D9 A8 8C	82 55 DD BD 61 D9 A8 8C
F8 DE C9 6C ED 0B 28 C9	F8 DE C9 6C ED 0B 28 C9
6E 68 44 8F 8E 6A 13 3C	6E 68 44 8F 8E 6A 13 3C
C7 B6 64 C5 5D 1A 96 CC	C7 B6 64 C5 5D 1A 96 CC
33 07 D5 58 EB D7 85 4E	33 07 D5 58 EB D7 85 4E
70 98 6A A8 7D 7A 01 33	70 98 6A A8 7D 7A 01 33
2E 93 49 AF FB A1 72 84	2E 93 49 AF FB A1 72 84
B2 78 F2 A2 DD FC 67 48	B2 78 F2 A2 DD FC 67 48

Como se puede ver después de procesar el primer bloque del mensaje este no tiene gran variación uno respecto del otro salvo un solo bit. Al procesar el segundo bloque obtenemos lo siguiente:

K0	**LT-CHA**	**K0**	**LT-CHA**
B9 17 05 BD 6D 0C 7A 1C	02 95 DD BD 61 D9 A8 8C	B9 D7 05 BD 6D 0C 7A 1C	02 55 DD BD 61 D9 A8 8C
FB FB 52 A0 DC C7 B4 09	F8 DE C9 6C ED 0B 28 C9	FB FB 52 A0 DC C7 B4 09	F8 DE C9 6C ED 0B 28 C9
7B DB B7 CF 9F 51 2B 5C	6E 68 44 8F 8E 6A 13 3C	7B DB B7 CF 9F 51 2B 5C	6E 68 44 8F 8E 6A 13 3C
DB 45 39 C5 6D 5E 88 8C	C7 B6 64 C5 5D 1A 96 CC	DB 45 39 C5 6D 5E 88 8C	C7 B6 64 C5 5D 1A 96 CC
1E 9E A0 D8 DE A2 E5 8E	33 07 D5 58 EB D7 85 4E	1E 9E A0 D8 DE A2 E5 8E	33 07 D5 58 EB D7 85 4E
5F 3E 99 E8 5C 66 14 73	70 98 6A A8 7D 7A 01 33	5F 3E 99 E8 5C 66 14 73	70 98 6A A8 7D 7A 01 33
39 6B 89 4F F6 14 B4 44	2E 93 49 AF FB A1 72 84	39 6B 89 4F F6 14 B4 44	2E 93 49 AF FB A1 72 84
8A 7C 5C A2 FC 9C CD C8	B2 78 F2 A2 DD FC 65 48	8A 7C 5C A2 FC 9C CD C8	B2 78 F2 A2 DD FC 65 48

Ronda 1:

K1	**LT-CHA[1]**	**K1**	**LT-CHA[1]**
D7 6D 2C 3F A1 6D C2 A4	2A FC BD B9 30 F5 45 09	D7 6D 2C 3F A1 6D C2 A4	2A FC BD B9 30 F5 45 09
EF 44 6A CE 56 BB 67 27	7F FB FA 76 02 77 DC F5	EF 44 6A CE 56 BB 67 27	7F FB FA 76 02 77 DC F5
7C 5B 68 35 9A 58 3A DD	7B BE 5A E5 91 FF B0 07	7C 5B 68 35 9A 58 3A DD	7B BE 5A E5 91 FF B0 07
15 B2 B2 F9 FB 21 52 95	31 B2 C5 C5 2E D9 7F CB	15 B2 B2 F9 FB 21 52 95	31 B2 C5 C5 2E D9 7F CB
89 75 30 5D 17 CA 78 C4	B3 3F B5 EB 58 96 9F E6	89 75 30 5D 17 CA 78 C4	B3 3F B5 EB 58 96 9F E6
A7 DC 98 FF 67 A6 EB 4E	75 F0 EF B0 46 E0 92 C1	A7 DC 98 FF 67 A6 EB 4E	75 F0 EF B0 46 E0 92 C1
95 BF 1D A2 DB 91 3D E7	93 71 5C 25 A8 23 D2 E2	95 BF 1D A2 DB 91 3D E7	93 71 5C 25 A8 23 D2 E2
30 BB F2 82 7B 42 2F 9E	C6 8D 9B 1F 04 F5 00 97	30 BB F2 82 7B 42 2F 9E	C6 8D 9B 1F 04 F5 00 97

Ronda 2:

K2	LT-CHA[2]	K2	LT-CHA[2]
A7 DB AF 93 9C 7F CD 91	48 06 5A 82 91 BB 39 9E	A7 DB AF 93 9C 7F CD 91	48 06 5A 82 91 BB 39 9E
C3 7F 0B 47 2A 14 86 F8	FC 1F 9E 7B DE 0A D4 A0	C3 7F 0B 47 2A 14 86 F8	FC 1F 9E 7B DE 0A D4 A0
21 0A 38 18 4E 5B FC E0	19 3B 49 35 87 84 FE C9	21 0A 38 18 4E 5B FC E0	19 3B 49 35 87 84 FE C9
B7 89 0C 6A 35 C4 EA 48	A9 3B A4 DA FE 37 39 07	B7 89 0C 6A 35 C4 EA 48	A9 3B A4 DA FE 37 39 07
E5 1D 64 42 AD C0 4E BC	9D 6C 49 17 A0 10 B0 20	E5 1D 64 42 AD C0 4E BC	9D 6C 49 17 A0 10 B0 20
AB 39 95 E6 58 C6 B7 85	4D 84 AB DF 6B 83 E2 52	AB 39 95 E6 58 C6 B7 85	4D 84 AB DF 6B 83 E2 52
C9 CD D6 53 99 B3 E2 28	46 61 74 6E 74 3E 6F A3	C9 CD D6 53 99 B3 E2 28	46 61 74 6E 74 3E 6F A3
5F B7 05 67 74 CB 51 BC	BA EF E8 DA 4D 04 10 96	5F B7 05 67 74 CB 51 BC	BA EF E8 DA 4D 04 10 96

Ronda 12:

K12	LT-CHA[12]	K12	LT-CHA[12]
A2 69 1E 29 91 AD 6A 8D	0B 86 C2 C2 32 05 F2 A8	A2 69 1E 29 91 AD 6A 8D	0B 86 C2 C2 32 05 F2 A8
8E A0 8A E4 0E 34 58 24	9E 98 58 50 8E 84 92 48	8E A0 8A E4 0E 34 58 24	9E 98 58 50 8E 84 92 48
BE C0 08 AF F8 B0 C9 36	B5 23 35 70 9B 94 96 9E	BE C0 08 AF F8 B0 C9 36	B5 23 35 70 9B 94 96 9E
D3 4F 30 43 90 43 D1 09	E1 8D CB C1 69 B9 75 4A	D3 4F 30 43 90 43 D1 09	E1 8D CB C1 69 B9 75 4ª
4B D0 21 B9 3D FD E9 5E	65 19 43 FC D2 46 9F 42	4B D0 21 B9 3D FD E9 5E	65 19 43 FC D2 46 9F 42
6A 61 AD 86 DA 29 79 49	FD F4 03 22 64 FE E6 9D	6A 61 AD 86 DA 29 79 49	FD F4 03 22 64 FE E6 9D
0E FD 71 11 1D 46 0F 5A	A4 3C 8C 11 D7 71 09 FC	0E FD 71 11 1D 46 0F 5A	A4 3C 8C 11 D7 71 09 FC
8B F9 A6 42 CF 99 7F 1B	A6 E6 59 D6 66 F4 60 00	8B F9 A6 42 CF 99 7F 1B	A6 E6 59 D6 66 F4 60 00

Hash final de ambos mensajes

LT-CHA(M)	LT-CHA(M)
09 13 1F 7F 53 DC 5A 24	09 D3 1F 7F 53 DC 5A 24
66 46 91 3C 63 8F BA 81	66 46 91 3C 63 8F BA 81
DB 4B 71 FF 15 FE 85 A2	DB 4B 71 FF 15 FE 85 A2
26 3B AF 04 34 A3 E3 86	26 3B AF 04 34 A3 E3 86
56 1E 96 A4 39 91 1A 0C	56 1E 96 A4 39 91 1A 0C
8D 6C 69 8A 19 84 E7 AE	8D 6C 69 8A 19 84 E7 AE
8A AF C5 BE 2C D0 7B 78	8A AF C5 BE 2C D0 7B 78
14 9E AB 74 BB 08 05 48	14 9E AB 74 BB 08 05 48

Como se ve con este ejemplo el grado de variabilidad entre cada uno de los mensajes es ínfimo por lo que con esta versión de nuestra función es muy vulnerable. Al utilizar diferentes funciones de permutación para la generación de llave y para el procesamiento del bloque se tiene lo siguiente, utilizando el mismo conjunto de entradas del ejemplo anterior.

El primer bloque a ser procesado junto con la primera constante de ronda son:

K0	S[8]	S[9]
3B 82 D8 00 0C D5 D2 90	00 80 00 00 00 00 00 00	00 40 00 00 00 00 00 00
03 25 9B CC 31 CC 9C C0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
15 B3 F3 40 11 3B 38 60	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
1C F3 5D 00 30 44 1E 40	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
2D 99 75 80 35 75 60 C0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
2F A6 F3 40 21 1C 15 40	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
17 F8 C0 E0 0D B5 C6 C0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
38 04 AE 00 21 60 AA 80	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00

Ronda 1:

K1	LT-CHA[1]	LT-CHA[1]
CE 94 BD 7D EF D3 C1 85	58 02 2B EB 79 45 57 13	58 02 2B EB 79 45 57 13
89 4C 76 FC F1 F6 E3 D9	1F DA E0 6A 67 60 75 4F	1F DA E0 6A 67 60 75 4F
3B 3B 45 FF B4 AB 97 9ª	AD AD D3 69 22 3D 01 0C	AD AD D3 69 22 3D 01 0C
A8 83 14 40 5A E9 8E FD	3E 15 82 D6 CC 7F 18 6B	3E 15 82 D6 CC 7F 18 6B
B9 9A 8A 94 FA 63 DB 08	2F 0C 1C 02 6C F5 4D 9E	2F 0C 1C 02 6C F5 4D 9E
E6 E1 F5 1D F8 28 58 E0	70 77 63 8B 6E BE CE 76	70 77 63 8B 6E BE CE 76
D7 23 B4 54 4A 69 F7 BA	41 B5 22 C2 DC FF 61 2C	41 B5 22 C2 DC FF 61 2C
FF 3F 93 AA 5E 63 B6 73	**54 1E B2 CE 8D CA AA 6D**	**91 C6 6A 8D 16 F0 B7 8F**

En las siguientes rondas sólo marcaremos los bits que se mantienen constantes después de procesar el bloque a fin de notar el impacto en el cálculo del nuevo bloque para mostrar que la permutación ahora definida permite una mejor difusión del los bits.

Ronda 2

K2	LT-CHA[2]	LT-CHA[2]
A2 97 E2 3C D4 D0 A6 AC	96 DC 57 61 4F F7 84 0D	CA 73 1C 76 58 CE AA B5
AC E2 E0 D9 28 3B 74 76	40 58 52 B4 E9 1F A1 88	BF B2 02 F1 46 B0 7B FD
AE 2F A1 1E 4D 1C 64 70	D0 CE 05 91 62 CC 84 72	89 81 B7 67 69 75 3D 92
57 44 6E EC 9F 43 E4 5E	45 F8 B4 97 1F 24 01 50	A9 A9 DB 35 CD 3B BC ED
C0 05 4B A0 E9 92 CD C7	F0 08 58 53 A7 E9 01 F5	39 78 E1 E1 FE 92 91 3C
9F 24 13 0B 77 07 CD 49	79 96 52 2F 3A 57 0F F5	20 CF B9 9D A4 18 C8 E3
30 18 75 C3 E9 0E 6C BC	03 02 34 7D C7 17 6D 71	E6 8D BB C7 F2 C3 07 2ª
57 1A 0C 45 BF 04 BA 7E	DD CF 1F 31 78 3D 71 26	30 CB A5 8B 9D 62 26 98

Ronda 3

K3	LT-CHA[3]	LT-CHA[3]
C3 92 72 85 71 DA 82 E3	72 E5 BE C8 E1 80 23 51	B9 B8 88 34 65 EB 66 45
7A 5B D4 C4 2F 3C EE 42	6D 07 94 73 0D 2C C1 D8	B6 BA 21 F3 07 48 3B AE
0C 53 DD A4 A5 B5 25 08	E5 77 84 B3 A9 28 70 55	29 74 F0 42 E5 E5 D2 DF

22 A8 A4 9A 9C 1C 5C 92	35 E8 4C 59 2D 88 30 5D	12 B5 4A ED 63 C6 5E CD
E6 72 38 9C 45 B3 64 BE	DB 9F B8 06 AC 0D F3 20	9A F7 6C 31 A9 C3 A8 C5
5A 9C 32 DE 02 6D 88 CB	CD 1F 66 F5 C0 65 67 83	89 A1 97 53 B6 5E AC BA
AE 48 C5 BE 1B DD 45 A3	7C 9D D0 F2 C6 50 63 4A	D1 86 10 E5 C4 E1 67 2B
49 FA 25 7B A2 A4 3C 08	10 F1 A8 E0 45 E0 0C D1	00 CC A6 EF 59 29 D0 FD

Ronda 4

K4	LT-CHA[4]	LT-CHA[4]
A3 E9 8D 1A 2A 65 8B B8	3D 03 37 47 67 68 A6 F9	01 07 B6 32 F9 7E 02 CF
24 9A 37 A3 7D F7 38 B4	AB 87 67 AE 96 92 5A 28	1B 3F 2D 86 29 4F 03 65
CF D2 82 6D F5 41 30 2D	64 9A 95 75 A1 4F 2F 50	AB 19 3E 58 25 4E 84 01
02 C4 CA 67 FE F5 31 E4	A0 B6 2C 6E C5 A2 BA 16	A9 5C 66 03 C5 95 02 AD
22 6A 59 12 B3 0D 87 8E	64 FB B7 9E 2D 94 F2 A6	56 E7 F4 6B CC 06 CF 91
60 E4 D5 0F D1 F4 5C FE	E2 08 3F 54 6B 4C 73 4A	BF 8A DD 1B CB E0 EE 71
D1 22 04 AB A3 27 DA 00	10 5A B8 59 FE C1 DA FE	A7 4A 9E DA 1B CF EC D2
08 6D 12 6C 50 AA 39 8F	27 11 74 C5 63 16 9C 00	6E 15 BE D0 58 2F 7B 89

Ronda 5

K5	LT-CHA[5]	LT-CHA[5]
98 4A A7 0E 3F 0F 02 A4	5B D2 EB 8A DA C6 A2 5B	14 F0 62 CA 6E CB 4C 02
57 AD 27 C1 A6 F9 A0 4F	80 C1 81 9E E8 C8 AE 61	FE D2 02 2E CE E3 47 29
EA 37 57 B4 C1 F9 FD 1E	70 16 C7 A4 B7 64 C4 E6	E3 5B 13 F6 DF AE C4 98
07 70 68 7C D4 BD 13 15	07 DD C1 EC 86 99 A9 8B	E7 63 96 94 E7 00 19 1C
A1 94 C1 23 1A 1A 69 49	C7 FE FC 77 7D 4A 24 9A	F5 2C D9 51 E6 AE 4A D4
94 D0 AF F1 67 45 67 76	D9 98 B8 08 E7 5F 5E D9	BC 23 F6 7F 2A DF 3C 60
D8 F4 7B 63 47 B7 89 B8	DC 90 FA 5B A7 A2 FB 2D	E7 6C 75 D3 85 3C 84 47
A5 4D 53 7D BE A2 67 B5	36 00 37 DC 6A 66 1A 82	8F 9C BF 1E 06 44 83 1C

Ronda 6

K6	LT-CHA[6]	LT-CHA[6]
EF 4F 92 6F 89 2C C7 EF	D9 D8 CE C2 39 40 34 24	D6 A0 01 50 14 9F 3F 74
B7 A9 1F 28 46 29 5E 5F	BB FF E0 B2 2A F4 4E 39	ED F1 E8 29 2D 12 3C F4
0C 3A DA EB 64 3F 34 68	4D 33 89 B8 CF AB 32 02	78 23 83 66 04 68 01 3A
18 C3 B4 D9 09 B8 1C 22	07 D8 22 99 AB 39 CC C1	A3 23 F7 A3 8D D9 99 94
96 1E B1 DF 3F CC 2F F7	30 F7 A2 19 BA 9A B6 F5	5E 99 1C BB 83 55 BF B4
E6 CA 08 87 B7 B9 28 E1	A7 D2 D8 41 41 BC 50 75	5A D6 4F 78 3A 62 A7 00
05 AD 0A 94 5E 0E 45 AB	1A ED 3D 7C A6 9B 32 B3	AA 4E 1B 89 79 20 65 FF
A9 20 38 54 13 A4 E3 D3	5A 5B EA 55 7A 67 E3 66	7C 83 EE C2 B6 64 76 13

Ronda 7

K7	LT-CHA[7]	LT-CHA[7]
51 06 8B 94 54 89 33 9C	DB 15 51 2C 47 26 9F 17	9D 8C F3 E7 17 57 A2 85
C0 B3 31 5F BC 44 FC A1	43 12 F4 D8 EE 6C 95 DB	7F 88 1C 1B F6 40 BE 33
BA EC B3 98 0D 26 0C 8A	EE D4 13 E0 DC 6C C7 75	78 E1 23 9A 59 A0 BB 3A

62 48 F1 B7 3E 1F D4 06	12 F6 08 15 1C 70 8F 70	F6 77 9A 4C 26 6A 0A E4
12 E5 76 B4 86 E4 2A 9C	B6 E7 AB 62 97 A4 E1 FD	74 EC F4 77 C0 0E 89 BE
FA 37 40 74 64 D7 B2 06	E3 AB A0 C7 63 D9 51 1F	09 B7 6C FF B9 A2 43 C2
5D E4 45 F2 C7 79 49 DF	47 C4 B6 E5 24 81 C2 18	5D BC EA 04 15 DF EB 2D
A7 C3 D6 E2 4B 4E 9C 73	11 5C B5 D6 6A 82 BC 63	A1 DD 9A F9 99 44 29 B6

Ronda 8

K8	LT-CHA[8]	LT-CHA[8]
49 4C AF 0F 18 14 4B AE	3B 8C AF 4D B2 9A A7 62	0B A2 7D 1A 9A 60 A6 BB
31 01 49 54 FC 6A E4 CA	21 D4 DD 41 74 79 AA 01	98 D6 5A B2 3C 61 2E AB
09 FF 41 F2 5C 6B F9 8D	13 47 D6 C5 A3 8B 90 47	C8 6E 5D 7C B5 26 1A FF
4B D4 D6 CE A4 00 3C EE	BC 67 FD FF 26 54 AD DD	A0 18 C8 0A 16 65 7E 7F
DB 6D 3F BD AE B4 23 29	28 0D 12 FF 85 2E 7F 29	F4 59 0D 11 AF 0D 2C C9
88 0C E6 14 36 7D 13 C7	88 A6 7F C5 FC A4 E6 67	A6 04 52 DA 44 83 B2 59
98 7A DD 5B C4 A5 8C 1A	AE F7 D5 58 81 DA C5 32	7A 8F 55 8A 73 F0 D2 3D
CA FA 26 04 16 F5 6B C6	42 D6 D1 88 10 49 2F E2	F1 29 10 ED 96 59 93 1D

Ronda 9

K9	LT-CHA[9]	LT-CHA[9]
9B D6 42 05 CB E2 18 29	33 40 17 78 1D F3 50 3A	C8 8E C7 3C FF 62 1B 9D
AE 2E A7 37 03 D2 F8 93	D3 AB 64 A9 71 CC 65 00	EA 89 61 D7 31 83 64 99
C6 13 72 B1 C9 14 A1 7C	6B 9D DF 36 4E AA DC 3A	38 A5 84 6E A4 07 13 A9
04 72 DD 01 B4 1D C0 7E	6B 72 7B 5C 73 2F C8 F0	44 E0 47 79 50 02 4B E7
B7 F1 62 FB 66 DF 33 F5	93 7B 2C 27 DF 97 D4 BC	14 8B 00 76 9F 87 EE 1C
E2 2D 57 42 43 07 86 C3	6A 70 5E 6D 38 51 18 3C	71 A9 CD 21 BD E4 DA 8E
0D 27 78 48 DA 05 2A F6	84 C6 1E E7 8C 98 26 88	99 32 02 41 55 D4 9D 3F
AF E5 A5 65 30 E2 C8 53	17 71 4A 5A A9 3F 5C CD	0E 67 65 58 BD F7 B9 63

Ronda 10

K10	LT-CHA[10]	LT-CHA[10]
DC 7A 7B 7C 41 75 B6 90	B2 D6 5B 27 3E BC C5 31	66 B0 AC CB 1C BD D1 48
3D 8A 81 61 B1 EC 4C 74	07 23 7B 78 63 47 75 92	6C CB 8A A1 67 33 90 B5
82 94 E8 65 15 0A 88 0E	2A 7E 63 1B 37 88 25 C3	EF F9 24 BF 74 EC 50 13
C7 10 E1 22 79 5E 78 26	0F A5 16 0C FE 0C AC 82	DC 2F E3 86 15 4A 73 07
E1 6B CE 1C 07 83 7B 61	22 48 E9 B7 B5 D7 59 A7	46 6C BA 18 3D 80 7A DE
9A C5 9A 13 D4 49 F1 3C	88 85 DF B0 B0 DF 25 75	85 24 4A 4F 9D CA C2 E3
3F 3D 5D 44 C4 E8 74 18	24 67 59 AD 1C 10 C6 08	4B 36 CE E1 A3 47 49 F5
32 83 89 2D C1 8D CA 2C	85 37 E4 15 10 FD 8A 6A	25 3F 60 B7 1D 49 2F 75

Ronda 11

K11	LT-CHA[11]	LT-CHA[11]
4C 64 4D DB 43 1F 2F E5	59 78 80 09 1C DB 0D 35	24 72 9B 38 17 AB 01 FA
D7 DE A5 D0 63 20 F7 F9	19 2A 50 5D DE 24 32 90	68 28 16 6F 0E AE 0A D8
CE 6D 8F 63 8A EC D3 77	E0 23 32 C6 E6 27 FC BC	98 DA 7E 94 CE FD 5F 33

A0 7A B6 B6 49 49 9B 9D	7B 31 7B 49 97 BD 00 A3	06 A3 D3 31 20 26 73 8E
7F 36 05 6F 50 3A 95 7D	12 21 14 **56** 6A EA 1B 69	5B 73 27 **56** DB 86 08 5F
39 C2 07 A0 0C 35 DB 28	BD 4B 05 B3 C9 30 E**2** CA	E7 13 42 C8 8D EA 2**2** 1B
EA 4C C4 7C 6D FA EB 00	E0 46 D4 **3**C 9A B5 5A A3	6A D1 6C **3**0 C7 58 13 58
07 9A 5D 15 37 AF 9E 18	AF 4E D**8** 7F 90 4C 98 95	91 12 9**8** EA FF 92 71 FA

Ronda 12

K12	**LT-CHA[12]**	**LT-CHA[12]**
14 C7 29 E2 1E 4F 97 11	5E 81 E8 E8 53 39 0F 11	8B 3F 47 0D C8 18 2A 40
C9 CD 61 15 BB 33 1D 82	42 69 9D D0 3E 94 25 4**0**	34 76 D3 49 A7 11 1E 8**0**
6E D4 86 00 D4 25 BD 90	8E 71 90 2**5** 20 B0 B9 5A	6A E4 72 0**5** 83 DA 43 34
CA 27 F8 17 84 C1 77 01	FE F**B** AB E1 D0 D5 C1 C4	E1 5**B** 7E 56 96 92 87 36
68 7F 71 03 84 76 1B CF	61 **C**E C5 42 A8 31 C8 85	56 **C**6 F2 A4 09 65 8F CD
80 84 26 F4 04 0E 80 EC	D4 6E 7D E0 34 29 8A 64	1B 13 F0 C4 EE E8 10 30
51 F5 E0 59 8A E0 96 81	D3 5B 3**1** EB **2**4 2E 4C 47	1A 06 1**1** C9 **2**6 7D 06 9D
D3 47 1E E1 35 93 30 AE	8C 1C 1**D** C7 25 3**E** 40 46	4B BF F**D** 0E B3 7**E** 8B CA

Hash resultante de procesar el primer bloque del mensaje

LT-CHA(Bloque0)	**LT-CHA(Bloque0)**
5E 01 E8 E8 53 39 0F 11	8B 7F 47 0D C8 18 2A 40
42 69 9D D0 3E 94 25 4**0**	34 76 D3 49 A7 11 1E 8**0**
8E 71 90 2**5** 20 B0 B9 5A	6A E4 72 0**5** 83 DA 43 34
FE F**B** AB E1 D0 D5 C1 C4	E1 5B 7E 56 96 92 87 36
61 **C**E C5 42 A8 31 C8 85	56 **C**6 F2 A4 09 65 8F CD
D4 6E 7D E0 34 29 8A 64	1B 13 F0 C4 EE E8 10 30
D3 5B 3**1** EB **2**4 2E 4C 47	1A 06 1**1** C9 **2**6 7D 06 9D
8C 1C 1D C7 25 3**E** 40 46	4B BF FD 0E B3 7**E** 8B CA

Procesando el segundo bloque se tiene como valores iniciales los siguientes:

K0	**LT-CHA**	**K0**	**LT-CHA**
65 83 30 E8 5F EC DD 81	DE 01 E8 E8 53 39 0F 11	B0 FD 9F 0D C4 CD F8 D0	0B 7F 47 0D C8 18 2A 40
41 4C 06 1C 0F 58 B9 8**0**	42 69 9D D0 3E 94 25 4**0**	37 53 48 85 96 DD 82 4**0**	34 76 D3 49 A7 11 1E 8**0**
9B C2 63 6**5** 31 8B 81 3A	8E 71 90 2**5** 20 B0 B9 5A	7F 57 81 4**5** 92 E1 7B 54	6A E4 72 0**5** 83 DA 43 34
E2 0**8** F6 E1 E0 91 DF 84	FE F**B** AB E1 D0 D5 C1 C4	FD A**8** 23 56 A6 D6 99 76	E1 5**B** 7E 56 96 92 87 36
4C **5**7 B0 C2 9D 44 A8 45	61 **C**E C5 42 A8 31 C8 85	7B **5**F 87 24 3C 10 EF 0D	56 **C**6 F2 A4 09 65 8F CD
FB C8 8E A0 15 35 9F 24	D4 6E 7D E0 34 29 8A 64	34 B5 03 84 CF F4 05 70	1B 13 F0 C4 EE E8 10 30
C4 A3 F**1** 0B **2**9 9B 8A 87	D3 5B 3**1** EB **2**4 2E 4C 47	0D FE D**1** 29 **2**B C8 C0 5D	1A 06 1**1** C9 **2**6 7D 06 9D
B4 18 B**3** C7 04 5**E** EA C6	8C 1C 1**D** C7 25 3**E** 42 46	73 BB 5**3** 0E 92 1**E** 21 4A	4B BF F**D** 0E B3 7**E** 89 CA

Ronda 1

K1	LT-CHA[1]	K1	LT-CHA[1]
0E D1 C3 59 FA 5D AE 28	C8 26 05 F7 84 58 3A 9F	BC F7 9C 57 B6 C7 F5 DB	9F 02 2D 8A 90 E5 0C EC
2B AF 7C 1F 6B 85 F4 47	42 6E A9 C1 AB 2A 6F 94	DF CF 7D BC 17 79 48 3F	4C E8 A6 BE 91 E7 70 AD
B3 59 7E 4C B8 17 6E 92	FF 94 CF CD 1E D0 A0 92	3D 08 C6 A7 06 6D 4D 4F	04 61 10 74 1E 31 6C 75
23 2E F8 70 91 01 3D 33	A2 66 FB 5B 6C 1D 82 EA	48 36 9C 6B 60 A8 A8 97	42 B6 4B 72 08 A0 D0 5E
4A 51 42 5C 28 4A EC 49	74 95 E5 C7 2B 45 84 47	93 F8 22 6E B8 A5 30 EA	F0 99 36 96 A3 F3 16 EC
5D 53 06 35 1D 28 F9 8A	BE 68 ED 43 06 73 40 FE	B2 BC E9 9C BC F4 47 92	BB 48 04 F7 1D F9 1C C3
D7 F4 1C 72 32 4C C3 B4	F3 B8 44 0B 0B 7C 25 37	FE F7 D8 7C 9F C0 C9 49	64 54 53 F2 DD 63 B4 02
F7 AE 5B C4 43 F5 F5 0E	D6 63 96 61 79 F2 EE B3	0D 45 E4 16 37 97 08 16	2B DB DB 82 B8 F7 10 C4

Ronda 2

K2	LT-CHA[2]	K2	LT-CHA[2]
F9 72 53 D6 B1 9A 5E EF	33 BA D6 77 70 37 DE 1B	8F F8 B3 E4 93 02 0A 98	E4 CD EF 83 81 39 36 84
1E BE EE CD 6B FF A4 1B	0D 6B BF 6C AD A8 F7 4F	6B F6 7A EC 00 74 4B AA	2A 40 F2 F9 D0 AC 00 50
5D 58 BB A2 28 B7 73 E5	D7 98 5D 6D F7 F2 F0 7D	4C 6E 56 F0 43 C7 B4 9A.	34 A7 5A E8 52 CE C5 CB
72 28 97 35 1D 78 D2 94	B9 6D B4 10 18 70 5F 5E	71 E7 F9 33 99 C2 87 0D	56 84 1D A7 5C 17 A4 A9
E4 32 28 52 55 F6 19 29	D0 1B 1F A3 DA A9 2C 22	D2 D1 B4 B6 9B 9B 3A 21	10 6E E7 67 0F 33 12 BC
EC 91 78 D1 2F F3 B1 61	94 95 94 1A 2C 14 CA 0D	7E A2 AF 39 5F 5E F3 2A.	1D 27 76 1C DD 41 AA F3
F0 A9 DE 60 4E AB 60 8B	24 A4 B6 E2 F4 75 96 53	2A 0D 4C 49 25 8C 43 91	ED B7 CE D4 08 90 13 0B
7B B5 FE CB 38 1E FE 78	D2 18 B8 3F FA 15 AB 51	32 6E A4 B8 1A 13 A5 5F	F3 D3 23 21 C3 4F CE D1

Ronda 3

K3	LT-CHA[3]	K3	LT-CHA[3]
86 E7 3A DE 70 63 F6 CD	7C 0B 4A 05 89 B3 A1 D2	DE D7 6A B4 E7 A9 C1 A9	09 A3 C8 35 7B 44 60 15
0F F4 0F D6 5E 93 1C B6	B4 D6 EB 25 41 57 4D CC	30 D9 13 DF 5D D0 44 05	ED EE 55 5F 64 78 BE 78
BD A1 41 23 71 D5 EE D2	B7 6D F4 84 7E 3E 3A 4B	B5 F4 D8 56 A6 C0 CC 31	C4 99 C4 D2 2F 52 DA A6
AD B5 2B 0D 30 05 41 19	51 FD 9B ED 2C 8E 6C 23	D3 E7 85 54 48 7F FA AF	34 B8 C8 6D F7 A0 F7 95
E1 5A 72 02 EC E8 70 96	99 65 AB AA F0 7B AB 68	A3 E5 75 E0 1E 15 EF C8	7A 9B A2 2B 07 B4 0D D1
8C 00 6D C3 7C C1 E8 27	DF 64 58 C3 44 2D A7 66	D4 7A 51 B3 5E BC B3 81	5D BE 50 3E FF 48 0A 03
0B 41 EE 46 32 EF F8 D2	26 7E 41 66 62 E1 D8 36	34 14 69 F4 13 45 3A 1F	B5 50 3C B6 86 3B DE 21
66 94 8C F9 B7 63 67 BF	66 C9 2D AA FB B8 EC DA	F7 BC 0C 51 22 66 9B 89	19 49 54 C9 DB 1F EB B0

Ronda 12

K12	LT-CHA[12]	K12	LT-CHA[12]
54 09 AE 7A 73 0B B8 8E	3F 9A C7 AB 0E FA 38 9B	B5 78 34 D6 2A B5 14 5B	16 4A 07 5E BD F3 BF 89
42 7D CC EC 81 6E A8 F2	AB BA 2D 20 B3 16 C1 B0	8E 6B 09 AE 60 5E 9D 42	9F 03 BE 7D 29 0E 7E A0
B9 AA 55 77 32 FF 8D AD	4D 9F DC 5F 7E 50 BC 1A	87 68 67 0D 26 B6 FA 61	A4 D6 30 01 3D 58 9E 0A
39 58 0D B7 44 DD 53 1F	EF B2 6D 8B 67 29 66 E8	AC 75 23 02 DD B3 AE 70	11 12 0B 53 AC 5C 1E 77
12 E1 CF 1C 34 59 40 8F	C9 08 67 0F 0E 61 2C 84	83 2F 3E 23 03 31 1E B6	46 26 A4 3D 6F 08 AC 7F
E9 4F A6 F2 23 E3 18 8B	A2 ED 4A AA C4 BD C0 28	50 B8 ED 6E F6 43 64 70	52 11 16 BC 44 9C 2D 56
E5 D7 A3 DC 4B 9D 75 25	02 E8 3E 65 50 38 6F C8	55 19 87 3B 61 96 8D FD	CC CE 18 6E 5A 64 7B 53
88 4C 4E 04 48 EC DE 02	1A 36 95 CD 8F 74 97 82	A2 90 C8 C1 04 6F 53 1F	7C 29 19 AD 80 8D D4 19

Hash final del mensaje

LT-CHA(M)	LT-CHA(M)
E1 9B 2F 43 5D C3 37 8A.	1D 35 40 53 75 EB 95 C9
E9 D3 B0 F0 8D 82 E4 F0	AB 75 6D 34 8E 1F 60 20
C3 EE 4C 7A 5E E0 05 40	CE 32 42 04 BE 82 DD 3E
11 49 C6 6A B7 FC A7 2C	F0 49 75 05 3A CE 99 41
A8 C6 A2 4D A6 50 E4 01	10 E0 56 99 66 6D 23 B2
76 83 37 4A F0 94 4A 4C	49 02 E6 78 AA 74 3D 66
D1 B3 0F 8E 74 16 23 8F	D6 C8 09 A7 7C 19 7D CE
96 2A 88 0A AA 4A D5 C4	37 96 E4 A3 33 F3 5D D3

Debido a este comportamiento se decide dejar de lado la permutación Baker para procesar el bloque del mensaje dejando una permutación del bloque LT-CHA, la cual corresponde a una permutación circular de cada una de las columnas puesto que al realizar una permutación circular a nivel de renglones se tendrían unos resultados similares puesto que no hay combinaciones con las columnas ya que sería un mapeo similar al de Baker.

6.3.2 Colisiones en un subconjunto del espacio de prueba.

Como mencionamos anteriormente para que una función hash sea considerada resistente a colisiones -que dos mensajes diferentes tengan el mismo hash value- el tamaño del espacio de mensajes en el que ocurra esto debe ser de $2^{n/2}$, como fue mostrado en el ataque del cumpleaños.

Debido a que existe el problema del almacenamiento de los pares <msg, hash>, se opto por un espacio mucho menor cuyos elementos son las palabras contenidas en diferentes diccionarios; cada palabra es procesada para obtener su respectivo hash value calculado con LT-CHA, posteriormente es almacenado en una base de datos en PostgreSQL en una tabla en la que el hash value es la llave primaria de la misma, con el objetivo de verificar si se tiene o no una colisión en dicho espacio al romper la validación efectuada por el manejador. En caso de que exista dicha colisión se verifica que no corresponda al mismo mensaje de entrada, puesto que se tomaron como entradas listas de palabras de diferentes materias en las que algunos de los términos podrían coincidir, en caso de que no sea la misma este par <msg, hash> se almacena en otra tabla a fin de mantener un mapeo de las colisiones entre diferentes mensajes previos.

Para la prueba mencionada se procesaron 2, 209, 586 mensajes diferentes sin encontrar alguna colisión, cabe destacar que es un subconjunto del espacio teórico como mínimo.

Para finalizar se ponen como ejemplos de los hash values calculados (los cuales van entre comillas) con LT-CHA los siguientes:

“tinúviel” =
02101595D69A6731 236EF7830E22E896 3ACDAA332A451107 7B5B209C8F3E55EE
9C2E8CCE695BB072 1AA6473AB6FF6065 C7D05243025888FF ECF1F375B7025009

"Lúthien-Tinúviel" =
056DAB44975B7BF9 49A60FD73023662A 0D7FC74AB522D17C D67085EC2D14897C
BDD5A100A466FE3E 5AC7718D3C58DA72 685BF5CD3FB8B6B4 08DEEA7D61E65F5C

"Tinúviel" =
1C6411B45A29CD1F 6508360F65F38716 F76F81EF77F9A1A3 03385B3F2E251085
B828920E2C00D503 3835B7A29C5AB6ED EC794885D1D79856 6D5A982B A5A15171

"luthien-tinuviel" =
386C4C29D26D1F81 0CD7CF6BC45FCC73 20DCC31E04A57939 134FFC6A8CC805E3
44729B4793AE07ED F9E0F7B86D4CE1B7 D24F9CDF72A81A64 1D3D6E3B70B1ABC7

"Tinuviel" =
53AA723F0E7B9AB1 852CAB5D6E84A1A3 3D1CD62C0A3DD20E 2EDFB6DE3E427576
B3E3EF59D3B2DEA4 FBD45CF55C95FA37 3EA5E6623840DEDC D64A1BD96337447F

"lúthien" =
6BCCD1DBD4075993 9106C7D1D98CD207 641ADA0B73FC9892 95206AE61F0CB577
18EF3B3ED3D84981 FBBBF4DAAA1425BE 9979FD2F59A2BA62 D559E75599D27193

"Luthien-Tinuviel" =
6C0F39F08842119E 499159726EEB54E4 A7301DDC57E72718 A0A6BB8392ACB656
9F21D76CD6B8F2AD 37171C566DF6D5E3 E8BD2B0C0628B680 1E3CFE388B86E83A

"lúthien-tinúviel" =
8C18E31EBE17D303 A57AAF598BDF8486 E62B9B9CDF9BF287 256F5EABD967FBE5
9C574BBE470CD995 D17C8FF4E204F359 AD8D319A1444C542 47384BAA20AEAA8C

"tinuviel" =
B3EA24A1FB769224 48082B800E3D50A5 1831C394B5A63671 413BEC55FE570FFD
B4F79D8D5C55E209 0CCE8F952CE70653 7FBDD40AEE19193B E6D41C6085DA6F39

"Luthien" =

E2330CEDCFD3BB12	B189EE0723F2E8CD	C97BF93AEC71EBB8	0E88B138E7AD9240
7D4D1D74A9A20EF7	88427CF82FA9AC5B	421516DF408A3AE1	AEFC339F30C44FF0

"luthien" =

EBC75A23B88852DD	0013576DF1EB745B	DFD68CE8B8612563	C3E6BE2F90186055
FBF3D63446643FF9	4F2766A46C47E66D	670EE25C1273DC89	CE5D3CA8967FD337

"Lúthien" =

FE2E06D93E6E1D72	3A624FEECF450071	7BC2C8449F942E34	ECB4F6E1775974FA
7ECC40EA67221686	A2305DF8EC263876	3B02B0B0B8F86723	2ED303E2E1F7785F

7. Conclusiones

Comencé a escribir esta tesis con la firme convicción de que era posible crear un algoritmo simple para la generación de un valor hash el cual tendría como base y núcleo principal un objeto fractal sobre el cual giraría todo el procesamiento.

Creí pues, en un principio que dicha afirmación era cierta y que podría incluso ser que el algoritmo que describiera al objeto fractal fuera variable o dicho de otra forma fuese configurable, a fin de dejar todas las variables de estado configurables y dejar solo establecido de forma fija el número de pasos y el procedimiento a seguir para el cálculo del valor hash.

Sin embargo, a través de la investigación y de mis primeros diseños de la función hash basada en esta idea, encontré que dicha premisa no se cumple del todo, principalmente en la característica de sencillez y simpleza, al menos no en los términos que imagine en un principio. A continuación explicaré porqué.

En primera instancia, encontré que el principal problema a resolver es la discretización de un espacio lineal de valores, que a su vez sea lo más amplio posible y que la definición del objeto fractal pueda ser distribuida uniformemente en dicho espacio, cosa que no siempre es factible un ejemplo claro de esto es el conjunto de Cantor que se describió durante el desarrollo teórico, como se mostro en su momento cada vez que se itera sobre dicho conjunto se pierde un tercio del mismo.

Empero, se reduce el espacio de valores en el que es posible definir al objeto como un fractal, por lo que si se define un algoritmo que utilice dicho objeto fractal debe considerar el caso para todos los puntos que no son definidos por el objeto fractal, una solución posible es considerar a dichos puntos como una órbita adicional al algoritmo fractal en la composición de funciones, empero, esto complica el planteamiento de que el algoritmo sea simple.

En segunda instancia se tiene el problema de la exactitud en los cálculos de los objetos fractales, debido a que la mayoría de los algoritmos que definen a dichos objetos fractales,

utilizan valores reales, los cuales dependen de la capacidad de representación de la máquina así como de la arquitectura. Además de que las operaciones de punto flotante son costosas en cuanto a instrucciones a nivel máquina lo que va en detrimento del rendimiento y la eficiencia, por lo que siempre éstas se encontrarán en desventajas contras las operaciones que utilicen sustituciones, permutaciones y corrimientos como en la mayoría de los algoritmos de función hash que existen actualmente.

En mis primeros diseños de la función *Lúthien – Tinúviel Chaos Hash* mi idea base fue que el núcleo fuera un objeto fractal dentro de los definidos del conjunto de Julia y que el mensaje de entrada fungiera a su vez como la sal del algoritmo, de tal manera que pudiera dejar al objeto fractal que se calculará o iterará las veces necesarias o por mí definidas a fin de calcular los valores de los parámetros y posteriormente solo efectuar la suma *XOR* con el valor de entrada previa una permutación.

Empero, dicho diseños tienen el defecto del cálculo de los valores del objeto fractal, puesto que dependen del grado de precisión en operaciones y representaciones de coma flotante por lo cual debía ser acotado, una solución se basa en tablas de definición de correspondencia de rangos pero esto por si mismo complicaba los cálculos y hacia infactible un algoritmo eficiente en tiempo ya que se debía definir una ventana de calificación para cada rango.

Otro de mis diseños estaba basado en la integral de línea, en la que se definía una función hash cuyo núcleo fuera el objeto fractal "***X***", el cual producía valores determinados en coma flotante y a su vez una función de integral de línea, acotada a un rango de valores en la que se sustituían los valores obtenidos a fin de mapear dichos valores en el espacio definido de la integral, si dicho valor cae en el rango definido se tomaba dicho valor como parte del hash intermedio en caso contrario entonces dicho valor se consideraba como sal, para la siguiente iteración del objeto fractal, el problema de este diseño radica nuevamente en los cálculos de coma flotante y en la definición del objeto fractal a fin de que para la mayoría de posibles valores fuera definido, que a su vez el dominio de la integral de línea lo incluyera.

Como vemos estas dos instancias son por sí mismas dos limitantes para la característica de simpleza y eficiencia del algoritmo de función hash. Una solución a estos problemas es el pre-cálculo de los valores dinámicos y la especificación de un algoritmo que defina la

posterior generación de valores, otra solución consiste en la utilización de mapas caóticos bidimensionales, en los cuales para cada punto en el plano en el que es definido es mapeado a otro, siguiendo una regla de correspondencia basada en un algoritmo predeterminado, uno de los mapas más utilizados es el mapa caótico bidimensional de Baker utilizado en algunos algoritmos de cifrado caóticos.

Es por ello que la opción utilizada para la generación de constantes en mi diseño final de la función hash caótica Lúthien-Tinúviel está basada en dicho mapa y además se elige el pre-cálculo de dichas constantes a fin de determinar de manera única y fija su representación en hexadecimal para cada una de las rondas, junto con la definición de un algoritmo que permita a su vez la generación de constantes dependientes tanto del mensaje de entrada como de la variable de estado que define previamente a la constante anteriormente generada.

Como se menciono antes las funciones hash son importantes puesto que son una herramienta que nos permite verificar la integridad de un mensaje determinado, sin embargo, el conjunto de todas las salidas posibles de una función hash es un subconjunto pequeño del universo de posibles mensajes, por lo tanto existen y existirán siempre colisiones entre diferentes mensajes.

Puesto que las colisiones son una característica inevitable de estas funciones una de las propiedades en un buen diseño es que el espacio en el que un valor hash colisione debe ser lo más amplio posible y que la diferencia entre dichos mensajes que provocan la colisión sea lo más significativa posible, cuando esto no ocurre así entonces se dice que se encontró una isla de colisión la cual servirá como punto de entrada para los ataques basados en los llamados puntos fijos, ya que en dichas islas la probabilidad de encontrar uno es mayor debido al alto grado de lineabilidad en dicha región.

Una forma de evitar que este espacio sea fácilmente determinable los algoritmos actuales de funciones hash se basan en operaciones de corrimiento, sustitución o en operaciones algebraicas -utilizando los denominados campos finitos de Galois- generalmente los algoritmos que hacen uso de estos últimos se basan en el diseño de una función de cifrado, cuya característica particular es que ésta no es invertible.

Una característica importante de una función hash como vimos a partir de los resultados del primer diseño de la función LT-CHA es que el conjunto de salida resultante, el cual está acotado por la longitud del valor hash, necesita tener una variabilidad entre dos mensajes semejantes que varíen en un solo bit a fin de evitar lo que nosotros denominamos espacios periódicos de colisión, en otras palabras bloques de mensajes cuasi-semejantes cuyo valor hash presente un comportamiento casi lineal el cual facilitaría un criptoanálisis lineal o diferencial basándonos exclusivamente en dicha variabilidad para facilitar una colisión.

La forma más fácil y cómoda para discretizar la trayectoria de un mapa logístico es efectuar un mapeo de los valores posibles con su correspondiente representación discreta una operación semejante a las S-box utilizadas en algoritmos de cifrado.

A partir del análisis de nuestro primer diseño de la función LT-CHA, el cual fue puesto en la sección de análisis de resultados, se utilizaba el mapeo Baker modificado tanto para el procesamiento del bloque LT-CHA como para la generación de las constantes en turno, obteniendo como resultado que al aplicar dicha operación de esa forma se generaba un conjunto periódico de colisión lo cual era un problema.

La causa es que el mapa Baker por sí solo no asegura una buena distribución del mensaje puesto que no hace una difusión a nivel de columna renglón sino de secciones por lo que fue necesario cambiar dicho diseño manteniendo esta distribución para la generación de constantes dinámicas.

Para el procesamiento del mensaje se opto por utilizar una sustitución basada en una operación de difusión de columna renglón del bloque LT-CHA, con lo que aseguramos que un pequeño cambio en el bloque LT-CHA genere una nueva difusión dependiente de las columnas y renglones.

Nuestro diseño como se menciono está basado en la función hash Whirlpool integrante del portafolio de algoritmos criptográficos de NESSIE, como sabemos para la sección de la generación de constantes de LT-CHA se utilizan valores predefinidos -a manera de semillas- los cuales fueron calculados a partir del mapa de Lorenz.

La utilización de mapas logísticos para la generación de constantes es relativamente impráctica debido a que existe el problema de la significancia de los términos, en los cálculos (ya que como sabemos dependen del grado de precisión en su representación) lo cual es un problema que de momento quedo abierto en nuestro trabajo.

Unas soluciones que fueron vislumbradas para la discretización de una trayectoria real a un espacio determinístico discreto son las siguientes:

- La definición de reglas de mapeo de rangos calculados para las trayectorias de mapas logísticos a valores discretos (o incluso rangos estándar) como parámetros para la siguiente iteración en la generación de constantes.

- La utilización de la integración de línea para la perturbación controlada de una trayectoria en un sistema dinámico a fin de evitar la dispersión o contracción en la trayectoria evitando así un mayor grado de precisión en la representatividad.

Las ideas que se proponen anteriormente chocan directamente con el concepto de caos matemático, debido a que tratamos de controlar una trayectoria tratándola de volver determinista, la idea en si no es determinar la no lineabilidad de la dinámica de un sistema sino la de generar perturbaciones que podríamos llamar o denominar “benignas” que nos permitan representar de una mejor manera a la función en un sistema finito determinístico discreto.

Una conclusión importante generada como resultado del presente trabajo de investigación es que el diseño de una función hash debe contener una fase de preprocesamiento en la cual se incluya lo longitud del mensaje original como lo proponen R. Merkle e I. Damgård y que a su vez se debe definir una longitud máxima para el mensaje a procesar, ya que con esto se evitan ataques de colisiones basados en inyección de bloques para la generación de colisiones intermedias.

Si bien es cierto esto no previene de otros ataques si permite un mejor diseño evitando los basados en la longitud, esta es la principal razón de que para algunos algoritmos se establezca la longitud máxima del mensaje a ser procesado, como es el caso de Whirlpool, SHA-x y de LT-CHA, como contraparte de las funciones de la familia MD-x.

Una conclusión derivada del análisis realizado a cada una de las rondas que forman parte de la función Lúthien – Tinúviel, es la siguiente: la utilización del mapa Baker tanto para la generación de llaves de ronda como para el procesamiento del bloque LT-CHA, produce para ciertos rangos de valores del bloque de entrada las llamadas islas de colisión en la que la mayoría de los bits son iguales entre los valores hash resultantes.

Los rangos de valores de entrada en los que se presentan dichas islas de colisión para el siguiente espacio muestral definido "*El conjunto S formado por todos los hash values de las cadenas de 512-bits conteniendo un solo 1-bit*", son los siguientes:

- Los 1-bits en el rango de posiciones de 8 – 15 y 24 – 31.
- Los 1-bits en el rango de posiciones de 80 – 87 y 128 – 143.
- Los 1-bits en el rango de posiciones de 216 – 223 y 248 – 263.
- Los 1-bits en el rango de posiciones de 304 – 311 y 336 – 343.
- Los 1-bits en el rango de posiciones de 360 – 367 y 376 – 383.
- Los 1-bits en el rango de posiciones de 392 -399 y 408 – 415.
- Los 1-bits en el rango de posiciones de 464 – 471 y 496 – 503.

Al cambiar la definición de la función LT-CHA en la fase de combinación lineal y de permutación en la generación de constantes de ronda y en el procesamiento del bloque se evita la formación de estas islas de colisión además de lograr una mejor distribución estadística con respecto a las medidas estadísticas estándar.

8. Apéndices

8.1 RIPEMD

El diseño está basado en MD4, esencialmente consiste en que existen dos versiones paralelas de dicha función. En 1996 Dobbertin encontró una colisión para RIPEMD reducido a dos rondas; esto implicó un rediseño de dicha función teniendo como funciones resultantes a RIPEMD-128 y RIPEMD-160.

Internamente la función consiste de dos flujos que son realizadas en paralelo. Cada flujo es una versión modificada de la función hash MD4 (sin la función de retroalimentación), consistente de 48 pasos secuenciales en tres rondas. Las diferencias con respecto a MD4:

- El orden en el cual las palabras del mensaje son aplicadas en rondas 2 y 3 son diferentes.
- Las constantes de rotación circular cambian.
- Un flujo utiliza las constantes de ronda de MD4, para el otro son diferentes. Esta es la única diferencia entre los dos.

Al finalizar, la salida de cada uno de los flujos es combinada junto con la variable de encadenamiento de entrada. La manera en que se combinan depende de la implementación especifica del algoritmo a continuación se muestra la definición de RIPEMD-128 y RIPE-160.

8.1.1 RIPEMD-128

Las características principales que definen a esta función hash [5] (pp. 1-ss.) son:

- **Entrada**: cadena de bits "*x*" de longitud arbitraria $b \geq 0$.
- **Salida**: un *hash value* de 128 bits de "*x*".
- **Tamaño del bloque**: 512 bits.
- **Número de rondas**: 4

8.1.1.1 Definición de constantes.

Se definen el vector de inicialización (las 4 palabras iniciales de 32 bits) *IV* denominados buffer MD.

$h_1 = 0x67452301, h_2 = 0xefcdab89, h_3 = 0x98badcfe, h_4 = 0x10325476$.

Nótese que las primeras 3 constantes del *IV* son las mismas que en MD4.

Debido a las características del algoritmo (contiene dos flujos paralelos) las constantes aditivas definidas son:

Ronda	Izquierda	Definición	Derecha	Definición	Intervalo
1	0	0	50A28BE6	$2^{30}\sqrt[3]{2}$	0..15
2	5A827999	$2^{30}\sqrt{2}$	5C4DD124	$2^{30}\sqrt[3]{3}$	16..31
3	6ED9EBA1	$2^{30}\sqrt{3}$	6D703EF3	$2^{30}\sqrt[3]{5}$	32..47
4	8F1BBCDC	$2^{30}\sqrt{5}$	0	0	48..63

Tabla 16. Constantes de RIPEMD-128

El orden para acceder a las palabras fuente definido (cada lista va de 0 a 15) es:

z[0..15] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
z[16..31] = [7, 4, 13, 1, 10, 6, 15, 3, 12, 0, 9, 5, 2, 14, 11, 8]
z[32..47] = [3, 10, 14, 4, 9, 15, 8, 1, 2, 7, 0, 6, 13, 11, 5, 12]
z[48..63] = [1, 9, 11, 10, 0, 8, 12, 4, 13, 3, 7, 15, 14, 5, 6, 2]

z'[0..15] = [5, 14, 7, 0, 9, 2, 11, 4, 13, 6, 15, 8, 1, 10, 3, 12]
z'[16..31] = [6, 11, 3, 7, 0, 13, 5, 10, 14, 15, 8, 12, 4, 9, 1, 2]
z'[32..47] = [15, 5, 1, 3, 7, 14, 6, 9, 11, 8, 12, 2, 10, 0, 4, 13]
z'[48..63] = [8, 6, 4, 1, 3, 11, 15, 0, 5, 12, 2, 13, 9, 7, 10, 14]

Esto se puede resumir si se define una permutación ρ como sigue:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\rho(i)$	7	4	13	1	10	6	15	3	12	0	9	5	2	14	11	8

Tabla 17. Permutación ρ en RIPEMD-128

Además de la permutación $\pi(i) = (9i + 5) \bmod 16$ como:

Línea	Ronda 1	Ronda 2	Ronda 3	Ronda 4
Izquierda	Id	ρ	ρ^2	ρ^3
Derecha	$\pi(i)$	$\rho(\pi(i))$	$\rho^2(\pi(i))$	$\rho^3(\pi(i))$

Tabla 18. Permutación π en RIPEMD-128

Finalmente se define el número de corrimientos a la izquierda en bits (para ambos flujos):

s[0..15] = [11, 14, 15, 12, 5, 8, 7, 9, 11, 13, 14, 15, 6, 7, 9, 8]
s[16..31] = [12, 13, 11, 15, 6, 9, 9, 7, 12, 15, 11, 13, 7, 8, 7, 7]
s[32..47] = [13, 15, 14, 11, 7, 7, 6, 8, 13, 14, 13, 12, 5, 5, 6, 9]
s[48..63] = [14, 11, 12, 14, 8, 6, 5, 5, 15, 12, 15, 14, 9, 9, 8, 6]

Las funciones de cada ronda se definen como:

$$F(X, Y, Z) = X \oplus Y \oplus Z$$
$$G(X, Y, Z) = (XY) \vee ((\neg X)Z)$$
$$H(X, Y, Z) = (X \vee (\neg Y)) \oplus Z$$
$$I(X, Y, Z) = (XZ) \vee (Y(\neg Z))$$

8.1.1.2 Preprocesamiento.

Expansión del mensaje.

Se expande o rellena la entrada "*x*" hasta que su longitud (en bits) sea congruente a *448 mod 512*, es decir, 64 bits más del siguiente múltiplo de 512 bits, ya que esta expansión se realiza siempre, incluso cuando la longitud de la entrada ya es congruente a *448 mod 512*.

La forma de realizar esta expansión es la siguiente:

- Se concatena un **"1-bit"** al final del mensaje *M*.
- Se le agrega al resultado anterior *r* "**0-bit**" ($0 \leq r \leq 511$) de tal forma que la longitud de la cadena resultante sea congruente a *448 mod 512* bits.
- Se concatena al final una representación de 64 bits de la longitud inicial del mensaje (*Mlen*) antes de la fase de expansión, es decir, $b \bmod 2^{64}$, siendo *b* la longitud inicial del mensaje.

El mensaje resultante (salida) del preprocesamiento tiene una longitud exacta a un múltiplo de 512 bits, por lo que se forman *16*n palabras* (conjunto de 32 bits). Que pueden ser vistas como un arreglo *M[0 .. N-1]* donde *N* es un múltiplo de 16.

Se inicializa el buffer MD (conjunto de 4 palabras utilizadas para realizar el cálculo del *hash value*) con los valores previamente mencionados.

$$(A, B, C, D) \leftarrow (h_1, h_2, h_3, h_4).$$

8.1.1.3 Procesamiento

La salida *M[0 .. N-1]* resultante del preprocesamiento, contiene *N* bloques de 16 palabras de 32 bits.

```
PARA i = 0 HASTA N/16-1 HAZ
INICIA
    /* Copiar bloque en almacenamiento temporal X. */
    PARA j = 0 HASTA 15 HAZ
            X[j] ← M[i*16+j]
    FIN PARA
    /* Almacenar. */
    A ← H1
    B ← H2
    C ← H3
    D ← H4
    AA ← H1
    BB ← H2
    CC ← H3
    DD ← H4
    /* Ronda 1. */
    // [abcd k s] indica la operación        a = (a + F(b, c, d) + X[k]) <<< s.
    [ABCD  0 11]   [DABC 1 14]    [CDAB 2 15]    [BCDA 3 12]
    [ABCD  4  5]   [DABC 5 8]     [CDAB 6 7]     [BCDA 7 9]
    [ABCD  8 11]   [DABC 9 13]    [CDAB 10 14]   [BCDA 11 15]
    [ABCD 12  6]   [DABC 13 7]    [CDAB 14 9]    [BCDA 15 8]
    /* Ronda 2. */
    // [abcd k s] indica la operación        a = (a + G(b, c, d) + X[k] + 5A827999) <<< s.
    [ABCD 7 7]     [DABC 4 6]     [CDAB 13 8]    [BCDA 1 13]
```

[ABCD 10 11] [DABC 6 9] [CDAB 15 7] [BCDA 3 15]
[ABCD 12 7] [DABC 0 12] [CDAB 9 15] [BCDA 5 9]
[ABCD 2 11] [DABC 14 7] [CDAB 11 13] [BCDA 8 12]

/* Ronda 3. */

/ / [abcd k s] indica la operación a = (a + H(b, c, d) + X[k] + 6ED9EBA1) <<< s.

[ABCD 3 11] [DABC 10 13] [CDAB 14 6] [BCDA 4 7]
[ABCD 9 14] [DABC 15 9] [CDAB 8 13] [BCDA 1 15]
[ABCD 2 14] [DABC 7 8] [CDAB 0 13] [BCDA 6 6]
[ABCD 13 5] [DABC 11 12] [CDAB 5 7] [BCDA 12 5]

/* Ronda 4. */

/ / [abcd k s] indica la operación a = (a + I(b, c, d) + X[k] + 8F1BBCDC) <<< s.

[ABCD 1 11] [DABC 9 12] [CDAB 11 14] [BCDA 10 15]
[ABCD 0 14] [DABC 8 15] [CDAB 12 9] [BCDA 4 8]
[ABCD 13 9] [DABC 3 14] [CDAB 7 5] [BCDA 15 6]
[ABCD 14 8] [DABC 5 6] [CDAB 6 5] [BCDA 2 12]

/* Ronda PARALELA 1. */

/ / [abcd k s] indica la operación a = (a + I(b, c, d) + X[k] + 50A28BE6U) <<< s.

[AA BB CC DD 5 8] [DD AA BB CC 14 9] [CC DD AA BB 7 9] [BB CC DD AA 0 11]
[AA BB CC DD 9 13] [DD AA BB CC 2 15] [CC DD AA BB 11 15] [BB CC DD AA 4 5]
[AA BB CC DD 13 7] [DD AA BB CC 6 7] [CC DD AA BB 15 8] [BB CC DD AA 8 11]
[AA BB CC DD 1 14] [DD AA BB CC 10 14] [CC DD AA BB 3 12] [BB CC DD AA 12 6]

/* Ronda PARALELA 2. */

/ / [abcd k s] indica la operación a = (a + H(b, c, d) + X[k] + 5C4DD124) <<< s.

[AA BB CC DD 6 9] [DD AA BB CC 11 13] [CC DD AA BB 3 15] [BB CC DD AA 7 7]
[AA BB CC DD 0 12] [DD AA BB CC 13 8] [CC DD AA BB 5 9] [BB CC DD AA 10 11]
[AA BB CC DD 14 7] [DD AA BB CC 15 7] [CC DD AA BB 8 12] [BB CC DD AA 12 7]
[AA BB CC DD 4 6] [DD AA BB CC 9 15] [CC DD AA BB 1 13] [BB CC DD AA 2 11]

/* Ronda PARALELA 3. */

/ / [abcd k s] indica la operación a = (a + G(b, c, d) + X[k] + 6D703EF3) <<< s.

[AA BB CC DD 15 9] [DD AA BB CC 5 7] [CC DD AA BB 1 15] [BB CC DD AA 3 11]
[AA BB CC DD 7 8] [DD AA BB CC 14 6] [CC DD AA BB 6 6] [BB CC DD AA 9 14]
[AA BB CC DD 11 12] [DD AA BB CC 8 13] [CC DD AA BB 12 5] [BB CC DD AA 2 14]
[AA BB CC DD 10 13] [DD AA BB CC 0 13] [CC DD AA BB 4 7] [BB CC DD AA 13 5]

/* Ronda PARALELA 4. */

/ / [abcd k s] indica la operación a = (a + F(b, c, d) + X[k]) <<< s.

[AA BB CC DD 8 15] [DD AA BB CC 6 5] [CC DD AA BB 4 8] [BB CC DD AA 1 11]
[AA BB CC DD 3 14] [DD AA BB CC 11 14] [CC DD AA BB 15 6] [BB CC DD AA 0 14]

```
[AA BB CC DD 5 6]      [DD AA BB CC 12 9]     [CC DD AA BB 2 12]     [BB CC DD AA 13 9]
[AA BB CC DD 9 12]     [DD AA BB CC 7 5]      [CC DD AA BB 10 15]    [BB CC DD AA 14 8]
/* Se actualizan los valores del buffer MD con los valores con los que fueron iniciado */
DD = H2 + C + DD
H2 = H3 + D + AA
H3 = H4 + A + BB
H4 = H1 + B + CC
H1 = DD
TERMINA PARA
```

De *forma algebraica* puede definirse de la siguiente manera:

```
PARA cada M[i] desde 0 hasta N-1,
INICIA
    /* Se copia el i^th bloque en un almacenamiento temporal */
    X[j] ← M16i+j, 0 ≤ j ≤ 15
    /* Se almacenan los valores iniciales del buffer MD */
    (A, B, C, D) ← (H1, H2, H3, H4)
    (AA, BB, CC, DD) ← (H1, H2, H3, H4)
    /* Se procesa el bloque con las funciones de ronda, constante aditiva y de corrimiento */
    /* Ronda 1  */
    PARA j = 0 HASTA 15 HAZ
    INICIA
        /* Se ejecuta */
        t ← ( A + F(B, C, D) + X[z[j]] + y[j] ),
        /* Se actualiza el orden y se efectúa el corrimiento correspondiente */
        (A, B, C, D) ← (D, t ← s[j], B, C)
    TERMINA PARA
    /* Ronda 2 */
    PARA j = 16 HASTA 31 HAZ
    INICIA
        t ← ( A + G(B, C, D) + X[z[j]] + y[j] ),
        /* Se actualiza el orden y se efectúa el corrimiento correspondiente */
        (A, B, C, D) ← (D, t ← s[j], B, C)
    TERMINA PARA
```

```
/* Ronda 3 */
PARA j = 32 HASTA 47 HAZ
INICIA
    t ← ( A + H(B, C, D) + X[z[j]] + y[j] ),
    /* Se actualiza el orden y se efectúa el corrimiento correspondiente */
    (A, B, C, D) ← (D, t ← s[j], B, C)
TERMINA PARA
/* Ronda 4 */
PARA j = 48 HASTA 63 HAZ
INICIA
    t ← ( A + I(B, C, D) + X[z[j]] + y[j] ),
    /* Se actualiza el orden y se efectúa el corrimiento correspondiente */
    (A, B, C, D) ← (D, t ← s[j], B, C)
TERMINA PARA
/* Se procesa el bloque con las funciones de ronda, constante aditiva y de corrimiento */
/* Ronda PARALELA 1  */
PARA j = 0 HASTA 15 HAZ
INICIA
    /* Se ejecuta */
    t ← ( AA + I(BB, CC, DD) + X[z'[j]] + y[j] ),
    /* Se actualiza el orden y se efectúa el corrimiento correspondiente */
    (AA, BB, CC, DD) ← (DD, t ← s[j], BB, CC)
TERMINA PARA
/* Ronda PARALELA 2 */
PARA j = 16 HASTA 31 HAZ
INICIA
    t ← ( AA + H(BB, CC, DD) + X[z'[j]] + y[j] ),
    /* Se actualiza el orden y se efectúa el corrimiento correspondiente */
     (AA, BB, CC, DD) ← (DD, t ← s[j], BB, CC)
TERMINA PARA
/* Ronda PARALELA 3 */
PARA j = 32 HASTA 47 HAZ
INICIA
    t ← ( AA + G(BB, CC, DD) + X[z'[j]] + y[j] ),
    /* Se actualiza el orden y se efectúa el corrimiento correspondiente */
    (AA, BB, CC, DD) ← (DD, t ← s[j], BB, CC)
TERMINA PARA
```

```
        /* Ronda PARALELA 4 */
        PARA j = 48 HASTA 63 HAZ
        INICIA
            t ← ( AA + F(BB, CC, DD) + X[z'[j]] + y[j] ),
            /* Se actualiza el orden y se efectúa el corrimiento correspondiente */
            (AA, BB, CC, DD) ← (DD, t ← s[j], BB, CC)
        TERMINA PARA
        /* Al terminar de ejecutar las rondas se actualizan las variables de encadenamiento */
        DD = H2 + C + DD
        H2 = H3 + D + AA
        H3 = H4 + A + BB
        H4 = H1 + B + CC
        H1 = DD
    TERMINA PARA
```

8.1.1.4 Salida Hash Value.

El hash value es la concatenación: *H4* || *H3* || *H2* || *H1*. Esto es empieza con el byte de menor orden de H1 y termina con el byte de alto orden de H4.

8.1.2 RIPEMD-160

Las características principales que permiten la definición de esta función son [5]:

- **Entrada**: cadena de bits "*x*" de longitud arbitraria $b \geq 0$.
- **Salida**: un *hash value* de 160 bits de "*x*".
- **Tamaño del bloque**: 512 bits.
- **Número de rondas**: 5.

8.1.2.1 Definición de constantes.

Se definen el vector de inicialización (las 5 palabras iniciales de 32 bits) *IV* denominados buffer MD.

$h_1 = 0x67452301, h_2 = 0xefcdab89, h_3 = 0x98badcfe, h_4 = 0x10325476, h_5 = c3d2e1f0$

Nótese que las primeras 3 constantes del *IV* son las mismas que en MD4.

Debido a las características del algoritmo (contiene dos flujos paralelos) las constantes aditivas definidas son:

Ronda	Izquierda	Definición	Derecha	Definición
1	0	0	50A28BE6	$2^{30}\sqrt[3]{2}$
2	5A827999	$2^{30}\sqrt{2}$	5C4DD124	$2^{30}\sqrt[3]{3}$
3	6ED9EBA1	$2^{30}\sqrt{3}$	6D703EF3	$2^{30}\sqrt[3]{5}$
4	8F1BBCDC	$2^{30}\sqrt{5}$	7A6D76E9	$2^{30}\sqrt[3]{7}$
5	A953FD4E	$2^{30}\sqrt{7}$	0	0

Tabla 19. Constantes de RIPEMD-160

Nótese que el valor de las constantes de las 3 primeras rondas es la misma que RIPEMD-128.

Tomando como referencia la definición de las permutaciones ρ y π se define el acceso a las palabras como sigue:

Línea	Ronda 1	Ronda 2	Ronda 3	Ronda 4	Ronda 5
Izquierda	Id	ρ	ρ^2	ρ^3	ρ^4
Derecha	$\pi(i)$	$\rho(\pi(i))$	$\rho^2(\pi(i))$	$\rho^3(\pi(i))$	$\rho^4(\pi(i))$

Tabla 20. Permutaciones ρ y π en RIPEMD-160

Finalmente se define el número de corrimientos a la izquierda en bits (para ambos flujos):

s[0..15] = [11, 14, 15, 12, 5, 8, 7, 9, 11, 13, 14, 15, 6, 7, 9, 8]
s[16..31] = [12, 13, 11, 15, 6, 9, 9, 7, 12, 15, 11, 13, 7, 8, 7, 7]
s[32..47] = [13, 15, 14, 11, 7, 7, 6, 8, 13, 14, 13, 12, 5, 5, 6, 9]
s[48..63] = [14, 11, 12, 14, 8, 6, 5, 5, 15, 12, 15, 14, 9, 9, 8, 6]
s[64..79] = [15, 12, 13, 13, 9, 5, 8, 6, 14, 11, 12, 11, 8, 6, 5, 5]

Nótese que con respecto a RIPEMD-128 solo se agregan los corrimientos correspondientes a la quinta ronda.

Las funciones de las primeras cuatro rondas son las mismas que en RIPEMD-128 por lo que solo se incluye la de la quinta ronda:

$F(X, Y, Z) = X \oplus Y \oplus Z$

$G(X, Y, Z) = (XY) \vee ((\neg X)Z)$

$H(X, Y, Z) = (X \vee (\neg Y)) \oplus Z$

$I(X, Y, Z) = (XZ) \vee (Y(\neg Z))$

$J(X, Y, Z) = X \oplus (Y \vee (\neg Z))$

8.1.2.2 Preprocesamiento.

Expansión del mensaje.

Se expande o rellena la entrada "*x*" hasta que su longitud (en bits) sea congruente a *448 mod 512*, es decir, 64 bits más del siguiente múltiplo de 512 bits, ya que esta expansión se realiza siempre, incluso cuando la longitud de la entrada ya es congruente a *448 mod 512*.

La forma de realizar esta expansión es la siguiente:

- Se concatena un **"1-bit"** al final del mensaje *M*.
- Se le agrega al resultado anterior *r* "**0-bit**" ($0 \leq r \leq 511$) de tal forma que la longitud de la cadena resultante sea congruente a *448 mod 512* bits.
- Se concatena al final una representación de 64 bits de la longitud inicial del mensaje (*Mlen*) antes de la fase de expansión, es decir, $b \bmod 2^{64}$, siendo *b* la longitud inicial del mensaje.

El mensaje resultante (salida) del preprocesamiento tiene una longitud exacta a un múltiplo de 512 bits, por lo que se forman *16*n palabras* (conjunto de 32 bits). Que pueden ser vistas como un arreglo *M[0 .. N-1]* donde *N* es un múltiplo de 16.

Se inicializa el buffer MD (conjunto de 4 palabras utilizadas para realizar el cálculo del *hash value*) con los valores previamente mencionados.

$$(A, B, C, D, E) \leftarrow (h_1, h_2, h_3, h_4, h_5).$$

8.1.2.3 Procesamiento

La salida *M[0 .. N-1]* resultante del preprocesamiento, contiene *N* bloques de 16 palabras de 32 bits.

```
PARA i = 0 HASTA N/16-1 HAZ
INICIA
    /* Copiar bloque en almacenamiento temporal X. */
    PARA j = 0 HASTA 15 HAZ
        X[j] ← M[i*16+j]
    FIN PARA
    /* Almacenar. */
    A ← H1
    B ← H2
    C ← H3
    D ← H4
    E ← H5
    AA ← H1
    BB ← H2
    CC ← H3
    DD ← H4
    EE ← H5
    /* Ronda 1. */
    /* [abcde k s] indica la operación
        a = ((a + F(b, c, d) + X[k]) <<< s) + e
        c = c <<< 10                                    */
    [ABCDE 0 11]   [EABCD 1 14]   [DEABC 2 15]   [CDEAB 3 12]   [BCDEA 4 5]
    [ABCDE 5 8]    [EABCD 6 7]    [DEABC 7 9]    [CDEAB 8 11]   [BCDEA 9 13]
    [ABCDE 10 14]  [EABCD 11 15]  [DEABC 12  6]  [CDEAB 13 7]   [BCDEA 14 9]
    [ABCDE 15 8]
    /* Ronda 2. */
    /* [abcde k s] indica la operación
        a = ((a + G(b, c, d) + X[k]) <<< s) + e
        c = c <<< 10                                    */
    [EABCD 7 7]    [DEABC 4 6]    [CDEAB 13 8]   [BCDEA 1 13]   [ABCDE 10 11]
    [EABCD 6 9]    [DEABC 15 7]   [CDEAB 3 15]   [BCDEA 12 7]   [ABCDE 0 12]
    [EABCD 9 15]   [DEABC 5 9]    [CDEAB 2 11]   [BCDEA 14 7]   [ABCDE 11 13]
    [EABCD 8 12]
    /* Ronda 3. */
    /* [abcde k s] indica la operación
        a = ((a + H(b, c, d) + X[k]) <<< s) + e
        c = c <<< 10                                    */
```

[DEABC 3 11] [CDEAB 10 13] [BCDEA 14 6] [ABCDE 4 7] [EABCD 9 14]
[DEABC 15 9] [CDEAB 8 13] [BCDEA 1 15] [ABCDE 2 14] [EABCD 7 8]
[DEABC 0 13] [CDEAB 6 6] [BCDEA 13 5] [ABCDE 11 12] [EABCD 5 7]
[DEABC 12 5]
/* Ronda 4. */
/* [abcde k s] indica la operación
a = ((a + I(b, c, d) + X[k]) <<< s) + e
c = c <<< 10 */
[CDEAB 1 11] [BCDEA 9 12] [ABCDE 11 14] [EABCD 10 15] [DEABC 0 14]
[CDEAB 8 15] [BCDEA 12 9] [ABCDE 4 8] [EABCD 13 9] [DEABC 3 14]
[CDEAB 7 5] [BCDEA 15 6] [ABCDE 14 8] [EABCD 5 6] [DEABC 6 5]
[CDEAB 2 12]
/* Ronda 5. */
/* [abcde k s] indica la operación
a = ((a + J(b, c, d) + X[k]) <<< s) + e
c = c <<< 10 */
[BCDEA 4 9] [ABCDE 015] [EABCD 5 5] [DEABC 9 11] [CDEAB 7 6]
[BCDEA 12 8] [ABCDE 2 13] [EABCD 10 12] [DEABC 14 5] [CDEAB 1 12]
[BCDEA 3 13] [ABCDE 8 14] [EABCD 11 11] [DEABC 6 8] [CDEAB 15 5]
[BCDEA 13 6]
/* Ronda PARALELA 1. */
/* [abcde k s] indica la operación
a = ((a + J(b, c, d) + X[k]) <<< s) + e
c = c <<< 10 */
[AA BB CC DD EE 5 8] [EE AA BB CC DD 14 9] [DD EE AA BB CC 7 9] [CC DD EE AA BB 0 11]
[BB CC DD EE AA 9 13]
[AA BB CC DD EE 2 15] [EE AA BB CC DD 11 15] [DD EE AA BB CC 4 5] [CC DD EE AA BB 13 7]
[BB CC DD EE AA 6 7]
[AA BB CC DD EE 15 8] [EE AA BB CC DD 8 11] [DD EE AA BB CC 1 14] [CC DD EE AA BB 10 14]
[BB CC DD EE AA 3 12]
[AA BB CC DD EE 12 6]
/* Ronda PARALELA 2. */
/* [abcde k s] indica la operación
a = ((a + I(b, c, d) + X[k]) <<< s) + e
c = c <<< 10 */
[EE AA BB CC DD 6 9] [DD EE AA BB CC 11 13] [CC DD EE AA BB 3 15] [BB CC DD EE AA 7 7]
[AA BB CC DD EE 0 12]

[EE AA BB CC DD 13 8] [DD EE AA BB CC 5 9] [CC DD EE AA BB 10 11] [BB CC DD EE AA 14 7]
[AA BB CC DD EE 15 7]
[EE AA BB CC DD 8 12] [DD EE AA BB CC 12 7] [CC DD EE AA BB 4 6] [BB CC DD EE AA 9 15]
[AA BB CC DD EE 1 13]
[EE AA BB CC DD 2 11]
/* Ronda PARALELA 3. */
/* [abcde k s] indica la operación
a = ((a + H(b, c, d) + X[k]) <<< s) + e
c = c <<< 10 */
[DD EE AA BB CC 15 9] [CC DD EE AA BB 5 7] [BB CC DD EE AA 1 15] [AA BB CC DD EE 3 11]
[EE AA BB CC DD 7 8]
[DD EE AA BB CC 14 6] [CC DD EE AA BB 6 6] [BB CC DD EE AA 9 14] [AA BB CC DD EE 11 12]
[EE AA BB CC DD 8 13]
[DD EE AA BB CC 12 5] [CC DD EE AA BB 2 14] [BB CC DD EE AA 10 13] [AA BB CC DD EE 0 13]
[EE AA BB CC DD 4 7]
[DD EE AA BB CC 13 5]
/* Ronda PARALELA 4. */
/* [abcde k s] indica la operación
a = ((a + G(b, c, d) + X[k]) <<< s) + e
c = c <<< 10
*/
[CC DD EE AA BB 8 15] [BB CC DD EE AA 6 5] [AA BB CC DD EE 4 8] [EE AA BB CC DD 1 11]
[DD EE AA BB CC 3 14]
[CC DD EE AA BB 11 14] [BB CC DD EE AA 15 6] [AA BB CC DD EE 0 14] [EE AA BB CC DD 5 6]
[DD EE AA BB CC 12 9]
[CC DD EE AA BB 2 12] [BB CC DD EE AA 13 9] [AA BB CC DD EE 9 12] [EE AA BB CC DD 7 5]
[DD EE AA BB CC 10 15]
[CC DD EE AA BB 14 8]
/* Ronda PARALELA 5. */
/* [abcde k s] indica la operación
a = ((a + F(b, c, d) + X[k]) <<< s) + e
c = c <<< 10
*/
[BB CC DD EE AA 12 8] [AA BB CC DD EE 15 5] [EE AA BB CC DD 10 12] [DD EE AA BB CC 4 9]
[CC DD EE AA BB 1 12]
[BB CC DD EE AA 5 5] [AA BB CC DD EE 8 14] [EE AA BB CC DD 7 6] [DD EE AA BB CC 6 8]
[CC DD EE AA BB 2 13]

```
[BB CC DD EE AA 13 6] [AA BB CC DD EE 14 5] [EE AA BB CC DD 0 15] [DD EE AA BB CC 3 13]
[CC DD EE AA BB 9 11]
[BB CC DD EE AA 11 11]
/* Se actualizan los valores del buffer MD con los valores con los que fueron iniciado */
DD = H2 + C + DD
H2 = H3 + D + EE
H3 = H4 + E + AA
H4 = H5 + A + BB
H5 = H0 + B + CC
H1 = DD
TERMINA PARA
```

De forma algebraica puede definirse de la siguiente manera:

```
PARA cada M[i] desde 0 hasta N-1,
INICIA
    /* Se copia el i^th bloque en un almacenamiento temporal */
    X[j] ← M16i+j, 0 ≤ j ≤ 15
    /* Se almacenan los valores iniciales del buffer MD */
    (A, B, C, D, E) ← (H1, H2, H3, H4, H5)
    (AA, BB, CC, DD, EE) ← (H1, H2, H3, H4, H5)
    /* Se procesa el bloque con las funciones de ronda, constante aditiva y de corrimiento */
    /* Ronda 1  */
    PARA j = 0 HASTA 15 HAZ
    INICIA
        /* Se ejecuta */
        t ← ( A + F(B, C, D) + X[z[j]] + y[j] ),
        t = t << s[j]
        C = C << 10
        /* Se actualiza el orden y se efectúa el corrimiento correspondiente */
        (A, B, C, D, E) ← (E, t, B, C, D)
    TERMINA PARA
    /* Ronda 2 */
    PARA j = 16 HASTA 31 HAZ
    INICIA
        /* Se ejecuta */
        t ← ( A + G(B, C, D) + X[z[j]] + y[j] ),
```

```
    t = t << s[j]
    C = C << 10
    /* Se actualiza el orden y se efectúa el corrimiento correspondiente */
    (A, B, C, D, E) ← (E, t, B, C, D)
TERMINA PARA
/* Ronda 3 */
PARA j = 32 HASTA 47 HAZ
INICIA
    /* Se ejecuta */
    t ← ( A + H(B, C, D) + X[z[j]] + y[j] ),
    t = t << s[j]
    C = C << 10
    /* Se actualiza el orden y se efectúa el corrimiento correspondiente */
    (A, B, C, D, E) ← (E, t, B, C, D)
TERMINA PARA
/* Ronda 4 */
PARA j = 48 HASTA 63 HAZ
INICIA
    /* Se ejecuta */
    t ← ( A + I(B, C, D) + X[z[j]] + y[j] ),
    t = t << s[j]
    C = C << 10
    /* Se actualiza el orden y se efectúa el corrimiento correspondiente */
    (A, B, C, D, E) ← (E, t, B, C, D)
TERMINA PARA
/* Ronda 5 */
PARA j = 64 HASTA 79 HAZ
INICIA
    /* Se ejecuta */
    t ← ( A + J(B, C, D) + X[z[j]] + y[j] ),
    t = t << s[j]
    C = C << 10
    /* Se actualiza el orden y se efectúa el corrimiento correspondiente */
    (A, B, C, D, E) ← (E, t, B, C, D)
TERMINA PARA
/* Se procesa el bloque con las funciones de ronda, constante aditiva y de corrimiento */
/* Ronda PARALELA 1  */
```

```
PARA j = 0 HASTA 15 HAZ
INICIA
    /* Se ejecuta */
    t ← ( A + J(B, C, D) + X[z[j]] + y[j] ),
    t = t << s[j]
    C = C << 10
    /* Se actualiza el orden y se efectúa el corrimiento correspondiente */
    (A, B, C, D, E) ← (E, t, B, C, D)
TERMINA PARA
/* Ronda PARALELA 2 */
PARA j = 16 HASTA 31 HAZ
INICIA
    /* Se ejecuta */
    t ← ( A + I(B, C, D) + X[z[j]] + y[j] ),
    t = t << s[j]
    C = C << 10
    /* Se actualiza el orden y se efectúa el corrimiento correspondiente */
    (A, B, C, D, E) ← (E, t, B, C, D)
TERMINA PARA
/* Ronda PARALELA 3 */
PARA j = 32 HASTA 47 HAZ
INICIA
    /* Se ejecuta */
    t ← ( A + H(B, C, D) + X[z[j]] + y[j] ),
    t = t << s[j]
    C = C << 10
    /* Se actualiza el orden y se efectúa el corrimiento correspondiente */
    (A, B, C, D, E) ← (E, t, B, C, D)
TERMINA PARA
/* Ronda PARALELA 4 */
PARA j = 48 HASTA 63 HAZ
INICIA
    /* Se ejecuta */
    t ← ( A + G(B, C, D) + X[z[j]] + y[j] ),
    t = t << s[j]
    C = C << 10
    /* Se actualiza el orden y se efectúa el corrimiento correspondiente */
```

```
        (A, B, C, D, E) ← (E, t, B, C, D)
    TERMINA PARA
    /* Ronda PARALELA 5 */
    PARA j = 64 HASTA 79 HAZ
    INICIA
        /* Se ejecuta */
        t ← ( A + F(B, C, D) + X[z[j]] + y[j] ),
        t = t << s[j]
        C = C << 10
        /* Se actualiza el orden y se efectúa el corrimiento correspondiente */
        (A, B, C, D, E) ← (E, t, B, C, D)
    TERMINA PARA
    /* Al terminar de ejecutar las rondas se actualizan las variables de encadenamiento */
    DD = H2 + C + DD
    H2 = H3 + D + EE
    H3 = H4 + E + AA
    H4 = H5 + A + BB
    H5 = H0 + B + CC
    H1 = DD
TERMINA PARA
```

8.1.2.4 Salida Hash Value.

El hash value es la concatenación: *H5* || *H4* || *H3* || *H2* || *H1*. Esto es empieza con el byte de menor orden de H1 y termina con el byte de alto orden de H5.

8.2 HAVAL

Propuesto por Yuliang Zheng, Josef Pieprzyk y Jennifer Seberry en 1992. Su estructura es similar a MD4 y MD5. Sin embargo, HAVAL [8] permite que la longitud del hash value sea variable, Esto permite alternar entre eficiencia y seguridad a partir de un parámetro, el número de rondas, el cual puede ser elegido entre 3, 4 o 5.

Las características principales son:

- **Entrada**: cadena de bits "*x*" de longitud arbitraria $0 \leq b < 2^{64}$.
- **Salida**: un *hash value* de 128, 160, 192, 224, 256.
- **Tamaño del bloque**: 1024 bits (32 palabras de 32 bits)
- **Número de rondas**: 3, 4, 5

8.2.1 Definición de constantes.

Se definen el vector de inicialización (las 8 palabras iniciales de 32 bits) *IV*.

$h_1 = EC4E6C89$ $h_2 = 082EFA98$ $h_3 = 299F31D0$ $h_4 = A4093822$

$h_5 = 03707344$ $h_6 = 13198A2E$ $h_7 = 85A308D3$ $h_8 = 243F6A88$

HAVAL utiliza 136 constantes de 32 bits. Las primeras 8 palabras son utilizadas para inicializar el IV ($D_{0,7}$, $D_{0,6}$, $D_{0,5}$, $D_{0,4}$, $D_{0,3}$, $D_{0,2}$, $D_{0,1}$, $D_{0,0}$), la ronda 2 utiliza 32 palabras ($K_{2,31}$, $K_{2,30}$, …, $K_{2,0}$) al igual que la ronda 3 ($K_{3,31}$, $K_{3,30}$, …, $K_{3,0}$), 4 ($K_{4,31}$, $K_{4,30}$, …, $K_{4,0}$) y 5 ($K_{5,31}$, $K_{5,30}$, …, $K_{5,0}$).

Las primeras 8 palabras corresponden a los primeros 256 bits de la parte fraccional de π. Seguidos en las demás rondas por los 1024 bits de la parte fraccional de π.

243F6A88	85A308D3	13198A2E	03707344	A4093822	299F31D0	082EFA98	EC4E6C89
452821E6	38D01377	BE5466CF	34E90C6C	C0AC29B7	C97C50DD	3F84D5B5	B5470917
9216D5D9	8979FB1B	D1310BA6	98DFB5AC	2FFD72DB	D01ADFB7	B8E1AFED	6A267E96
BA7C9045	F12C7F99	24A19947	B3916CF7	0801F2E2	858EFC16	636920D8	71574E69
A458FEA3	F4933D7E	0D95748F	728EB658	718BCD58	82154AEE	7B54A41D	C25A59B5
9C30D539	2AF26013	C5D1B023	286085F0	CA417918	B8DB38EF	8E79DCB0	603A180E
6C9E0E8B	B01E8A3E	D71577C1	BD314B27	78AF2FDA	55605C60	E65525F3	AA55AB94
57489862	63E81440	55CA396A	2AAB10B6	B4CC5C34	1141E8CE	A15486AF	7C72E993
B3EE1411	636FBC2A	2BA9C55D	741831F6	CE5C3E16	9B87931E	AFD6BA33	6C24CF5C
7ª325381	28958677	3B8F4898	6B4BB9AF	C4BFE81B	66282193	61D809CC	FB21A991
487CAC60	5DEC8032	EF845D5D	E98575B1	DC262302	EB651B88	23893E81	D396ACC5
0F6D6FF3	83F44239	2E0B4482	A4842004	69C8F04A	9E1F9B5E	21C66842	F6E96C9A
670C9C61	ABD388F0	6A51A0D2	D8542F68	960FA728	AB5133A3	6EEF0B6C	137A3BE4

BA3BF050	7EFB2A98	A1F1651D	39AF0176	66CA593E	82430E88	8CEE8619	456F9FB4
7D84A5C3	3B8B5EBE	E06F75D8	85C12073	401A449F	56C16AA6	4ED3AA62	363F7706
1BFEDF72	429B023D	37D0D724	D00A1248	DB0FEAD3	49F1C09B	075372C9	80991B7B
25D479D8	F6E8DEF7	E3FE501A	B6794C3B	976CE0BD	04C006BA	C1A94FB6	409F60C4

Tabla 21. Constantes de HAVAL.

El orden de acceso a las palabras durante el desarrollo es el siguiente:

Original	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
H_1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Ord_2	5	14	26	18	11	28	7	16	0	23	20	22	1	10	4	8
H_2	30	3	21	9	17	24	29	6	19	12	15	13	2	25	31	27
Ord_3	19	9	4	20	28	17	8	22	29	14	25	12	24	30	16	26
H_3	31	15	7	3	1	0	18	27	13	6	21	10	23	11	5	2
Ord_4	24	4	0	14	2	7	28	23	26	6	30	20	18	25	19	3
H_4	22	11	31	21	8	27	12	9	1	29	5	15	17	10	16	13
Ord_5	27	3	21	26	17	11	20	29	19	0	12	7	13	8	31	10
H_5	5	9	14	30	18	6	28	24	2	23	16	22	4	1	25	15

Tabla 22. Orden de acceso en HAVAL.

Las funciones de ronda se definen como:

$$F1(x_6, x_5, x_4, x_3, x_2, x_1, x_0) = (x_1 x_4) \oplus (x_2 x_5) \oplus (x_3 x_6) \oplus (x_0 x_1) \oplus x_0$$

$$F2(x_6, x_5, x_4, x_3, x_2, x_1, x_0) = (x_1 x_2 x_3) \oplus (x_2 x_4 x_5) \oplus (x_1 x_2) \oplus (x_1 x_4) \oplus (x_2 x_6) \oplus (x_3 x_5) \oplus (x_4 x_5) \oplus (x_0 x_2) \oplus x_0$$

$$F3(x_6, x_5, x_4, x_3, x_2, x_1, x_0) = (x_1 x_2 x_3) \oplus (x_1 x_4) \oplus (x_2 x_5) \oplus (x_3 x_6) \oplus (x_0 x_3) \oplus x_0$$

$$F4(x_6, x_5, x_4, x_3, x_2, x_1, x_0) = (x_1 x_2 x_3) \oplus (x_2 x_4 x_5) \oplus (x_3 x_4 x_6) \oplus (x_1 x_4) \oplus (x_2 x_6) \oplus (x_3 x_4) \oplus (x_3 x_5) \oplus (x_3 x_6) \oplus (x_4 x_5) \oplus (x_4 x_6) \oplus (x_0 x_4) \oplus x_0$$

$$F5(x_6, x_5, x_4, x_3, x_2, x_1, x_0) = (x_1 x_4) \oplus (x_2 x_5) \oplus (x_3 x_6) \oplus (x_0 x_1 x_2 x_3) \oplus (x_0 x_5) \oplus x_0$$

Cada función recibe 7 palabras como parámetros al inicio del procesamiento son los valores del IV, sin embargo, el orden en el que se toman no es el mismo según sea el número de pasadas este cambia, por lo que en la tabla siguiente se define la permutación θ que establece el orden:

Permutación	X_6	X_5	X_4	X_3	X_2	X_1	X_0
$\theta_{3,1}$	X_1	X_0	X_3	X_5	X_6	X_2	X_4
$\theta_{3,2}$	X_4	X_2	X_1	X_0	X_5	X_3	X_6
$\theta_{3,3}$	X_6	X_1	X_2	X_3	X_4	X_5	X_0
$\theta_{4,1}$	X_2	X_6	X_1	X_4	X_5	X_3	X_0
$\theta_{4,2}$	X_3	X_5	X_2	X_0	X_1	X_6	X_4
$\theta_{4,3}$	X_1	X_4	X_3	X_6	X_0	X_2	X_5
$\theta_{4,4}$	X_6	X_4	X_0	X_5	X_2	X_1	X_3
$\theta_{5,1}$	X_3	X_4	X_1	X_0	X_5	X_2	X_6
$\theta_{5,2}$	X_6	X_2	X_1	X_0	X_3	X_4	X_5
$\theta_{5,3}$	X_2	X_6	X_0	X_4	X_3	X_1	X_5
$\theta_{5,4}$	X_1	X_5	X_3	X_2	X_0	X_4	X_6
$\theta_{5,5}$	X_2	X_5	X_0	X_6	X_4	X_3	X_1

Tabla 23. Permutación definida para HAVAL.

8.2.2 Preprocesamiento.

Expansión del mensaje.

Se expande el mensaje de entrada "*x*" hasta que su longitud (en bits) sea congruente a *944 mod 1024*, esta expansión se realiza siempre, incluso cuando la longitud de la entrada ya es congruente.

La forma de realizar esta expansión es la siguiente:

- Se concatena un **"1-bit"** al final del mensaje *M*.
- Se le agrega al resultado anterior *r* "**0-bit**" ($0 \leq r \leq 1023$) de tal forma que la longitud de la cadena resultante sea congruente a *944 mod 1024* bits.

Al final se agregan:

- Un campo de 3 bits que representa la versión actualmente 1.
- Un campo de 3 bits que representa el número de pasadas o rondas (3, 4 o 5).

- Un campo de 10 bits que representa la longitud del *hash value* (128, 160,192, 224 o 256).
- Una representación de 64 bits de la longitud inicial del mensaje (*Mlen*) antes de la fase de expansión.

El mensaje resultante (salida) del preprocesamiento tiene una longitud exacta a un múltiplo de 1024 bits, por lo que se forman *n bloques* (conjunto de 1024 bits).

8.2.3 Procesamiento

Debido a que las características del algoritmo dependen del número de rondas estas se describirán indicando las diferencias:

8.2.3.1 Ronda 1

Llamemos a esta ronda H_1 cuyos parámetros son las 8 palabras de 32 bits constantes ($E_{0,7}$, $E_{0,6}$, …, $E_{0,0}$) que llamaremos E_0 y las 32 palabras (W_{31}, W_{30}, …, W_0) que conforman el bloque a procesar denotado por B. La salida de esta función la denotaremos como E_1 ($E_{1,7}$, $E_{1,6}$, …, $E_{1,0}$).

La función H_1 se define como:

```
FUNCTION H1(E0[0..7], B[0..31])
INICIO
    T[0..7] = E0[0..7]                                  /* Inicializamos el IV */
    PARA i = 0 HASTA 31 HAZ
    INICIA
        /* El orden en el que se procesan depende del número de rondas */
        P = F1(θ3,1(Ti,6, Ti,5, Ti,4, Ti,3, Ti,2, Ti,1, Ti,0))      /* Si ronda = 3 */
        P = F1(θ4,1(Ti,6, Ti,5, Ti,4, Ti,3, Ti,2, Ti,1, Ti,0))      /* Si ronda = 4 */
        P = F1(θ5,1(Ti,6, Ti,5, Ti,4, Ti,3, Ti,2, Ti,1, Ti,0))      /* Si ronda = 5 */
        /* Calculamos temporal */
        R = (P << 7) ⊗ (Ti,7 << 11) ⊗ Wi                    /*Donde ⊗ indica suma mod 2^32*/
        /* Actualizamos los valores */
```

$T_{i+1,7} = T_{i,6}$

$T_{i+1,6} = T_{i,5}$

$T_{i+1,5} = T_{i,4}$

$T_{i+1,4} = T_{i,3}$

$T_{i+1,3} = T_{i,2}$

$T_{i+1,2} = T_{i,1}$

$T_{i+1,1} = T_{i,0}$

$T_{i+1,0} = R$

TERMINA PARA

/* La salida corresponde a $E_1 = E_{1,7}, E_{1,6}, \ldots, E_{1,0}$ */

$E_1[0..7] = T_{32,i}$ $\quad 0 \leq i \leq 7$

TERMINA

8.2.3.2 Ronda 2

Llamemos a esta ronda H_2 cuyos parámetros son las 32 palabras de 32 bits constantes ($K_{2,31}, K_{2,30}, \ldots, K_{2,0}$) que llamaremos K, la salida anterior E_1 y las 32 palabras que forman el bloque en el orden definido para la segunda ronda (ord2) denotado por B. La salida de esta función la denotaremos como E_2 ($E_{2,7}, E_{2,6}, \ldots, E_{2,0}$).

La función H_2 se define como:

FUNCTION $H_2(E_1[0..7], B[0..31], K[0..31])$

INICIO

$T[0..7] = E_1[0..7]$ /* Inicializamos el IV */

PARA i = 0 HASTA 31 HAZ

INICIA

/* El orden en el que se procesan depende del número de rondas */

$P = F_2(\theta_{3,2}(T_{i,6}, T_{i,5}, T_{i,4}, T_{i,3}, T_{i,2}, T_{i,1}, T_{i,0}))$ /* Si ronda = 3 */

$P = F_2(\theta_{4,2}(T_{i,6}, T_{i,5}, T_{i,4}, T_{i,3}, T_{i,2}, T_{i,1}, T_{i,0}))$ /* Si ronda = 4 */

$P = F_2(\theta_{5,2}(T_{i,6}, T_{i,5}, T_{i,4}, T_{i,3}, T_{i,2}, T_{i,1}, T_{i,0}))$ /* Si ronda = 5 */

/* Calculamos temporal */

$R = (P << 7) \otimes (T_{i,7} << 11) \otimes W_{ord2(i)} \otimes K_i$ /*Donde $\otimes$ indica suma mod 2^{32}*/

/* Actualizamos los valores */

$T_{i+1,7} = T_{i,6}$

$T_{i+1,6} = T_{i,5}$

$T_{i+1,5} = T_{i,4}$

$T_{i+1,4} = T_{i,3}$

$T_{i+1,3} = T_{i,2}$

$T_{i+1,2} = T_{i,1}$

$T_{i+1,1} = T_{i,0}$

$T_{i+1,0} = R$

TERMINA PARA

/* La salida corresponde a $E_2 = E_{2,7}, E_{2,6}, \ldots, E_{2,0}$ */

$E_2[0..7] = T_{32,i}$ $\quad 0 \leq i \leq 7$

TERMINA

8.2.3.3 Ronda 3

Llamemos a esta ronda H_3 cuyos parámetros son las 32 palabras de 32 bits constantes ($K_{3,31}, K_{3,30}, \ldots, K_{3,0}$) que llamaremos K, la salida anterior E_2 y las 32 palabras que forman el bloque en el orden definido para la tercera ronda (ord3) denotado por B. La salida de esta función la denotaremos como E_3 ($E_{3,7}, E_{3,6}, \ldots, E_{3,0}$).

La función H_3 se define como:

FUNCTION $H_3(E_2[0..7], B[0..31], K[0..31])$

INICIO

$T[0..7] = E_2[0..7]$ /* Inicializamos el IV */

PARA i = 0 HASTA 31 HAZ

INICIA

/* El orden en el que se procesan depende del número de rondas */

$P = F_3(\theta_{3,3}(T_{i,6}, T_{i,5}, T_{i,4}, T_{i,3}, T_{i,2}, T_{i,1}, T_{i,0}))$ /* Si ronda = 3 */

$P = F_3(\theta_{4,3}(T_{i,6}, T_{i,5}, T_{i,4}, T_{i,3}, T_{i,2}, T_{i,1}, T_{i,0}))$ /* Si ronda = 4 */

$P = F_3(\theta_{5,3}(T_{i,6}, T_{i,5}, T_{i,4}, T_{i,3}, T_{i,2}, T_{i,1}, T_{i,0}))$ /* Si ronda = 5 */

/* Calculamos temporal */

$R = (P << 7) \otimes (T_{i,7} << 11) \otimes W_{ord3(i)} \otimes K_i$ /*Donde $\otimes$ indica suma mod 2^{32}*/

/* Actualizamos valores */

$T_{i+1,7} = T_{i,6}$

$T_{i+1,6} = T_{i,5}$

$T_{i+1,5} = T_{i,4}$

$T_{i+1,4} = T_{i,3}$

$T_{i+1,3} = T_{i,2}$

$T_{i+1,2} = T_{i,1}$

$T_{i+1,1} = T_{i,0}$

$T_{i+1,0} = R$

TERMINA PARA

/* La salida corresponde a $E_3 = E_{3,7}, E_{3,6}, \ldots, E_{3,0}$ */

$E_3[0..7] = T_{32,i}$ $\quad 0 \leq i \leq 7$

TERMINA

8.2.3.4 Ronda 4

Llamemos a esta ronda H_4 cuyos parámetros son las 32 palabras constantes ($K_{4,31}, K_{4,30}, \ldots, K_{4,0}$) que llamaremos K, la salida anterior E_3 y las 32 palabras que forman el bloque en el orden definido para la cuarta ronda (ord4) denotado por B. La salida de esta función la denotaremos como E_4 ($E_{4,7}, E_{4,6}, \ldots, E_{4,0}$); cabe destacar que esto es válido solo para cuando el numero de rondas es 4 o 5.

La función H_4 se define como:

FUNCTION $H_4(E_3[0..7], B[0..31], K[0..31])$

INICIO

$T[0..7] = E_3[0..7]$ /* Inicializamos el IV */

PARA i = 0 HASTA 31 HAZ

INICIA

/* El orden en el que se procesan depende del número de rondas */

$P = F_4(\theta_{4,4}(T_{i,6}, T_{i,5}, T_{i,4}, T_{i,3}, T_{i,2}, T_{i,1}, T_{i,0}))$ /* Si ronda = 4 */

$P = F_4(\theta_{5,4}(T_{i,6}, T_{i,5}, T_{i,4}, T_{i,3}, T_{i,2}, T_{i,1}, T_{i,0}))$ /* Si ronda = 5 */

/* Calculamos temporal */

$R = (P << 7) \otimes (T_{i,7} << 11) \otimes W_{ord4(i)} \otimes K_i$ /*Donde $\otimes$ indica suma mod 2^{32}*/

/* Actualizamos los valores */

$T_{i+1,7} = T_{i,6}$

$T_{i+1,6} = T_{i,5}$

$T_{i+1,5} = T_{i,4}$

$T_{i+1,4} = T_{i,3}$

$T_{i+1,3} = T_{i,2}$

$T_{i+1,2} = T_{i,1}$

$T_{i+1,1} = T_{i,0}$

$T_{i+1,0} = R$

TERMINA PARA

/* La salida corresponde a $E_4 = E_{4,7}, E_{4,6}, \ldots, E_{4,0}$ */

$E_4[0..7] = T_{32,i}$ $\quad 0 \leq i \leq 7$

TERMINA

8.2.3.5 Ronda 5

Solo es válido cuando el numero de rondas es 5, H_5 recibe como parámetros las 32 palabras constantes ($K_{5,31}, K_{5,30}, \ldots, K_{5,0}$) que llamaremos K, la salida anterior E_4 y las 32 palabras que forman el bloque en el orden definido para la quinta ronda (ord5) denotado por B. La salida de esta función la denotaremos como E_5 ($E_{5,7}, E_{5,6}, \ldots, E_{5,0}$).

La función H_5 se define como:

FUNCTION $H_5(E_4[0..7], B[0..31], K[0..31])$

INICIO

$T[0..7] = E_4[0..7]$ /* Inicializamos el IV */

PARA i = 0 HASTA 31 HAZ

INICIA

$P = F_5(\theta_{5,5}(T_{i,6}, T_{i,5}, T_{i,4}, T_{i,3}, T_{i,2}, T_{i,1}, T_{i,0}))$

$R = (P << 7) \otimes (T_{i,7} << 11) \otimes W_{ord4(i)} \otimes K_i$ /*Donde $\otimes$ indica suma mod 2^{32}*/

/* Actualizamos los valores */

$T_{i+1,7} = T_{i,6}$

$T_{i+1,6} = T_{i,5}$

$T_{i+1,5} = T_{i,4}$

$T_{i+1,4} = T_{i,3}$

$T_{i+1,3} = T_{i,2}$

$T_{i+1,2} = T_{i,1}$

$T_{i+1,1} = T_{i,0}$

$T_{i+1,0} = R$

TERMINA PARA

/* La salida corresponde a $E_4 = E_{4,7}, E_{4,6}, \ldots, E_{4,0}$ */

$E_4[0..7] = T_{32,i}$

TERMINA

8.2.4 Salida Hash Value.

Ya que la longitud es variable la forma de obtener el hash value se muestra a continuación considerando que la última salida de la función es de 256 bits después de procesar todos los bloques del mensaje.

Se tiene lo siguiente: Representemos a la última salida como $D_n = D_{n,7}\ D_{n,6}\ \ldots\ D_{n,0}$ con una longitud de 256 bits, en otras palabras como el conjunto de 8 palabras.

A continuación se definirán las formas de obtener las diferentes longitudes según sea el caso.

8.2.4.1 Hash value de 128 bits

Como primer paso se divide $D_{n,7}$, $D_{n,6}$, $D_{n,5}$ y $D_{n,4}$ de la siguiente forma:

$$D_{n,7} = X^{[8]}_{7,3} X^{[8]}_{7,2} X^{[8]}_{7,1} X^{[8]}_{7,0}$$
$$D_{n,6} = X^{[8]}_{6,3} X^{[8]}_{6,2} X^{[8]}_{6,1} X^{[8]}_{6,0}$$
$$D_{n,5} = X^{[8]}_{5,3} X^{[8]}_{5,2} X^{[8]}_{5,1} X^{[8]}_{5,0}$$
$$D_{n,4} = X^{[8]}_{4,3} X^{[8]}_{4,2} X^{[8]}_{4,1} X^{[8]}_{4,0}$$

El hash obtenido es $Y_3 Y_2 Y_1 Y_0$ donde:

$$Y_3 = D_{n,3} \otimes (X^{[8]}_{7,3} X^{[8]}_{6,2} X^{[8]}_{5,1} X^{[8]}_{4,0})$$
$$Y_2 = D_{n,2} \otimes (X^{[8]}_{7,2} X^{[8]}_{6,1} X^{[8]}_{5,0} X^{[8]}_{4,3})$$
$$Y_1 = D_{n,1} \otimes (X^{[8]}_{7,1} X^{[8]}_{6,0} X^{[8]}_{5,3} X^{[8]}_{4,2})$$
$$Y_0 = D_{n,0} \otimes (X^{[8]}_{7,0} X^{[8]}_{6,3} X^{[8]}_{5,2} X^{[8]}_{4,1})$$

8.2.4.2 Hash value de 160 bits

Se divide $D_{n,7}$, $D_{n,6}$ y $D_{n,5}$ de la siguiente forma:

$$D_{n,7} = X_{7,4}^{[7]} X_{7,3}^{[6]} X_{7,2}^{[7]} X_{7,1}^{[6]} X_{7,0}^{[6]}$$
$$D_{n,6} = X_{6,4}^{[7]} X_{6,3}^{[6]} X_{6,2}^{[7]} X_{6,1}^{[6]} X_{6,0}^{[6]}$$
$$D_{n,5} = X_{5,4}^{[7]} X_{5,3}^{[6]} X_{5,2}^{[7]} X_{5,1}^{[6]} X_{5,0}^{[6]}$$

El hash value se obtiene a partir de la concatenación $Y_4Y_3Y_2Y_1Y_0$ donde:

$$Y_4 = D_{n,4} \otimes (X_{7,4}^{[7]} X_{6,3}^{[6]} X_{5,2}^{[7]})$$
$$Y_3 = D_{n,3} \otimes (X_{7,3}^{[6]} X_{6,2}^{[7]} X_{5,1}^{[6]})$$
$$Y_2 = D_{n,2} \otimes (X_{7,2}^{[7]} X_{6,1}^{[6]} X_{5,0}^{[7]})$$
$$Y_1 = D_{n,1} \otimes (X_{7,1}^{[6]} X_{6,0}^{[6]} X_{5,4}^{[7]})$$
$$Y_0 = D_{n,0} \otimes (X_{7,0}^{[6]} X_{6,4}^{[7]} X_{5,3}^{[6]})$$

8.2.4.3 Hash value de 192 bits

Se divide $D_{n,7}$ y $D_{n,6}$ en:

$$D_{n,7} = X_{7,5}^{[6]} X_{7,4}^{[5]} X_{7,3}^{[5]} X_{7,2}^{[6]} X_{7,1}^{[5]} X_{7,0}^{[5]}$$
$$D_{n,6} = X_{6,5}^{[6]} X_{6,4}^{[5]} X_{6,3}^{[5]} X_{6,2}^{[6]} X_{6,1}^{[5]} X_{6,0}^{[5]}$$

Y definiendo:

$$Y_5 = D_{n,5} \otimes (X_{7,5}^{[6]} X_{6,4}^{[5]})$$
$$Y_4 = D_{n,4} \otimes (X_{7,4}^{[5]} X_{6,3}^{[5]})$$
$$Y_3 = D_{n,3} \otimes (X_{7,3}^{[5]} X_{6,2}^{[6]})$$
$$Y_2 = D_{n,2} \otimes (X_{7,2}^{[6]} X_{6,1}^{[5]})$$
$$Y_1 = D_{n,1} \otimes (X_{7,1}^{[5]} X_{6,0}^{[5]})$$
$$Y_0 = D_{n,0} \otimes (X_{7,0}^{[5]} X_{6,5}^{[6]})$$

El hash value resultante es: $Y_5Y_4Y_3Y_2Y_1Y_0$

8.2.4.4 Hash value de 224 bits

Se divide a $D_{n,7}$ en: $D_{n,7} = X_{7,6}^{[5]} X_{7,5}^{[5]} X_{7,4}^{[4]} X_{7,3}^{[5]} X_{7,2}^{[4]} X_{7,1}^{[5]} X_{7,0}^{[4]}$

Definiendo:

$$Y_6 = D_{n,6} \otimes X_{7,0}^{[4]}$$
$$Y_5 = D_{n,5} \otimes X_{7,1}^{[5]}$$
$$Y_4 = D_{n,4} \otimes X_{7,2}^{[4]}$$
$$Y_3 = D_{n,3} \otimes X_{7,3}^{[5]}$$
$$Y_2 = D_{n,2} \otimes X_{7,4}^{[4]}$$
$$Y_1 = D_{n,1} \otimes X_{7,5}^{[5]}$$
$$Y_0 = D_{n,0} \otimes X_{7,6}^{[5]}$$

El hash value es: $Y_6 Y_5 Y_4 Y_3 Y_2 Y_1 Y_0$

8.2.4.5 Hash value de 256 bits

El *hash value* es: $D_{n,7}\ D_{n,6}\ D_{n,5}\ D_{n,4}\ D_{n,3}\ D_{n,2}\ D_{n,1}\ D_{n,0}$

Nota:

Las versiones de HAVAL que han sido rotas (encontrado colisiones) son:

- HAVAL-128 con cualquier número de pasadas.
- HAVAL de 3 rondas con cualquier longitud de hash value
- El modo de cifrado de 4 y 5 pasadas de HAVAL.

El estado de las demás versiones es incierto, pero parece admisible para todas las variantes de esta función.

9. BIBLIOGRAFIA

Papers y libros

[1] Alfred J. Menezes, Paul C von Oorschot y Scott A. Vanstone, "*Handbook of Applied Cryptography*", CRC Press, 1997. 780 pp.

[2] "*New European Schemes for Signatures, Integrity, and Encryption*.", Versión 0.15 (beta), abril 2004, pp. 829, http://www.cryptonessie.org

[3] Douglas R. Stinson, "*Cryptography Theory and Practice*", 1995, CRC Press, pp. 434.

[4] 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.

[5] Hans Dobbertin, Antoon Bosselaers y Bart Preneel, "*RIPEMD-160, a strengthened version of RIPEMD*", Abril 1996, pp. 13.

[6] R. L. Rivest, "*The MD4 Message-Digest Algorithm*", en *Advances in Cryptology – Crypto'90* (A. Menezes y S. A. Vanstone, eds.), no. 537 en Lecture Notes in Computer Science, pp. 303-311, Springer-Verlag, 1991.

[7] FIPS 180-2, "*Secure Hash Standard (SHS)*.", National Institute of Standards and Technology, Agosto 2002, pp. 71.

[8] Yuliang Zheng, Josef Pieprzyk y Jennifer Seberry, "*HAVAL – a one-way hashing algorithm with variable length of output (Extend Abstract)*", en *Advances in Cryptology*

Auscrypt'92 (J. Seberry e Y. Zheng, eds.), no. 718 en Lecture Notes in Computer Science, pp. 83-104, Springer-Verlag, 1993.

[9] Paulo S. L. M. Barreto y V. Rijmen, "*The Whirlpool hashing function*". Primitive submitted to NESSIE, Septiembre 2000, Disponible en http://www.criptonessie.org .

[10] 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.

[11] Jiri Fridrich, "*Symmetric Ciphers Based On Two-Dimensional Chaotic Maps*", International Journal of Bifurcation and Chaos, Vol. 8, No. 6 (1998) 1259-1284.

[12] Darrel Hankerson, Alfred Menezes y Scott Vanstone, *"Guide to Elliptic Curve Cryptography"*, pp. 311, Springer-Verlag, 2004.

RFCS

[RFC-1320] R. L. Rivest, "*The MD4 Message-Digest Algorithm*". IETF Request for Comments, RFC 1320, Apr. 1992

[RFC-1321] R. L. Rivest, "*The MD5 Message-Digest Algorithm*", IETF Request for Comments, RFC 1321, Abril 1992, pp. 21.

"The hash function RIPEMD-160.".
http://www.esat.kuleuven.ac.be/~bosselae/ripemd160.html