Universidad Nacional Autónoma de México
Posgrado en Ciencia e Ingeniería de la Computación
Redes y seguridad en cómputo

Sistema de cifrado inspirado en cubos de Rubik, resistente a ataques de fuerza bruta mediante procesamiento paralelo

Tesis
que para optar por el grado de:
Maestro en Ciencias
(Computación)

presenta:
Ing. Carlos Enrique Quijano Tapia

Tutor:
Dr. Vladislav Khartchenko, FES-Cuautitlán

México, D. F. Marzo 2015

Jurado asignado:

Presidente: Dr. Francisco Javier García Ugalde

Secretario: Dr. Jorge Luis Ortega Arjona

Vocal: Dr. Vladislav Khartchenko

1^{er} suplente: Dra. Maria Elena Larraga Ramirez

2^{do} suplente: Dr. David Tinoco Varela

Lugar o lugares donde se realizó la tesis:
Facultad de Estudios Superiores de Cuautitlán

Tutor de tesis:
Dr. Vladislav Khartchenko

Firma

Dedicado a
Cristian, Xipe y Yatzil

Agradecimientos

A lo largo del tiempo, tantas personas han tenido influencia significativa en mi vida, me han llenado de experiencias y con el tiempo he aprendido a tomar lo mejor y lo peor de cada una de ellas, ayudándome a crecer y reflexionar sobre el rumbo mismo de mi existencia. Les debo mi pasado, presente y futuro ser. Son tantas las personas y tantos los agradecimientos, debo tanto que no se como pagar, y no me queda más que resumirlos así:

A mi esposa y mi mejor amiga por estos años de compañía, consejo y apoyo

A mis hijos por su cariño

A mis padres por su vida de dedicación

A mi suegra y mis cuñados por aceptarme en su familia y brindarme su apoyo

A mis hermanos por los años que hemos compartido y lo que aún aprendo con ellos

A mis maestros y a mi tutor por su paciencia

A mis amigos por lo que me enseñaron

A mis compañeros por sus consejos

Al Dr. Vladislav Khartchenko (SNI Nivel III, expediente 18740), por las enseñanzas y oportunidades otorgadas

Al proyecto IN112913-PAPIIT “Estructuras algebraicas relacionadas a los grupos cuánticos”, por la oportunidad de participar en él

Y por supuesto al CONACyT cuyo financiamiento hizo posible todo este trabajo, ya que de otra manera no se hubiese podido sacar adelante con mis propios medios

A TODOS USTEDES ¡GRACIAS!

Índice general

Índice de figuras

Índice de tablas

Índice de algoritmos

Capítulo 1

Introducción

1.1. Resumen

Este trabajo surgió bajo la hipótesis de que los sistemas criptográficos modernos se han diseñado con el propósito de ser lo más eficientes y rápidos posibles, siendo una desafortunada consecuencia indirecta que se puedan explorar grandes cantidades de llaves por segundo. Fácilmente el lector podrá argumentar que este problema es irrelevante, debido a que la mayoría de los sistemas criptográficos tienen un universo de llaves tan grande que la exploración completa del mismo requeriría tiempos ridículamente grandes, haciendo inviable la explotación de esta vulnerabilidad y estarían en lo correcto, pero sólo para el caso en que la llave se escogiese en forma completamente aleatoria, minimizando el efecto del factor humano.

Sí tenemos la paciencia para estudiarnos, descubriremos que somos criaturas predecibles, que tienen hábitos muy arraigados y suelen utilizar patrones de forma consciente o inconsciente para casi todo, aunque no nos guste admitirlo. Este hecho es ampliamente explotado por analistas forenses informáticos y criminales cibernéticos que mediante diversas técnicas priorizan las zonas de dicho universo de llaves y virtualmente reducen el universo de búsqueda. Uno puede pensar que podemos usar diversas técnicas para construir llaves más seguras, pero la realidad es que todas ellas no son más que transformaciones basadas en ciertos patrones más o menos regulares y estas técnicas son ampliamente conocidas.

Sí a este escenario le agregamos el cómputo de propósito general en unidades de procesamiento de gráfico, el constante abaratamiento del hardware y el aumento de su velocidad, las redes de equipos de cómputo y los nuevos paradigmas que puedan surgir, el número de llaves exploradas simultáneamente por segundo pueden crecer significativamente. Entonces nos enfrentamos ante la posibilidad de que muchos los sistemas criptográficos no sean tan

seguros como creemos, por descuidar el factor humano.

En este trabajo construiremos un nuevo algoritmo de cifrado que permitirá limitar la efectividad de ataques de fuerza bruta que prioricen llaves y se apoyen en cómputo paralelo, pero nos enfocáremos en tres ejes fundamentales: aprovechar la robustez de los sistemas criptográficos existentes, limitar el cómputo simultáneo de llaves e inducir un retraso artificial entre cada verificación de llave, para lograr esto último se utilizarán operaciones costosas en términos de cómputo y en grandes cantidades, además de introducir la posibilidad de personalizar el protocolo de cifrado, dificultando el criptoanálisis, ya que cada variante se volverá un caso único y esto lo encarecerá significativamente, restringiendo la posibilidad de utilizarlos.

1.2. Hipótesis

Este trabajo contempla la posibilidad de que alguna entidad tenga suficiente información sobre otra, como para poder hacer una buena aproximación sobre la forma en que se construyen las llaves usadas para cifrar su información, siempre y cuando dichas llaves no sean generadas al azar.

En este contexto intentaremos responder las siguientes preguntas:

1. ¿Podemos crear un algoritmo que nos permita reducir el número de llaves que puedan ser verificadas en un tiempo determinado y que su complejidad sea lineal?

2. ¿Podemos hacer que dicho algoritmo sea personalizable?, es decir, que cada entidad pueda crear su propia variante sin que esto implique necesariamente cambiar el algoritmo. De esta manera si una variación del algoritmo llega a ser vulnerada, ello no implica necesariamente que se vulneren otras.

3. ¿Podemos hacer que dicho algoritmo sea extensible?, es decir, que una entidad pueda extender embebiendo otros algoritmos de cifrado dentro de éste y que dicho procedimiento sea fácil de implementar.

1.3. Objetivos

1. Construir un nuevo algoritmo de cifrado que permita limitar la efectividad de ataques de fuerza bruta que prioricen llaves y apoyados en cómputo paralelo.

2. Enfocarse en limitar el cómputo simultáneo de llaves e inducir un retraso artificial entre cada verificación de llave.

3. Introducir la posibilidad de personalizar el protocolo de cifrado, dificultando el criptoanálisis y volviendo cada variante un caso único, encareciéndolo significativamente, lo que restringirá la accesibilidad de los mismos.

4. Dejar abierta la posibilidad de embeber otros sistemas criptográficos *véase la sección 1.4 "Alcance"*.

1.4. Alcance

En este trabajo se definirá como parte del cifrado una serie de permutaciones simples de bloques que recuerdan al cubo de Rubik, pero adicionalmente a éstas, también se podrá aplicar como permutación de bloque cualquier otro algoritmo de cifrado, sí bien se explicará el procedimiento para embeber dichos algoritmos, no se estandarizará, es decir asignándole una posición y nomenclatura de token *véase capítulo 3 "Construcción del Algoritmo"*, ya que esto se consideró una característica extra que escapa a la analogía del cubo de Rubik y en caso de tener que intercambiar información cifrada entre dos entidades, bastaría con ponerse de acuerdo sobre estas modificaciones.

Como efecto secundario de esta característica, sí no se tiene que intercambiar información cifrada con otra entidad, estas variantes con diferentes algoritmos embebidos e incluso con un orden distinto, serían difíciles de replicar, lo que puede reforzar su seguridad.

1.5. Métodos

1. Se inducirá al uso intensivo de recursos de cómputo, pero manteniendo una complejidad lineal, a fin de reducir la capacidad de exploración de llaves.
2. Limitar el cómputo paralelo al explorar las llaves:
   a) Seccionar los bloques de operaciones que se puedan calcular en forma paralela, induciendo dependencia de cálculos, ya que dos tareas no pueden ser paralelizadas sí los parámetros de entrada de una tarea dependen de la salida de la otra.
   b) Propiciar que las operaciones susceptibles a ser calculadas en forma paralela lo hagan en bloques muy pequeños, preferentemente requiriendo mucho tráfico entre procesador y memoria, y que a su vez generen mucha comunicación para coordinación. De esta manera se busca congestionar las comunicaciones del procesamiento, esto ayudará a limitar el número de operaciones simultáneas que pueda ejecutar un equipo de cómputo, además de exigir mayores recursos que podrían encarecer la verificación de llaves.
   c) Limitar la ganancia obtenida al calcular paralelamente las secciones que no se pueda evitar calcular en forma paralela, aplicando la Ley de Amdahl *véase la sección 2.3.1.*
3. Personalización. Se busca permitir un cifrado dinámico dentro del algoritmo, bajo la premisa de "si una variación del algoritmo llega a ser vulnerada, ello no necesariamente implica que se vulneren otras", así se volverán costosos los recursos requeridos para hacer los criptoanálisis necesarios para vulnerar una variación en particular.

   Para esto se va a generar un archivo llave f_{k}, que a su vez, será interpretado como un script de cifrado y descifrado s_{k}, similar a la verificación en dos pasos[1]. Es decir el esquema tradicional de cifrado es una función de transformación $t()$ que recibe un mensaje

[1]La verificación en dos pasos es conocida como "algo que se sabe y algo que se tiene", es decir lo que "se sabe" es la llave y lo que "se tiene" es algo que tienes en tu poder, generalmente un dispositivo que emite firmas electrónicas distintas en intervalos de tiempo definidos

m y una llave k, lo que genera un mensaje transformado m_t de la forma $m_t = t(m, \mathrm{k})$; lo que se propone es cambiarlo a una función de transformación $t()$ que reciba una función de transformación $t_{s_\mathrm{k}}()$ con la forma $m_t = t_{s_\mathrm{k}}(m, \mathrm{k})$, siendo $t_{s_\mathrm{k}}()$ una función de transformación construida a partir del script s_k como se verá en la *figura 3.1 de la página 19*.

Usar ésta pseudo-verificación en dos pasos podría permitir implementar nuevas y diversas políticas de seguridad orientadas a resguardar el archivo llave, que de estar bien diseñadas y aplicadas, incrementarían el trabajo de un intruso, ya que éste tendría que vulnerar más sistemas para poder comprometer información.

4. Construir en forma dinámica un *Generador de Flujos de Bits Pseudo-aleatorios Personalizable (GFBPP) véase sección 3.2*, que dependa de la llave k y del archivo llave f_k, usando Autómatas Celulares (AC) *véase sección 2.4*, mismo que será empleado para:

   a) "Ensuciar" o "enmascarar" los datos con el uso del operador *XOR* como primer paso del cifrado, dificultando así su criptoanálisis, ya que para "limpiar" los datos, primero se deberá poder replicar el *GFBPP* y encontrar los estados iniciales o semilla del mismo.

   b) Generar los argumentos para las funciones de permutación.

1.6. Antecedentes

Muchas veces se ha intentado vulnerar o debilitar sistemas criptográficos, para fines de este trabajo interesa centrarse en limitar aquellos intentos que donde se tenga información suficiente que permita encontrar relativamente rápido la llave usada, basándose en el uso de cómputo paralelo. Un ejemplo relativamente reciente (Diciembre de 2008) fue cuando lograron comprometer la seguridad del protocolo SSL utilizando consolas de *PlayStation 3*[5] aprovechando sus GPU's, estos ataques básicamente lo que hacían era calcular en paralelo firmas digitales del *hash criptográfico MD5*. Sí bien, encontrar llaves que correspondan a una huella digital hash conocida no es lo mismo que encontrar una llave de un algoritmo de cifrado, lo importante es el uso de cómputo en paralelo para encontrar llaves, ya que teniendo un buen sistema para crackear cifrados, sólo es cuestión de recursos montar una infraestructura más potente[15], ya que muchos de estos sistemas están disponibles en el mercado, en forma legal o no.

En este contexto se han formado profesionales cuya especialidad es desbloquear (descifrar) archivos cifrados y también se han creado múltiples técnicas para lograrlo, algunas de las más interesantes consisten en la investigación profunda del objetivo humano para determinar las llaves más probables[3].

También se ha hablado de enormes infraestructuras dedicadas en forma exclusiva a descifrar información, con capacidades que caen en el terreno de la especulación[14], más allá de su capacidad y tamaño real, es importante señalar que el avance del hardware hará que sean cada vez más comunes y aumentando la capacidad de exploración de posibles llaves.

1.7. Estructura de la tesis

En el *capítulo 2 "Marco Teórico"* exploraremos algunos de los conceptos fundamentales requeridos para entender este trabajo, pasando por Criptografía, Cómputo Paralelo, Autómatas celulares y Teoría de grupos.

En el *capítulo 3 "Construcción del algoritmo"* iremos modelándolo paso a paso.

En el *capítulo 4 "Resultados"* se presentan los Análisis de la aleatoriedad del *GFBPP* y los vectores de prueba para cada una de las permutaciones. No se presentan vectores de prueba del cifrado, ya que no se encontró una buena forma de representarlos en los límites físicos de la hoja de papel, pero se incluye un script de prueba en el disco adjunto a esta tesis.

En el último capítulo de presentan las discusiones y conclusiones.

Finalmente se presentarán Anexos que incluyen dos plantillas para construir los cubos, para facilitar el entendimiento de las permutaciones.

Capítulo 2

Marco teórico

2.1. Resumen

En este capítulo exploraremos brevemente algunos de los conceptos fundamentales necesarios para entender este trabajo: la *sección 2.2 "Criptografía"* hablará sobre las generalidades de los sistemas criptográficos, la *sección 2.2.1 "Cifrado de Vernam"* hablará sobre dicho método que será usado en el algoritmo propuesto, la *sección 2.2.2 "Ataques de fuerza bruta"* hablará sobre dichos ataques pero enfocándose en la construcción de diccionarios, la *sección 2.3 "Cómputo Paralelo"* tratará sobre conceptos básicos y los tipos de paralelismo para poder conocer las limitaciones del cómputo paralelo que se aprovechan en este algoritmo, la *sección 2.4 "Autómatas Celulares"* hablará sobre los conceptos básicos y propiedades de los *AC*, que son la base para construir el *GFBPP* y finalmente la *sección 2.5 "Teoría de Grupos"* tratará sobre los conceptos básicos referentes a las propiedades que permitirán aplicar grupos de permutaciones, ya que parte de la propuesta es usar grupos de permutaciones para proteger la información .

2.2. Criptografía

La criptografía es una disciplina cuyo objetivo es obtener confidencialidad en la comunicación, de manera que sea ésta ininteligible para cualquier entidad no autorizada que pudiera interceptar la comunicación o información.

La confidencialidad es resguardada con el uso de una llave, misma que es considerada como un secreto. En el contexto de la criptografía en cómputo, una llave estará constituida de información, ya sea con datos de un valor determinado o el resultado de un cálculo, digamos la interpretación de un lector de huellas digitales de una huella humana.

Basándose en el tipo de llave existen dos clasificaciones de criptografía:

- Simétrica. La criptografía simétrica tiene la forma (2.1) para el cifrado y (2.2) para el descifrado, además se caracteriza porque la llave para cifrar y la llave para descifrar son la misma.

$$m_t = t(m, \mathrm{k}) \tag{2.1}$$
$$m = t^{-1}(m_t, \mathrm{k}) \tag{2.2}$$

  donde

  m = Mensaje en claro

  m_t = Mensaje transformado

  $t()$ = Función de transformación o cifrado

  $t^{-1}()$ = Función de transformación inversa o descifrado

  k = Llave

  El principal problema de este tipo de sistema es el intercambio y distribución de llaves.

- Asimétrica. La criptografía asimétrica o de clave pública consiste en dos llaves distintas con la forma (2.3) para el cifrado y (2.4) para el descifrado. En este tipo de criptografía sí un mensaje es cifrado con una llave k_x, se necesitará la opuesta k_y para descifrarlo

$$m_t = t(m, \mathrm{k}_x) \tag{2.3}$$
$$m = t^{-1}(m_t, \mathrm{k}_y) \tag{2.4}$$

  donde

  m = Mensaje en claro

  m_t = Mensaje transformado

  $t()$ = Función de transformación o cifrado

  $t^{-1}()$ = Función de transformación inversa o descifrado

  k_x = Llave del emisor

  k_y = Llave del receptor

  El principal problema de este sistema es la autenticación, es decir la identificación inequívoca del emisor y receptor.

2.2.1. Cifrado de Vernam

El cifrado de Vernam es un sistema de cifrado de flujo[16], es decir: dado un mensaje, se genera un flujo de datos del mismo tamaño que el mensaje y éste es usado para cifrar un mensaje mediante la operación XOR. El flujo debe ser una secuencia de datos pseudo-aleatorios y la seguridad de éste sistema estará determinada por la aleatoriedad del flujo.

Una variante del cifrado de Vernam es la libreta de un sólo uso, la diferencia radica en que el flujo de datos es aleatorio, es decir no se puede predecir y por ello se ha demostrado matemáticamente que es irrompible[13], ya que existe la misma probabilidad de generar cualquier otro mensaje. Desafortunadamente en un sistema determinista como los sistemas de cómputo, no es posible implementarla, ya que la aleatoridad que se puede producir en estos sistemas, también es determinista y por tanto sólo es una pseudo-aleatoriedad.

2.2.2. Ataques de fuerza bruta

Un ataque de fuerza bruta consiste en la verificación secuencial de llaves hasta encontrar la aquella que pueda descifrar un mensaje deseado. La viabilidad de un ataque de este tipo depende del universo de llaves posibles y la velocidad con que estas pueden ser verificadas. En general un sistema criptográfico se considera seguro en base al margen de tiempo requerido para encontrar la clave k.

Un ataque de fuerza bruta por sí mismo no es amenazante, idealmente si una llave es elegida aleatoriamente y entre mayor sea su longitud, menos probable será encontrarla en un tiempo de razonable. El problema es que el mismo puede reducirse significativamente gracias a diversos factores (no excluyentes) los cuales no son:

1. Vulnerabilidades en el sistema criptográfico o en la implementación del mismo. Si bien las vulnerabilidades son un problema grave, su detección dependerá de muchos factores intrínsecos a cada algoritmo, su implementación y al analista; sin embargo, se puede ilustrar el problema con un ejemplo sencillo:

   Suponiendo que se usa un cifrado por desplazamiento de caracteres en el alfabeto (también conocido como cifrado de Cesar) y sí se conoce una parte y ubicación en el mensaje que se desea descifrar, se puede procesar solamente esa parte hasta obtenerla en texto en claro. De esta manera no se analiza todo el mensaje cifrado y se obtendrá un conjunto de llaves candidatas que se podrán verificar contra todo el mensaje.

2. Reducción y priorización de llaves. Para lograr esto se recurre a muchas técnicas de investigación, entre las que destaca la construcción de diccionarios[1], potenciado gracias al análisis de datos extraídos mediante técnicas de *cómputo forense* o *análisis de datos públicos*.

3. Incremento sustancial del poder de procesamiento. Esta parte es muy diversa, ya que se puede incrementar el poder de procesamiento usando cómputo en paralelo, hardware

[1]Existen también herramientas que permiten construir diccionarios con las variantes de sustitución de caracteres más comunes, por ejemplo *Rsmangler*

más potente, Circuitos Integrados de Aplicación Específica (ASIC) por sus siglas en ingles o todos los anteriores.

En este contexto una persona que rompe sistemas de seguridad o Cracker, podría lograr obtener acceso a información protegida en un tiempo razonable.

2.3. Cómputo Paralelo

- Procesamiento Secuencial
  - 1 Procesador
  - 1 Proceso
- Procesamiento Concurrente
  - 1 Procesador
  - m Procesos
- Procesamiento Paralelo y Procesamiento Distribuido
  - n Procesadores
  - m Procesos

Figura 2.1: Clasificación de procesamiento

Un *proceso* se puede definir como la manipulación de la memoria por una Unidad de Procesamiento (UP) y se puede clasificar en secuencial, concurrente, paralelo y distribuido. Esta clasificación se debe principalmente a la relación entre UP′s y procesos, como se ve en la figura 2.1. En el caso del *procesamiento paralelo* y el *procesamiento distribuido*, la diferencia radica en que en el *procesamiento distribuido* las UP′s se encuentran en diferentes equipos y que la capacidad de comunicación suele ser inferior, aunque hoy en día la diferencia es muy sutil.

Para fines de este trabajo, se tratará indistintamente al cómputo distribuido y al paralelo, ya que no es relevante para el objetivo si los procesos se encuentran en el mismo equipo o no, por tanto serán referidos como paralelos indistintamente.

La parte importante del cómputo paralelo es la optimización de recursos, lo cual tiene sus retos, ya que su diseño y programación requiere considerar diversos elementos como la coordinación, dependencias y arquitectura del hardware. La complejidad de estas tareas ha favorecido la creación de metodologías y patrones de diseño para software paralelo[8], con el fin de facilitar el diseño. Para poder optimizar los recursos, se suele buscar que los procesos trabajen en una tarea en común, esto implica que pueden clasificarse en *procesamiento disjunto* (no necesitan comunicarse entre sí) y *procesamiento cooperativo* (sí se comunican entre sí y requieren algún tipo de sincronización para evitar corromper sus datos).

Sin entrar en una metodología específica, se puede decir que sí existe alguna manera de paralelizar una tarea, generalmente se comienza analizando su *cadena crítica*, que ayuda a ver las dependencias presentes en la tarea. Con base en ello se puede determinar que partes se pueden paralelizar y cuáles no.

Una vez ubicadas las partes que se pueden hacer el paralelo, se puede identificar cualitativamente el tipo de *granularidad* de la implementación, ya sea *granularidad fina* (pocas

operaciones, muchas comunicaciones y coordinación) o *granularidad gruesa* (muchas operaciones, pocas comunicaciones y coordinación).

Otra parte importante a la hora de diseñar en paralelo es balancear: el trabajo, la coordinación, la comunicación y la sincronización, ya que esto determinará la distribución de tareas con el objetivo de lograr un óptimo uso de recursos.

Con base en todo lo anterior, se podrá elegir el *patrón* de paralelización que mejor se ajuste a los requerimientos del problema. Y finalmente, se podrá estimar la ganancia obtenida con ayuda de la *Ley de Amdahl* que se explica a continuación.

2.3.1. Ley de Amdahl

"La mejora obtenida en el rendimiento de un sistema debido a la alteración de uno de sus componentes está limitada por la fracción de tiempo que se utiliza dicho componente"[17].

En base a esta ley se puede estimar la ganancia o aceleración máxima de un sistema, que se obtiene al paralelizar una fracción del mismo, con ayuda de la siguiente ecuación:

$$A = \frac{1}{(1 - F_m) + \frac{F_m}{A_m}} \tag{2.5}$$

siendo:

- A = Aceleración final
- F_m = Porcentaje de tiempo de ejecución secuencial que ocupa el subsistema mejorado respecto a todo el sistema
- A_m = Aceleración como factor de mejora (1.5x, 2x, 3x, etc) inducido en el subsistema mejorado

2.4. Autómatas Celulares

Un *Autómata Celular (AC)* es una colección de células con estados determinados en una malla de forma específica, que evoluciona a través de un número de pasos de tiempo discretos acorde a un conjunto de reglas basadas en los estados de las células vecinas. Las reglas se aplican iterativamente para tantos pasos de tiempo como se desee[19]. Respecto a las reglas que obedecen, éstas pueden ser homogéneas (todas las células son regidas por la misma regla) o heterogéneas (no todas las células son regidas por la misma regla).

Los *autómatas celulares unidimensionales* son aquellos en que la malla esta definida en una sola dimensión, es decir, las células están organizadas como un arreglo.

Los *autómatas celulares elementales* son la clase más simple de *AC's* unidimensionales, tienen dos valores posibles para cada célula (0 y 1) y las reglas dependen solamente de los

estados de los vecinos más cercanos. Como resultado la evolución de un autómata celular elemental puede ser descrita completamente por una tabla que especifica el estado que una célula determinada tendrá en la próxima generación basado en el valor de la célula a su izquierda, el valor de la misma célula y el valor de la célula a su derecha. Puesto que hay $2x2x2 = 2^3 = 8$ posibles estados binarios para las tres células vecinas una célula dada, hay un total de $2^8 = 256$ autómatas celulares elementales, cada uno de los cuales se puede indexar con número binario de 8 *bits*[20].

Dado que no existe definición formal aceptada para los AC, se usará una aproximación al sistema de Lindenmayer[18] para definirlo:
sea

$$\mathbf{C} = \{V, S, \Delta, \Phi, \Theta(\theta), \gamma\} \tag{2.6}$$

donde

$V = \{0, 1\}$ Alfabeto

$S =$ Conjunto de células

$\Delta =$ Conjunto de estados del conjunto S o estado inicial

$\Phi = \{\phi_0, \phi_1, \ldots, \phi_{|S|-1}\}$ Es el conjunto de reglas para el conjunto S, con $|V|^{|V|^\theta} = 2^{2^\theta}$ reglas distintas posibles

$\Theta() =$ Es la configuración de la vecindad para cada $s \in S$ y la definimos

$\theta =$ magnitud de la vecindad $\theta \in \mathbb{Z}$ y $\theta \neq 0$

$\gamma =$ Condición de frontera *(explicada enseguida).*

En cómputo dado que los recursos son limitados los *AC* deberá estar definidos en un espacio finito y también se debe establecer una de las siguientes condiciones de frontera $\Gamma = \{\gamma_1, \gamma_2, \gamma_3, \gamma_4, \gamma_5\}$:

1. $\gamma_1 =$ **Frontera abierta**. Las células mas allá de la frontera tienen estados fijos.
2. $\gamma_2 =$ **Frontera periódica**. Las células de los extremos opuestos se tocan.
3. $\gamma_3 =$ **Frontera adiabática**. Las células dentro de la frontera están aisladas.
4. $\gamma_4 =$ **Frontera reflejada**. Las células dentro de la frontera se reflejan fuera de esta,
5. $\gamma_5 =$ **Sin Frontera**. Las fronteras se amplían en forma dinámica.

Cada evaluación del conjunto S se define como una generación $\mathrm{G} = \{g_1, g_2, \cdots, g_\infty\}$.

Las reglas se evalúan leyendo los estados de los vecinos de la célula s_n de la generación g_m como se ve en la *figura 2.2* y calcula el nuevo estado de la célula s_n para la generación g_{m+1}.

De acuerdo a el comportamiento general de un AC elemental que evoluciona con reglas homogénea determinada Stephen Wolfram[21] generó una clasificación:

Figura 2.2: Ejemplo de AC de vecindad $\Theta(3)$

1. **Clase I**. Produce estados homogéneos, sin importar los estados iniciales.
2. **Clase II**. Genera patrones simples y periódicos.
3. **Clase III**. Genera patrones caóticos.
4. **Clase IV**. Genera patrones complejos.

EJEMPLO **2.1** *Si definimos un AC unidimensional donde $\theta = 3$, con $|V|^{|V|\theta} = 2^{2^3} = 2^8 = 256$ reglas distintas, estas se evalúan como se ve en la tabla 2.1.*

G		7_{10}	6_{10}	5_{10}	4_{10}	3_{10}	2_{10}	1_{10}	0_{10}
g_m	**Vecinos**	111_2	110_2	101_2	100_2	011_2	010_2	001_2	000_2
g_{m+1}	**Salida**	0_2	0_2	0_2	1_2	1_2	1_2	1_2	0_2

Tabla 2.1: Regla $30_{10} = 00011110_2$

2.5. Teoría de Grupos

Un grupo es una estructura algebraica que consta de un conjunto G con una operación binaria, que en notación multiplicativa es llamada multiplicación[2]:

$$ab = c \tag{2.7}$$

dicha multiplicación de grupo debe satisfacer las siguientes condiciones:

1. Asociatividad $(ab)c = a(bc)$, $\forall a, b, c \in G$
2. Existencia del elemento neutro $\exists e \in G : ea = ae = a$
3. Existencia del elemento inverso $\forall a \in G \; \exists a^{-1} \in G : aa^{-1} = a^{-1}a = e$

2.5.1. Grupos cíclicos

Se dice que un grupo es cíclico sí: $\exists a \in G : 1, a, a^2, a^3, \ldots = G$, es decir a es generador de G o $\langle a \rangle = G$.

[2]No confundir con la multiplicación aritmética

2.5.2. Grupos de permutación

Sea un grupo de permutación[1] escrito en notación matricial

$$\pi = \begin{pmatrix} a_1 & a_2 & \dots & a_n \\ a_2 & a_3 & \dots & a_1 \end{pmatrix} \tag{2.8}$$

se dice que es dependiente porque puede escribirse como un sólo ciclo en su notación de ciclos $\pi = (a_1 a_2 a_3 \dots a_n) = (a_2 a_3 \dots a_n a_1) = (a_3 \dots a_n a_1 a_2) = \dots$.

En cambio el ciclo

$$\sigma = \begin{pmatrix} a_1 & a_2 & \dots & a_n & b_1 & b_2 & \dots & b_m \\ a_2 & a_3 & \dots & a_1 & b_2 & b_3 & \dots & b_1 \end{pmatrix} \tag{2.9}$$

se dice que es independiente porque **no** puede escribirse como un sólo ciclo en su notación de ciclos $\sigma = (a_1 a_2 a_3 \dots a_n)(b_1 b_2 b_3 \dots b_m)$ y $\{a_1 a_2 a_3 \dots a_n\} \cap \{b_1 b_2 b_3 \dots b_m\} \neq \emptyset$.

en ambos casos la evaluación de cada permutación se calcula buscando el valor y cambiando por valor colocado en la parte inferior (para la notación matricial) o por la siguiente (en la notación por ciclos).

LEMA **2.1** *Las permutaciones cíclicas independientes permutan.*

TEOREMA **2.1** *Cada permutación es un producto de permutaciones cíclicas (Dubreil1975 véase teorema 4, página 77).*

Capítulo 3

Construcción del algoritmo

3.1. Resumen

En este capítulo se mostrará el proceso de construcción del algoritmo de cifrado propuesto.

En la *sección 3.2* se construirá el *Generador de Flujos de Bits Pseudo-aleatorios Personalizable (GFBPP)* que se propone para el algoritmo de cifrado y descifrado como una forma sencilla de generar flujos de bits pseudo-aleatorios, este algoritmo es un Autómata Celular (AC) con reglas dinámicas.

En la *sección 3.3* se especificarán los parámetros modificables y sus valores por defecto.

En la *sección 3.4* se describirá la estructura general del cifrado, para finalmente ir construyendo los sub-algoritmos mediante un enfoque *top-down*.

Finalmente en la *sección 3.5* se construirán los algoritmos de permutación que fueron inspiradas en el cubo de Rubik.

3.2. Generador de Flujo de Bits Pseudo-aleatorios Personalizable

En ésta parte del trabajo se plantea de una forma poco ortodoxa y polémica de generación de flujos de bits pseudo-aleatorios. Para ello se empleará un AC unidimensional de frontera periódica, vecindad θ= 3 y reglas heterogéneas. Lo que se hará diferente será construir cada célula con un conjunto de reglas dinámicas Ψ, en lugar de una sola regla ϕ, es decir:

$$\Phi = \{\Psi_0, \Psi_1, \ldots, \Psi_{|S|-1}\} \tag{3.1}$$
$$\Psi = \{\phi_0, \phi_1, \ldots, \phi_{|\Psi|-1}\} \tag{3.2}$$

o uniendo todos los elementos

$$\begin{aligned}\Phi = \; & \{\{\phi_{[0,0]}, \phi_{[0,1]}, \ldots, \phi_{[0,|\Psi_1|-1]}\}, \\ & \{\phi_{[1,0]}, \phi_{[1,1]}, \ldots, \phi_{[1,|\Psi_2|-1]}\}, \\ & \ldots, \\ & \{\phi_{[|S|-1,0]}, \phi_{[|S|-1,1]}, \ldots, \phi_{[|S|-1,|\Psi_n|-1]}\}\}\end{aligned} \tag{3.3}$$

en vez de $\Phi = \{\phi_0, \phi_1, \ldots, \phi_{|S|-1}\}$. De esta forma la *ecuación* (2.6) define al AC para el GFBPP queda de la siguiente forma:

$$\mathbf{C} = \{V, S, \Delta, \Phi, \Theta(3), \gamma\} \tag{3.4}$$

Para determinar la regla que será evaluada, se deberá saber la generación actual del AC y obtener el residuo de la división de la generación con el tamaño del conjunto de reglas de la célula, así se podrá buscar esa posición en el listado de reglas de la célula y activarla, como se ve en el *algoritmo 3.2*. De esta forma se podrá calcular la regla correspondiente para todas las células usando el *algoritmo 3.1*

Algoritmo 3.1 acGenEval - Evaluación de generación del AC para GFBPP

Entrada: *ac*

1: $ac.g \leftarrow ac.g + 1$
2: $Delta \leftarrow 0$
3: **para toda** $s \in ac.S$ **hacer**
4: $\quad near \leftarrow cellNear(ac.S, s.n)$
5: $\quad phi \leftarrow calRule(ac.g, s)$
6: $\quad Delta \leftarrow cellEval(phi, near) << s.n$
7: **fin para**
8: $ac.setDelta(Delta)$

Algoritmo 3.2 calRule - Obtiene la regla de la célula para la generación actual

Entrada: g generación, s célula que se esta evaluando
Salida: *phi*

1: **devolver** $s.Psi[g \ mod \ size(s.Psi)]$

3.2.1. Justificación

Basados en la clasificación de Wolfram[21] sabemos que algunas reglas tienen comportamiento caótico y artículos como Seredynski(2003)[11], Zomaya(2004)[22] y Benkiniouar(2004)[2] nos muestran que la calidad de los flujos pseudo-aleatorios producto de un AC dependerá del conjunto de reglas aplicadas, pero la hipótesis polémica de esta propuesta es que un generador de flujos de bits con reglas dinámicas, tiene un comportamiento suficientemente complejo como para que se pueda elegir aleatoriamente "casi" cualquier conjunto de reglas, principalmente porque incrementa las combinaciones posibles de reglas, siendo un espacio de búsqueda mucho más amplio, además estas reglas cambian con cada generación.

Un beneficio extra al usar reglas dinámicas es que se aplicará el mismo conjunto de reglas hasta que se cumpla un ciclo. El tamaño de este ciclo estará definido como el Mínimo Común Múltiplo de todos los tamaños de los conjuntos de reglas, así a mayor cantidad de células, mayor tope máximo de reglas y a mayor cantidad de tamaños distintos, es mayor la posibilidad de que el ciclo sea más grande.

Debido a que obviamente algunos conjuntos no serán suficientemente aleatorios, se considera prudente usar las pruebas de aleatoriedad.

3.3. Parámetros modificables

Esta serie de algoritmos necesitará las siguientes variables de configuración, modificables por parámetros para permitir la personalización:

- **config.gfParam.initEvolutions** = 0. Indica las generaciones del Generador de Flujos de Bits Pseudo-aleatorios Personalizable (GFBPP) que serán descartadas.
- **config.maxIterations** = 32. Indica la cantidad máxima de permutaciones por cubo que se van a procesar. Aunque en este caso se eligió arbitrariamente un máximo de *32* rondas (por no dejar una cantidad demasiado pequeña), es necesario definir políticas para elegir este valor en base al tamaño, grado de confidencialidad de la información y el tiempo máximo que se esté dispuesto a esperar que se termine el procesamiento.
- **config.md.active** = $True$. Incluye el tamaño del mensaje en el md, en caso de estar desactivado incluye el tamaño del mensaje en el archivo cifrado.
- **config.tokens** = [$"dseta"$, $"mu"$, $"eta"$, $"fi"$, $"iota"$, $"ipsilon"$, $"lambda"$, $"ni"$, $"omega"$, $"rho"$, $"sigma"$, $"xi"$]. Listado de tokens que se van a usar, separados por comas. Al ser un simple listado, se puede aumentar la probabilidad de utilizar un tipo token en particular repitiéndolo, las veces deseadas.
- **parameters.crypt**. Booleano que indica si se va a cifrar mutuamente excluyente con *parameters.decrypt*.
- **parameters.decrypt** Booleano que indica si se va a descifrar mutuamente excluyente con *parameters.crypt*.
- **parameters.dependsOfKey** = $False$. Booleano que indica sí la posición de inicio de lectura en f_{k} depende de la llave. Sí está activo imposibilita el diseño del cifrado, ya que modifica el valor inicial de la variable $var.f_k.pos = SHA256(k) \vee 0xFFFF$, lo que hará completamente diferente el GFBPP al modificar la construcción de reglas, en vez de los estados iniciales como se verá en el *algoritmo 3.3*.
- **parameters.fk.name**. Nombre del archivo llave que se va usar.
- **parameters.inputFile**. Nombre del archivo de entrada.
- **parameters.k**. Llave de cifrado.
- **parameters.outputFile**. Nombre del archivo salida.

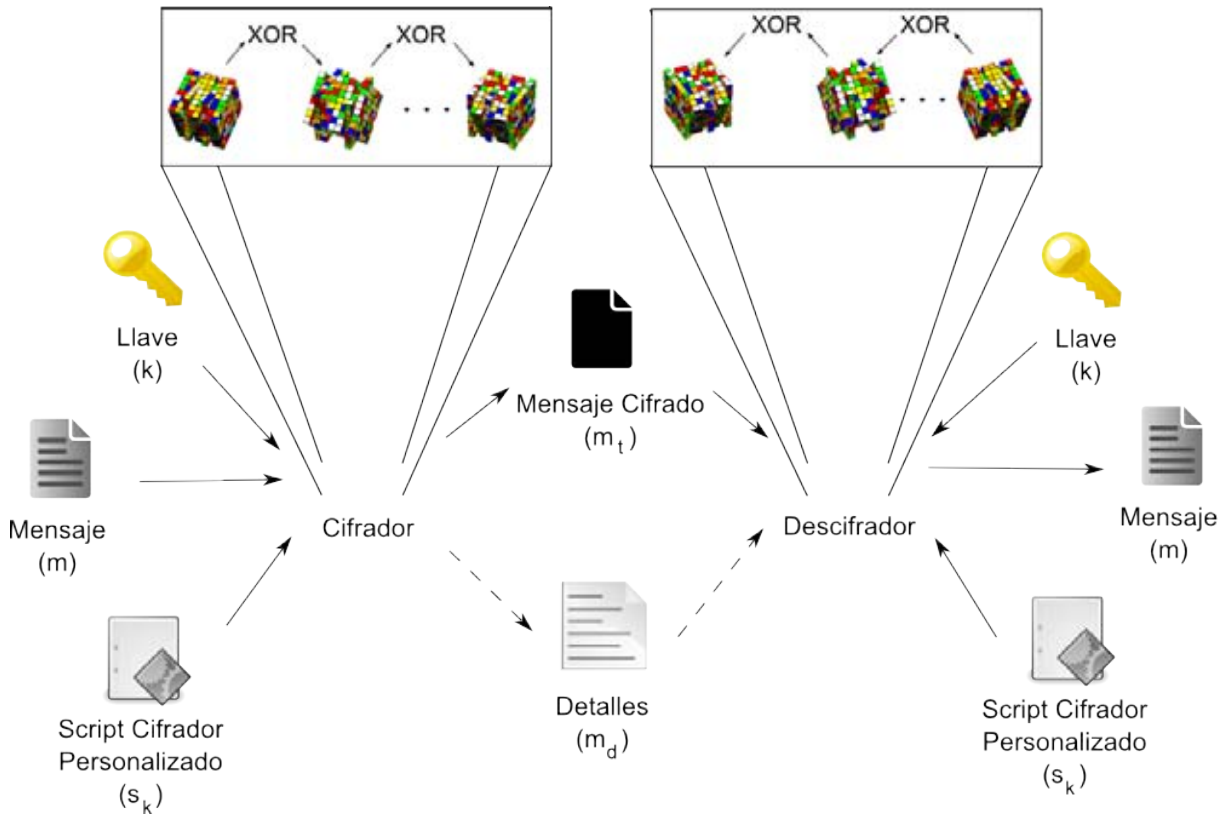


Figura 3.1: Diagrama general del Cifrado y Descifrado

3.4. Cifrado y Descifrado

La *figura 3.1* muestra de manera muy simplificada la idea general del cifrado. Como puede verse en la figura éste cifrado necesita un Script Cifrador Personalizado (s_k), este es generado a partir de un *archivo llave* (f_k) y una *llave* (k), de tal forma que s_k es generado por una función definida por la *ecuación* (3.5):

$$s_k = H(k, f_k) \tag{3.5}$$

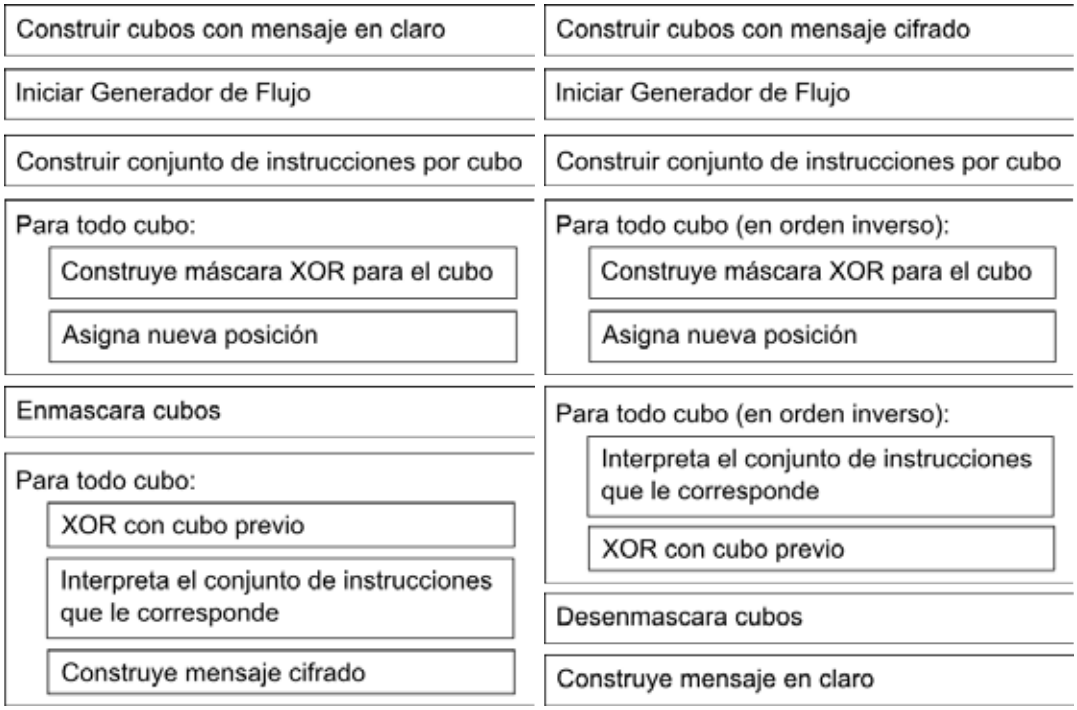
En la misma figura se representa el cifrado y descifrado compuesto por cubos, específicamente de *512 bits*, es decir de 8 bits por lado, cada uno permutado de manera distinta por s_k, pero precedido de una operación *XOR* con el cubo permutado anterior, esto se hace con el fin de obligar a resolver un cubo a la vez e impedir calcular en forma paralela los cubos. Algo que no se mostró en la figura, es que las posiciones de los cubos son desordenadas dependiendo de k y f_k, pero esto será definido más adelante.

Debido a que todo mensaje será segmentado en bloques de *512 bits*, los mensajes cifrados tendrán que ser redondeados hacia arriba para tener bloques completos, pero esto plantea un problema: ¿cómo se va a saber cuántos bytes deben ser descartados para obtener el mensaje original?, la única solución que se encontró fue almacenarlo y se definieron dos alternativas, que se describirán enseguida.

Como lo que se busca es otorgar al usuario la capacidad de personalizar este sistema de cifrado, en la *figura 3.1* puede verse que en forma opcional y paralela al mensaje cifrado, se

genera un archivo de detalles (*md*), este contiene de forma obligada el tamaño del mensaje original, pero también puede ser definido introduciéndose dicho tamaño en los últimos 8 bytes del último cubo, teniendo que ampliarse un cubo más, en caso de que no quedase suficiente espacio libre para almacenarlo en el último cubo, ambas características son mutuamente excluyentes. La razón para crear el archivo de detalles *md* es dificultar un ataque, siempre y cuando se pueda resguardar el archivo *md* en forma segura, pero esto sólo es un obstáculo complementario a las medidas de seguridad tomadas.

En las *figuras 3.2 (a) y (b)* podemos ver en forma más específica los algoritmos para cifrar y descifrar.



(a) Cifrado (b) Descifrado

Figura 3.2: Algoritmos de Cifrado y Descifrado

El algoritmo en general y de manera premeditada, no contiene ningún tipo de verificación que indique sí el mensaje descifrado es correcto o no, lo único que se puede usar para tal fin es el tamaño del mensaje de salida, que se describió previamente. El objetivo de esta medida es permitir que cualquier k y f_{k}, generen un mensaje de salida, consumiendo el tiempo de procesamiento que ello implica y obligando a verificar el mensaje por otros medios (verificando que el archivo abra correctamente, *huella digital*, *magic numbers*, ..., etc).

Respecto al orden de lectura de cubos y dentro de ellos, el formato utilizado es *little-endian*, lo que complica la verificación, pues el inicio del mensaje queda desplazado desde *0* hasta *504 bits* a la derecha, que aunado con el desorden de los cubos, dificultará la tarea.

El generador de flujo referido es el GFBPP y utiliza como semilla el *hash criptográfico SHA256* de la llave k, es decir $SHA256(k)$. El generador de flujo es construido e iniciado

utilizando el *algoritmo 3.3*.

Algoritmo 3.3 initGf - Construcción de generador de flujo

Entrada: $size$, $Delta$

```
1: gf.parameters ← new GFBPP(size)
2: gf.parameters.setDelta(Delta)
3: para todo s ∈ gf.parameters.S hacer
4:     s.Psi ← [ ]
5:     para todo phiin[1, ··· , cFk(True)] hacer
6:         s.Psi.append(cFk(True))
7:     fin para
8: fin para
```

La función cFk convierte al archivo f_{k} una lista circular, de forma que cada vez que es invocada devolverá únicamente un byte, como se ve en el *algoritmo 3.4*. Este algoritmo utiliza un solo parámetro que sirve para indicar sí se excluye o no los valores 0 y 255, esto es particularmente útil en el *algoritmo 3.3*, donde se requiere que el conjunto de reglas de cada célula del generador de flujo sea mayor a 0.

Algoritmo 3.4 cF_k - Lee la siguiente posición del archivo llave circular

Entrada: $nonZero = False$ "parámetro por default"

Salida: $Byte$

```
1: file ← open(var.f_k.name, rb)
2: read ← 0
3: mientras True hacer
4:     var.f_k.pos ← var.f_k.pos mod var.f_k.size
5:     file.seek(var.f_k.pos)
6:     read ← file.read(1)
7:     var.f_k.pos ← var.f_k.pos + 1
8:     si nonZero entonces
9:         si 0 < read < 255 entonces
10:            break
11:        fin si
12:    si no
13:        break
14:    fin si
15: fin mientras
16: file.close()
17: devolver read
```

Cada permutación es realizada ejecutando las instrucciones especificadas para cada cubo en el s_{k}, éstas son construidas como un arreglo de duplas en el que cada dupla contiene el token de la permutación que se va a utilizar y un listado de parámetros, como se ve en el *algoritmo 3.5*.

Nótese que el *algoritmo 3.5* comienza leyendo la configuración, como ya se indico en la nota del algoritmo, la implementación se dejo libre para permitir la personalización.

Algoritmo 3.5 mkInstructions - Construye las instrucciones del cifrado

Entrada: m
Salida: $instructions$

```
1: config ← loadConfig() {Esta implementación se deja abierta, algunos formatos sugeridos que
   podrían ser usados son los archivos INI o XML, pero también en algún módulo del código fuente}

2: para todo ev ∈ [0, ··· , config.gfParam.initEvolutions − 1] hacer
3:     acGenEval(gf.parameters)
4: fin para
5: instructions = {[ ]∀iCube ∈ [1, var.m.cubes]}
6: para todo iCube ∈ [0, ··· , var.m.cubes − 1] hacer
7:     totalInstructions ← (getParameter() << 24 ∨ getParameter() << 16 ∨
       getParameter() << 8 ∨ getParameter()) mod config.maxIterarions + 1
8:     para todo localInstructions ∈ [1, ··· , totalInstructions] hacer
9:         token ← fToken(cFk())
10:        param ← [ ]
11:        si token ∈ config.tokens3Parameters entonces
12:            param.append(getParameter())
13:            param.append(getParameter())
14:            param.append(getParameter())
15:        si no
16:            param.append(getParameter())
17:        fin si
18:        instructions[iCube].append((token, param))
19:    fin para
20: fin para
21: devolver instructions
```

La obtención de parámetros está definida por el *algoritmo 3.6*. Lo que hace es leer bloques de estados del GFBPP, llevando un contador para saber cuándo se ha alcanzado el último bloque, al llegar a esta situación provoca una nueva evolución y reinicia el contador.

Algoritmo 3.6 getParameter - Obtiene un byte del generador de flujo

Salida: *byte*

```
1: si (var.gfParam.pos + 1) ∗ 8 >= gf.parameters.size entonces
2:     var.gfParam.pos ← var.gfParam.pos mod gf.parameters.size
3:     acGenEval(gf.parameters)
4: fin si
5: read ← (gf.parameters.getDelta() >> var.gfParam.pos) ∧ 0xFF
6: var.gfParam.pos ← var.gfParam.pos + 1
7: devolver read
```

Para determinar el token correspondiente a un parámetro el *algoritmo 3.7* buscará en el listado de tokens usando el residuo de la división del parámetro con la longitud del listado de tokens.

Algoritmo 3.7 fToken - Encuentra instrucción en tabla de Tokens

Entrada: *instruction*
Salida: *token*
1: $token \leftarrow instruction \bmod len(config.tokens)$
2: **devolver** $config.tokens[token]$

El siguiente paso es doble ya que se va a construir la máscara XOR para cada cubo y se va a asignar una nueva posición.

Para dificultar el criptoanálisis, evitar una mala elección de llave k, del archivo llave f_{k}, de alguno de los valores de la configuración o que simplemente las permutaciones se anulen total o parcialmente entre sí, se determinó auxiliarse de cifrado de Vernam[16] gracias al GFBPP.

Lo primero es crear una *sal* para evitar que se usen tablas pre-calculadas de hash criptográficos, ésta consiste en una cadena de bytes cuyo valor inicial es el archivo llave, para que posteriormente sea concatenada una vez por cada cubo con la k y el resultado de $getParameter()$.

Lo segundo es determinar la nueva posición del cubo al final del proceso, para así poder reconstruir el mensaje, ésta posición se determina construyendo un listado de los cubos existentes y una cola con el nuevo orden. Para calcular el nuevo orden se obtiene el residuo de la división de $getParameter()$ con la longitud de la lista, ésta posición es extraída e introducida en la cola. Al final la lista quedara vacía y la cola se convertirá en una lista con las nuevas posiciones de los cubos.

Ambas operaciones se hacen de forma intercalada, ya que se consideró que era la forma más segura de hacerlo sí el GFBPP degradaba su aleatoriedad rápidamente, ya que no siempre se podrá controlar la calidad de los flujos del GFBPP. Estos pasos se verán en los *algoritmos 3.8 y 3.9*.

Hasta aquí el procedimiento es básicamente el mismo para el cifrado y el descifrado, pero como puede observarse en la *figura 3.2* los últimos pasos no son más que la operación opuesta. En el cifrado se sigue con el enmascarado de los cubos utilizando la mascará XOR, seguido de la interpretación de s_{k} mientras se va construyendo el mensaje cifrado, en el caso del descifrado primero es interpretado s_{k}, después se desenmascarán los cubos y finalmente se construye el mensaje en claro descartando los bytes del redondeo de cubos.

En el caso de la función $interpret()$, ésta es sólo una búsqueda de la permutación correspondiente al token indicado por s_{k}, entregándole el listado de parámetros correspondiente, que es cantidad máxima soportada por la permutación, una vez hecho esto hay tres tipos posibles de permutaciones:

1. Permutación simple. Es decir que se pueda aplicar a todo un cubo, éste tipo de permutación se presta para ampliar con facilidad este sistema de cifrado, agregando otros sistemas en la lista de tokens, definiendo su comportamiento en la función [$interpret()$] y modificando $mkInstructions$ para que estos tokens reciban su parámetro de cifrado con la longitud máxima soportada concatenando las salidas de $getParameter()$. El

único requisito para éste y los demás tipos, es que la permutación obtenida al cifrar sea del mismo tamaño que los datos en claro.

2. Permutación de segmentos con un parámetro. Toda la permutación de segmento primero determina los segmentos sobre los que va a operar, en este caso al ser los cubos de 8 bits por lado facilita usar como parámetro un byte, ya que serán los segmentos operados aquellos en el que el bit este activo.

3. Permutación de segmentos con tres parámetros. Al igual que en el caso anterior un parámetro indica los segmentos que se van a operar, en este caso el segundo parámetro, ya que el primero es el que se va a aplicar al token indicado, éstas permutaciones utilizan un tercer parámetro, ya que este determina sí el segmento es vertical u horizontal.

en todos los casos *interpret*() recibirá el *token*, *cubo* y *param*, devolviendo al final el cubo permutado. Nótese que para algunas permutaciones el parámetro inverso de una permutación no es él mismo, así que deberá determinar el inverso sí se está descifrando, en el caso de las permutaciones que se verán en la *sección 3.5*, se pueden calcular fácilmente sus inversos con el residuo de la división del negativo de *param* con el tamaño del grupo, como se ve en el *algoritmo 3.10 parte 1, 2 y 3*.

De esta manera los algoritmos de cifrado y descifrado, quedan formalmente definidos en los *algoritmos 3.8 y 3.9*.

Algoritmo 3.8 Cifrado

Entrada: m, f_k, k
Salida: m_t

1: $var.m = m$
2: $var.fk \leftarrow fk$
3: $prevCube \leftarrow 0$
4: {Construir cubos con mensaje en claro}
5: **si** $config.md.active$ **entonces**
6: $var.m.cubes \leftarrow ceil(var.m.size/64)$
7: $file \leftarrow open(var.md.name, w)$
8: $file.writeline(var.m.size)$
9: $file.close()$
10: **si no**
11: $var.m.cubes \leftarrow ceil((var.m.size + 8)/64)$
12: $var.m.data \leftarrow var.m.data \vee var.m.size << (var.m.cubes * 512 - 64)$
13: **fin si**
14: $initGF(256, sha256(k))$ {Iniciar Generador de Flujo}
15: $sk \leftarrow mkInstructions(var.m.data)$ {Construir conjunto de instrucciones por cubo}
16: $mask \leftarrow 2^{512} - 1$
17: $maskXOR \leftarrow 0$
18: $lstCubes \leftarrow [0, \cdots, var.m.cubes - 1]$
19: $lstNewOrder \leftarrow [\,]$
20: $salt \leftarrow read(fk)$
21: **para todo** $iCube \in [0, \cdots, var.m.cubes - 1]$ **hacer**
22: $salt \leftarrow salt + k + getParameter$
23: $maskXOR \leftarrow sha512(salt) << iCube * 512$ {Construye máscara XOR para el cubo}
24: $lstNewOrder.append(lstCubes.pop(getParameter() \bmod size(lstCubes)))$ {Asigna nueva posicion}
25: **fin para**
26: $var.m.data \leftarrow var.m.data \oplus maskXOR$ {Enmascara cubos}
27: $mt \leftarrow 0$
28: **para todo** $iCube \in [0, \cdots, var.m.cubes - 1]$ **hacer**
29: $cube \leftarrow (var.m.data >> iCube * 512) \vee mask$
30: {XOR con cubo previo}
31: **si** $iCube > 0$ **entonces**
32: $cube \leftarrow cube \oplus prevCube$
33: **fin si**
34: {Interpreta el conjunto de instrucciones que le corresponde}
35: **para todo** $item \in [0, \cdots, size(sk[iCube][item]) - 1]$ **hacer**
36: $(token, param) \leftarrow sk[iCube][item]$
37: $cube \leftarrow interpret(token, cube, param)$
38: **fin para**
39: $prevCube \leftarrow cube$
40: $mt \leftarrow mt \vee cube << iCube * 512$ {Construcción del mensaje cifrado}
41: **fin para**
42: **devolver** mt

Algoritmo 3.9 Descifrado

Entrada: m, k, f_k
Salida: m

1: $var.m \leftarrow m$
2: $var.fk \leftarrow fk$
3: $var.m.cubes \leftarrow ceil(var.m.size/64)$ {Construir cubos con mensaje cifrado}
4: $gf \leftarrow initGF(256, sha256(k))$ {Iniciar Generador de Flujo}
5: $instructions \leftarrow mkInstructions(var.m.data)$ {Construir conjunto de instrucciones por cubo}
6: $lstCubes \leftarrow [0, \cdots, var.m.cubes - 1]$
7: $lstNewOrder \leftarrow [\,]$
8: $mask \leftarrow 2^{512} - 1$
9: $maskXOR \leftarrow 0$
10: $salt \leftarrow read(fk)$
11: **para todo** $iCube \in [0, \cdots, var.m.cubes - 1]$ **hacer**
12: $salt \leftarrow salt + k + getParameter$
13: $maskXOR \leftarrow sha512(salt) << iCube * 512$ {Construye máscara XOR para el cubo}
14: $lstNewOrder.append(lstCubes.pop(getParameter() \bmod size(lstCubes)))$ {Asigna nueva posicion}
15: **fin para**
16: $mt \leftarrow 0$
17: **para todo** $iCube \in [var.m.cubes - 1, \cdots, 0]$ **hacer**
18: $cube \leftarrow (var.m.data >> iCube * 512) \vee mask$
19: {Interpreta el conjunto de instrucciones que le corresponde}
20: **para todo** $item \in [size(instructions[iCube][item]) - 1, \cdots, 0]$ **hacer**
21: $(token, param) \leftarrow instructions[iCube][item]$
22: $cube \leftarrow interpret(token, cube, param)$
23: **fin para**
24: {XOR con cubo previo}
25: **si** $iCube > 0$ **entonces**
26: $cube \leftarrow cube \oplus (var.m.data >> (iCube - 1) * 512) \vee mask$
27: **fin si**
28: $mt \leftarrow mt \vee cube << iCube * 512$
29: **fin para**
30: $mt \leftarrow mt \oplus maskXOR$ {Enmascara cubos}
31: {Construye mensaje en claro}
32: **si** $config.md.active$ **entonces**
33: $file \leftarrow open(var.md.name, r)$
34: $var.m.size \leftarrow file.readline()$
35: $file.close()$
36: **si no**
37: $prevSize \leftarrow var.m.size$
38: $var.m.size \leftarrow mt >> (var.m.cubes * 512 - 64)$
39: **si** $not(prevSize - 512 < var.m.size < prevSize + 512)$ **entonces**
40: **imprimir** Error fatal al descifrar, los parámetros, archivo llave o llave no pueden generar el archivo solicitado. Verifíquelos.
41: $exit(1)$
42: **fin si**
43: **fin si**
44: **devolver** $mt \wedge 2^{(var.m.size*8)} - 1$

Algoritmo 3.10 interpret - Interprete del cifrado (parte 1)

Entrada: *token*, *cube*, *param*
Salida: *cube*

si $token =$ "$dseta''$ **entonces**
    devolver $dseta(cube, param[0])$
si no, **si** $token =$ "mu'' **entonces**
    devolver $mu(cube, param[0])$
si no, **si** $token =$ "eta'' **entonces**
    si $parameters.decrypt$ **entonces**
        $param[0] \leftarrow -param[0] \bmod 4$
    fin si
    devolver $eta(cube, param[0])$
si no, **si** $token =$ "ni'' **entonces**
    si $parameters.decrypt$ **entonces**
        $param[0] \leftarrow -param[0] \bmod 4$
    fin si
    devolver $ni(cube, param[0])$
si no, **si** $token =$ "fi'' **entonces**
    si $parameters.decrypt$ **entonces**
        $rounds \leftarrow size(activeBits(param[0])) - 1$
        para todo $rnd \in [1, \cdots, rounds]$ **hacer**
            $cube \leftarrow fi(cube, param[0])$
        fin para
        devolver $cube$
    si no
        devolver $fi(cube, param[0])$
    fin si
si no, **si** $token =$ "$lambda''$ **entonces**
    si $parameters.decrypt$ **entonces**
        $rounds \leftarrow size(activeBits(param[0])) - 1$
        para todo $rnd \in [1, \cdots, rounds]$ **hacer**
            $cube \leftarrow lambda(cube, param[0])$
        fin para
        devolver $cube$
    si no
        devolver $fi(cube, param[0])$
    fin si
si no, **si** $token =$ "$iota''$ **entonces**
    si $parameters.decrypt$ **entonces**
        $param[0] \leftarrow -param[0] \bmod 64$
    fin si
    para todo $bit \in activeBits(param[1])$ **hacer**
        $segment \leftarrow chi(cube, 2^{bit}, param[2])$
        $segment \leftarrow iota(segment, param[0])$
        $cube \leftarrow Chi(cube, segment, 2^{bit}, param[2])$
    fin para
    devolver $cube$
fin si

Algoritmo 3.11 interpret - Interprete del cifrado (parte 2)

Entrada: *token*, *cube*, *param*
Salida: *cube*

1: **si** $token =$ "$ipsilon''$ **entonces**
2: **si** $parameters.decrypt$ **entonces**
3: $param[0] \leftarrow -param[0] \bmod 64$
4: **fin si**
5: **para todo** $bit \in activeBits(param[1])$ **hacer**
6: $segment \leftarrow chi(cube, 2^{bit}, param[2])$
7: $segment \leftarrow ipsilon(segment, param[0])$
8: $cube \leftarrow Chi(cube, segment, 2^{bit}, param[2])$
9: **fin para**
10: **devolver** *cube*
11: **si no**, **si** $token =$ "$omega''$ **entonces**
12: **si** $parameters.decrypt$ **entonces**
13: $param[0] \leftarrow -param[0] \bmod 64$
14: **fin si**
15: **para todo** $bit \in activeBits(param[1])$ **hacer**
16: $segment \leftarrow chi(cube, 2^{bit}, param[2])$
17: $segment \leftarrow omega(segment, param[0])$
18: $cube \leftarrow Chi(cube, segment, 2^{bit}, param[2])$
19: **fin para**
20: **devolver** *cube*
21: **si no**, **si** $token =$ "rho'' **entonces**
22: **si** $parameters.decrypt$ **entonces**
23: $param[0] \leftarrow -param[0] \bmod 64$
24: **fin si**
25: **para todo** $bit \in activeBits(param[1])$ **hacer**
26: $segment \leftarrow chi(cube, 2^{bit}, param[2])$
27: $segment \leftarrow rho(segment, param[0])$
28: $cube \leftarrow Chi(cube, segment, 2^{bit}, param[2])$
29: **fin para**
30: **devolver** *cube*
31: **si no**, **si** $token =$ "$sigma''$ **entonces**
32: **si** $parameters.decrypt$ **entonces**
33: $param[0] \leftarrow -param[0] \bmod 64$
34: **fin si**
35: **para todo** $bit \in activeBits(param[1])$ **hacer**
36: $segment \leftarrow chi(cube, 2^{bit}, param[2])$
37: $segment \leftarrow sigma(segment, param[0])$
38: $cube \leftarrow Chi(cube, segment, 2^{bit}, param[2])$
39: **fin para**
40: **devolver** *cube*
41: **fin si**

Algoritmo 3.12 interpret - Interprete del cifrado (parte 3)

1: **si** $token =$ "xi" **entonces**
2: **si** $parameters.decrypt$ **entonces**
3: $param[0] \leftarrow -param[0] \bmod 64$
4: **fin si**
5: **para todo** $bit \in activeBits(param[1])$ **hacer**
6: $segment \leftarrow chi(cube, 2^{bit}, param[2])$
7: $segment \leftarrow xi(segment, param[0])$
8: $cube \leftarrow Chi(cube, segment, 2^{bit}, param[2])$
9: **fin para**
10: **devolver** $cube$
11: **fin si**

3.5. Permutaciones

Los algoritmos de permutación aquí presentados cumplen con uno o varias de las características que dificultan el procesamiento en paralelo[1]:

- Generar puntos de sincronización
- Permutaciones cuya ganancia al procesar en paralelo sea nula o negativa
- Permitir el procesamiento en paralelo parcial o total de pequeñas tareas e inciertamente separadas. Irónicamente esto podría propiciar un desperdicio de recursos, en particular porque dificulta la planeación o la vuelve más costosa que la ganancia
- Generar cantidades variables de tareas que puedan calcularse en paralelo, ya que propicia ocasionalmente la existencia una o más Unidades de Procesamiento UP's ociosas dependiendo de las UP's existentes

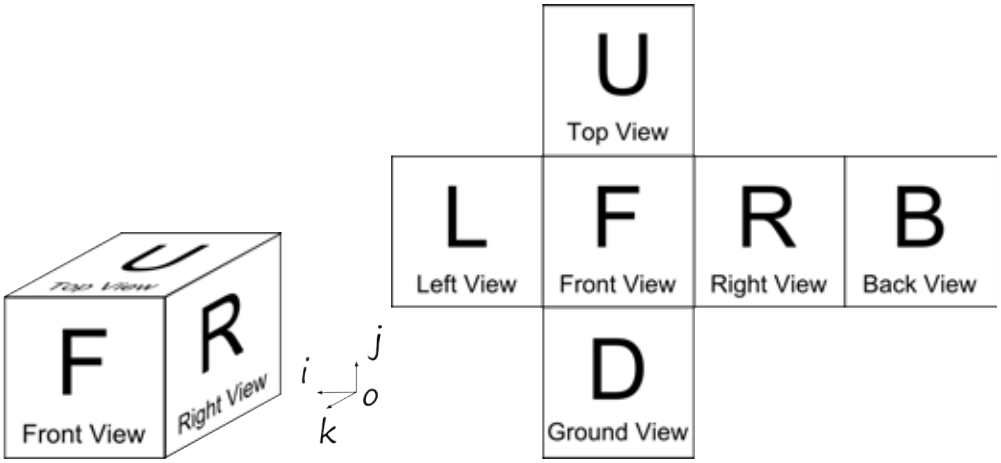
Estos algoritmos permutan los datos en bloques de 512 bits, cada bloque forma cubos de 8 x 8 x 8 bits. Después de un detallado análisis se decidió que:

1. Los símbolos para las caras son
   - **F**. Cara frontal sus variaciones según su posición son $\{F, \text{ꟻ}, \text{ꟻ}, \text{ꟻ}\}$
   - **B**. Cara posterior sus variaciones según su posición son $\{B, \text{ᗺ}, \text{ᗷ}, \text{ᗸ}\}$
   - **R**. Cara derecha sus variaciones según su posición son $\{R, \text{ꓤ}, \text{ꓬ}, \text{ꓣ}\}$
   - **L**. Cara izquierda sus variaciones según su posición son $\{L, \text{⅃}, \text{⅂}, \text{⅁}\}$
   - **U**. Cara superior sus variaciones según su posición son $\{U, \text{ᑌ}, \text{∩}, \text{ᑎ}\}$
   - **D**. Cara inferior sus variaciones según su posición son $\{D, \text{ᗡ}, \text{ᑯ}, \text{ᗤ}\}$
2. Las posiciones de cada bit serán definidas por lo alto del bit, es decir, el bit mas significativo "*511*" ocupa la posición $(7, 7, 7)$ y el menos significativo "*0*" la $(0, 0, 0)$ (*algoritmos 3.13, 3.14 y 3.15*)
3. La abscisa o eje x ira en dirección opuesta
4. La cara posterior estará definida en el plano x, y con $z = 0$ y la frontal con $z = 7$
5. La asignación de las caras "*izquierda, derecha, superior e inferior*" se harán basados en la variante americana del sistemas de vistas para dibujo técnico, como se ve en la *figura 3.3*
6. Finalmente como consecuencia de los puntos anteriores, el origen estará definido en la intersección de los planos formados por las caras R, B y D

Para facilitar la interpretación de estos parámetros se incluyeron plantillas recortables del cubo con las posiciones y coordenadas en el *apéndice A*, con las *figuras A.1* y *A.2*.

Debido a la naturaleza de las permutaciones usadas, se representarán los bloques, con diferentes notaciones. La idea es que las definiciones sean lo más claras posibles.

[1]Cabe aclarar que no se especifica en cuales permutaciones se aplica cada propósito, ya que en buena parte dependerá de la estrategia usada en la implementación del atacante.



(a) Representación del cubo 3D (b) Vistas en Sistema Americano

Figura 3.3: Representación del cubo

- α – Alpha. Representación de cubo como un matriz de datos (8 x 8 x 8), o "**notación en bits**".
  - P se usará para definir una coordenada en esta notación.
  - i, j y k corresponden a los ejes de coordenadas x, y y z.
- β – Beta. Representación de segmento de cubo como un arreglo de datos de longitud 8, o "**notación en Bytes**".
  - Q se usará para definir una coordenada en esta notación.
  - i y j corresponden a los ejes de coordenadas x y y.

Los algoritmos de permutación serán presentados con su nombre, una pequeña definición, una figura que represente su comportamiento (sí es que se necesita) y su algoritmo.

Antes de pasar a los algoritmos de permutación propuestos, se mostrará el funcionamiento de algunos algoritmos auxiliares, esto con el fin de poder facilitar el la lectura de los algoritmos de permutación.

Una función de utilidad en los algoritmos de permutación es *pos2coord* definida en el *algoritmo 3.13*, ya que permitirá calcular la coordenada de un bit en particular dentro del cubo.

Algoritmo 3.13 pos2coord - Calcula las coordenadas de un bit

Entrada: *pos*
Salida: Coordenadas i, j y k
1: $i \leftarrow pos \ mod \ 8$
2: $j \leftarrow ((pos - i) \ mod \ 64) \ // \ 8$
3: $k \leftarrow (pos - i) \ // \ 64$
4: **devolver** i, j, k

Otra función importante es *coord2pos* definida en los *algoritmos 3.13 y 3.15*, éstas funciones determinarán en base a una coordenada la posición dentro un cubo o un segmento para un determinado bit.

Algoritmo 3.14 coord2pos - Calcula la posición de un bit en base a las coordenadas tridimensionales

Entrada: i, j, k
Salida: *pos* posición
1: **devolver** $i + 8 * j + 64 * k$

Algoritmo 3.15 coord2pos - Calcula la posición de un bit en base a las coordenadas bidimensionales (Sobrecarga de función)

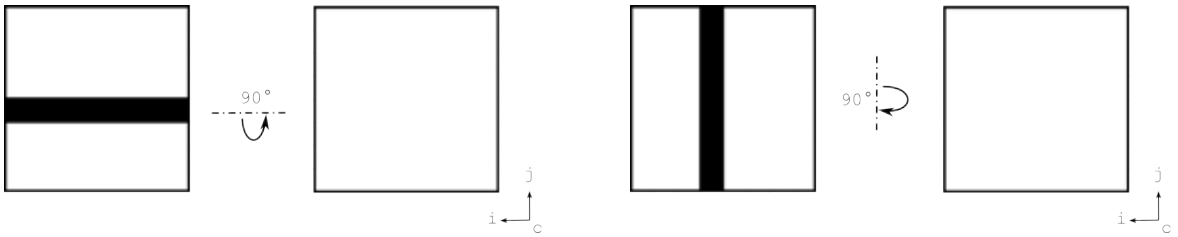
Entrada: i, j
Salida: *pos* posición
1: **devolver** $i + 8 * j$

La función *bitChanges* (*algoritmo 3.16*) recibe un byte que es convertido en bits, los bits activos son desplazados una posición en un corrimiento circular entre los bits activos, ésta función es útil en el intercambio de columnas y filas de los *algoritmos λ (3.23) y φ (3.24)*

Algoritmo 3.16 bitChanges - Obtiene un arreglo con las nuevas posiciones

Entrada: *byte*
Salida: *changes*
1: $alternate \leftarrow [\ ]$
2: $changes \leftarrow [\ ,\ ,\ ,\ ,\ ,\ ,\ ,\ ]$ {Tamaño 8}
3: **para todo** $bit \in [0, \ldots, 7]$ **hacer**
4: **si** $byte >> bit \wedge 1 = 1$ **entonces**
5: $alternate.append(bit)$
6: **si no**
7: $changes[bit] \leftarrow bit$
8: **fin si**
9: **fin para**
10: **para todo** $bit \in [0, \ldots size(alternate)]$ **hacer**
11: **si** $bit + 1 = size(alternate)$ **entonces**
12: $changes[alternate[bit]] \leftarrow alternate[0]$
13: **si no**
14: $changes[alternate[bit]] \leftarrow alternate[bit + 1]$
15: **fin si**
16: **fin para**
17: **devolver** *changes*

Finalmente se agregan los *algoritmos* χ (*algoritmo 3.17*) que permite extraer un segmento del cubo como se muestra en la *figura 3.4* y su inverso X (*algoritmo 3.18*) que lo reincorpora, como se vio en *algoritmo 3.10 parte 1, 2 y 3* de la función *interpret*() éstos son sumamente útiles cuando se encuentran funciones de permutación de segmento.



(a) Segmento Horizontal (b) Segmento vertical

Figura 3.4: Extracción de segmento del cubo

Algoritmo 3.17 chi - Extracción de segmento del cubo

Entrada: *cube*, *cutting* "sección requerida", *tilt* "par = Horizontal y non = Vertical"
Salida: *segment*

1: $tilt \leftarrow tilt \ mod \ 8$
2: $segment \leftarrow 0$
3: $glide \leftarrow 0$
4: $pos \leftarrow 0$
5: **para todo** $k \in [0, \ldots, 7]$ **hacer**
6: **para todo** $x \in [0, \ldots, 7]$ **hacer**
7: **si** $tilt = 0$ **entonces**
8: $pos \leftarrow coord2pos(x, cutting, k)$
9: **si no**
10: $pos \leftarrow coord2pos(cutting, j, k)$
11: **fin si**
12: $segment \leftarrow segment \vee (((cube >> pos) \wedge 0b1) << glide)$
13: $glide \leftarrow glide + 1$
14: **fin para**
15: **fin para**
16: **devolver** *segment*

Algoritmo 3.18 Chi - Integración de segmento con cubo

Entrada: *cube*, *segment*, *cutting* "sección requerida", *tilt* "par = Horizontal y non = Vertical"
Salida: *segment*

```
1: tilt ← tilt mod 8
2: glide ← 0
3: pos ← 0
4: newCube = 0
5: si tilt = 0 entonces
6:     para todo k ∈ [0, ..., 7] hacer
7:         para todo j ∈ [0, ..., 7] hacer
8:             si j <> cutting entonces
9:                 para todo i ∈ [0, ..., 7] hacer
10:                    pos ← coord2pos(i, cutting, k)
11:                    newCube ← newCube ∨ (((segment >> glide) ∧ 0b1) << pos)
12:                    glide ← glide + 1
13:                fin para
14:            si no
15:                para todo i ∈ [0, ..., 7] hacer
16:                    pos ← coord2pos(i, j, k)
17:                    newCube ← newCube ∨ (((cube >> pos) ∧ 0b1) << pos)
18:                fin para
19:            fin si
20:        fin para
21:    fin para
22: si no
23:    para todo k ∈ [0, ..., 7] hacer
24:        para todo j ∈ [0, ..., 7] hacer
25:            para todo i ∈ [0, ..., 7] hacer
26:                si i <> cutting entonces
27:                    pos ← coord2pos(cutting, j, k)
28:                    newCube ← newCube ∨ (((segment >> glide) ∧ 0b1) << pos)
29:                    glide ← glide + 1
30:                si no
31:                    pos ← coord2pos(i, j, k)
32:                    newCube ← newCube ∨ (((cube >> pos) ∧ 0b1) << pos)
33:                fin si
34:            fin para
35:        fin para
36:    fin para
37: fin si
38: devolver newCube
```

3.5.1. ν - Ni: Permutación girar cubo verticalmente

Ésta es una permutación de cubo, el objetivo es hacer girar las caras del cubo con la magnitud indicada en forma vertical. El cambio de coordenadas para cada elemento es regido por las ecuaciones (3.9), (3.10) y (3.11).

DEFINICIÓN **3.1** *Sí se representa la matriz de datos como un cubo, se tendrá el grupo de permutaciones:*

$$\nu = (F\ U\ B\ D)\ (F\ L\ B\ R)\ (F\ D\ B\ U)\ (F\ R\ B\ L)\ (U\ L\ D\ R)\ (U\ R\ D\ L) \quad (3.6)$$

pero debido a que al terminar los giros, las caras son renombradas de acuerdo a la figura 3.3, se puede simplificar (3.6) *así:*

$$\nu = (F\ \ U\ \ B\ \ D) \quad (3.7)$$

DEFINICIÓN **3.2** *Usando la notación en bits, se define el grupo de permutaciones ν:*

$$\nu = (1\ \ P'_{\nu}\ \ P'_{2\nu}\ \ P'_{3\nu}) \quad (3.8)$$

y los cambios en las coordenadas de los puntos como:

$$P'_{\nu} = i, k, j^{-1} \quad (3.9)$$
$$P'_{2\nu} = i, j^{-1}, k^{-1} \quad (3.10)$$
$$P'_{3\nu} = i, k^{-1}, j \quad (3.11)$$

su algoritmo se define como:

Algoritmo 3.19 ni - Permutación girar cubo verticalmente

Entrada: *cube*, *turns*
Salida: *cube*

```
 1: turns ← turns mod 4
 2: si turns = 0 entonces
 3:     devolver cube
 4: si no
 5:     newCube ← 0
 6:     para todo k ∈ [0, ..., 7] hacer
 7:         para todo j ∈ [0, ..., 7] hacer
 8:             pos ← coord2pos(0, j, k)
 9:             glide ← 0
10:             si turns = 1 entonces
11:                 turns ← coord2pos(0, k, 7 − j)
12:             si no, si turns = 2 entonces
13:                 turns ← coord2pos(0, 7 − j, 7 − k)
14:             si no, si turns = 3 entonces
15:                 turns ← coord2pos(0, 7 − k, j)
16:             fin si
17:             newCube ← newCube ∨ (cube >> pos ∧ 0xFF) << turns
18:         fin para
19:     fin para
20:     devolver newCube
21: fin si
```

3.5.2. η - Eta: Permutación girar cubo horizontalmente

Ésta es una permutación de cubo, el objetivo es hacer girar las caras del cubo con la magnitud indicada en forma horizontal. El cambio de coordenadas para cada elemento es regido por las ecuaciones (3.15), (3.16) y (3.17).

DEFINICIÓN **3.3** *Sí se representa la matriz de datos como un cubo, se tendrá el grupo de permutaciones:*

$$\eta = (F\ \ R\ \ B\ \ L)\ (\digamma\ \cap\ \beta\ \ \Gamma)\ (\exists\ \ \bot\ \ \beta\ \ \Re)\ (\digamma\ \ \Gamma\ \ \beta\ \ \cap)\ (U\ \ \Re\ \ \Gamma\ \ \bot)\ (\cap\ \ \bot\ \ \Gamma\ \ \Re) \quad (3.12)$$

pero debido a que al terminar los giros, las caras son renombradas de acuerdo a la figura 3.3, se puede simplificar (3.12) *así:*

$$\eta = (F\ \ R\ \ B\ \ L) \quad (3.13)$$

DEFINICIÓN **3.4** *Usando la* ***notación en bits****, se define el grupo de permutaciones* η*:*

$$\eta = (1, P'_{\eta}, P'_{2\eta}, P'_{3\eta}) \quad (3.14)$$

y los cambios en las coordenadas de los puntos como:

$$P'_{\eta} = k^{-1}, j, i \quad (3.15)$$
$$P'_{2\eta} = i^{-1}, j, k^{-1} \quad (3.16)$$
$$P'_{3\eta} = k, j, i^{-1} \quad (3.17)$$

su algoritmo se define como:

Algoritmo 3.20 eta - Permutación girar cubo horizontalmente

Entrada: *cube*, *turns*
Salida: *cube*

```
 1: turns ← turns mod 4
 2: si turns = 0 entonces
 3:     devolver cube
 4: si no
 5:     newCube ← 0
 6:     para todo k ∈ [0, ..., 7] hacer
 7:         para todo j ∈ [0, ..., 7] hacer
 8:             para todo i ∈ [0, ..., 7] hacer
 9:                 pos ← coord2pos(i, j, k)
10:                 glide ← 0
11:                 si turns = 1 entonces
12:                     glide ← coord2pos(7 − k, j, i)
13:                 si no, si turns = 2 entonces
14:                     glide ← coord2pos(7 − i, j, 7 − k)
15:                 si no, si turns = 3 entonces
16:                     glide ← coord2pos(k, j, 7 − i)
17:                 fin si
18:                 newCube ← newCube ∨ (cube >> pos ∧ 0b1) << glide
19:             fin para
20:         fin para
21:     fin para
22:     devolver newCube
23: fin si
```

3.5.3. ζ – Dseta: Negación de columnas

Ésta es una permutación de cubo, donde el objetivo es negar las columnas indicadas, es decir invertir su valor actual.

DEFINICIÓN **3.5** *Sea la permutación ζ, definida por el algoritmo 3.21.*

Algoritmo 3.21 dseta - Negación de columnas

Entrada: *cube*, *byte*
Salida: *cube*

```
1: si byte = 0 entonces
2:     devolver cube
3: si no
4:     newCube ← 0
5:     nCube ← ¬cube
6:     {Las siguientes dos líneas son porque la representación hexadecimal de la máscara es muy
       larga (128 caracteres)}
7:     mask ← 0x01010101010101010101010101010101
8:     mask ← mask ∨ mask << 128 ∨ mask << 256 ∨ mask << 384
9:     para todo pos ∈ [0, . . . , 7] hacer
10:        si byte >> pos ∧ 1 = 1 entonces
11:            newCube ← newCube ∨ (nCube ∧ mask << pos)
12:        si no
13:            newCube ← newCube ∨ (cube ∧ mask << pos)
14:        fin si
15:    fin para
16:    devolver newCube
17: fin si
```

3.5.4. μ – Mu: Negación de filas

Ésta es una permutación de cubo, donde el objetivo es negar las filas indicadas, es decir invertir su valor actual.

DEFINICIÓN **3.6** *Sea la permutación μ, definida por el algoritmo 3.22.*

Algoritmo 3.22 dseta - Negación de filas

Entrada: *cube*, *byte*
Salida: *cube*

si byte $= 0$ **entonces**
    devolver *cube*
si no
    $newCube \leftarrow 0$
    $nCube \leftarrow \neg cube$
    {Las siguientes dos líneas son porque la representación hexadecimal de la máscara es muy larga (128 caracteres)}
    $mask \leftarrow 0x00000000000000FF00000000000000FF$
    $mask \leftarrow mask \vee mask << 128 \vee mask << 256 \vee mask << 384$
    para todo $pos \in [0, \ldots, 7]$ **hacer**
        si $byte >> pos \wedge 1 = 1$ **entonces**
            $newCube \leftarrow newCube \vee (nCube \wedge mask << pos * 8)$
        si no
            $newCube \leftarrow newCube \vee (cube \wedge mask << pos * 8)$
        fin si
    fin para
    devolver *newCube*
fin si

3.5.5. λ – Lambda: Permutación de columnas por corrimiento circular

Ésta es una permutación de cubo cuyo objetivo es hacer un corrimiento circular entre las columnas indicadas.

Definición **3.7** *Sea la permutación λ que se muestra en la figura 3.5,*

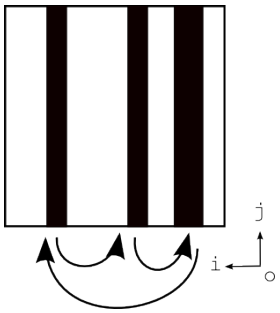


Figura 3.5: Permutación Lambda

definida por el algoritmo 3.23.

Algoritmo 3.23 lambda - Permutación de columnas por corrimiento circular

Entrada: *cube*, *byte*
Salida: *cube*

```
1: si byte = 0 entonces
2:     devolver cube
3: si no
4:     newCube ← 0
5:     glide ← 0
6:     {Las siguientes dos líneas son porque la representación hexadecimal de la máscara es muy
       larga (128 caracteres)}
7:     mask ← 0x01010101010101010101010101010101
8:     mask ← mask ∨ mask << 128 ∨ mask << 256 ∨ mask << 384
9:     changes ← bitChanges(byte)
10:    para todo bit ∈ [0, . . . , 7] hacer
11:        glide ← changes[pos] − pos
12:        si glide >= 0 entonces
13:            newCube ← newCube ∨ (cube ∧ mask << pos) << glide
14:        si no
15:            glide ← glide ∗ −1
16:            newCube ← newCube ∨ (cube ∧ mask << pos) >> glide
17:        fin si
18:    fin para
19:    devolver newCube
20: fin si
```

3.5.6. φ – Fi: Permutación de filas por corrimiento circular

Esta es una permutación de cubo cuyo objetivo es hacer un corrimiento circular entre las filas indicadas.

DEFINICIÓN **3.8** *Sea la permutación φ que se muestra en la figura 3.6,*

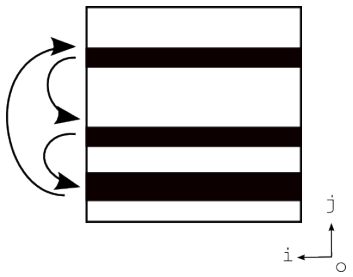


Figura 3.6: Permutación Fi

definida por el algoritmo 3.24.

Algoritmo 3.24 fi - Permutación de filas por corrimiento circular

Entrada: *cube*, *byte*
Salida: *cube*

si byte = 0 **entonces**
    devolver *cube*
si no
    $newCube \leftarrow 0$
    $glide \leftarrow 0$
    {Las siguientes dos líneas son porque la representación hexadecimal de la máscara es muy larga (128 caracteres)}
    $mask \leftarrow 0x00000000000000FF00000000000000FF$
    $mask \leftarrow mask \vee mask << 128 \vee mask << 256 \vee mask << 384$
    $changes \leftarrow bitChanges(byte)$
    para todo $bit \in [0, \ldots, 7]$ **hacer**
        $glide \leftarrow changes[pos] - pos$
        si $glide >= 0$ **entonces**
            $newCube \leftarrow newCube \vee (cube \wedge mask << pos * 8) << glide * 8$
        si no
            $glide \leftarrow glide * -1$
            $newCube \leftarrow newCube \vee (cube \wedge mask << pos * 8) >> glide * 8$
        fin si
    fin para
    devolver *newCube*
fin si

3.5.7. ι – Iota: Permutación corrimiento horizontal circular independiente

Ésta es una permutación de segmento cuyo objetivo es hacer un corrimiento circular de la magnitud indicada entre las filas.

Definición **3.9** *Sea la permutación ι que se muestra en la figura 3.7,*

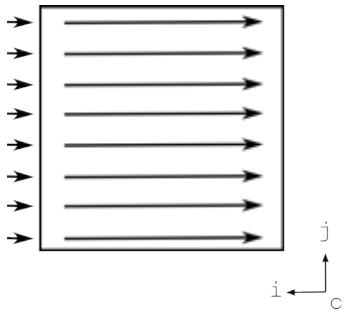


Figura 3.7: Permutación Iota

definida por el algoritmo 3.25.

Algoritmo 3.25 iota - Permutación corrimiento horizontal circular independiente

Entrada: *segment*, *glide*
Salida: *segment*
1: $glide \leftarrow glide \ mod \ 8$
2: **si** $glide = 0$ **entonces**
3: **devolver** *segment*
4: **si no**
5: $newSegment \leftarrow 0$
6: **para** $pos \in [0, \ldots, 7]$ **hacer**
7: $mask \leftarrow 0xFF << pos * 8$
8: $newSegment \leftarrow newSegment \vee (segment \wedge mask) >> glide \wedge mask$
9: $newSegment \leftarrow newSegment \vee (segment \wedge mask) << (8 - glide) \wedge mask$
10: **fin para**
11: **devolver** *newSegment*
12: **fin si**

3.5.8. ρ – Rho: Permutación corrimiento vertical circular independiente

Ésta es una permutación de segmento cuyo objetivo es hacer un corrimiento circular de la magnitud indicada entre las columnas.

DEFINICIÓN **3.10** *Sea la permutación ρ que se muestra en la figura 3.8,*

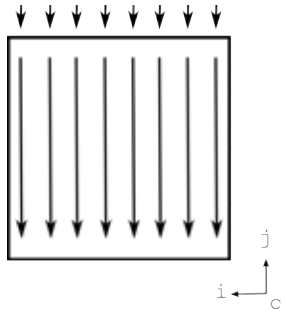


Figura 3.8: Permutación Rho

definida por el algoritmo 3.26.

Algoritmo 3.26 rho - Permutación corrimiento vertical circular independiente

Entrada: *segment*, *glide*
Salida: *segment*
1: $glide \leftarrow glide \ mod \ 8$
2: **si** $glide = 0$ **entonces**
3: **devolver** *segment*
4: **si no**
5: $glide \leftarrow glide * 8$
6: $newSegment \leftarrow segment >> glide \vee (segment \wedge (2^{glide} - 1)) << (64 - glide)$
7: **devolver** *newSegment*
8: **fin si**

3.5.9. ω – Omega: Permutación corrimiento horizontal circular zigzagueante

Ésta es una permutación de segmento cuyo objetivo es hacer un corrimiento circular de la magnitud indicada, en una trayectoria en zig-zag sobre las filas.

Definición **3.11** *Sea la permutación ω que se muestra en la figura 3.9,*

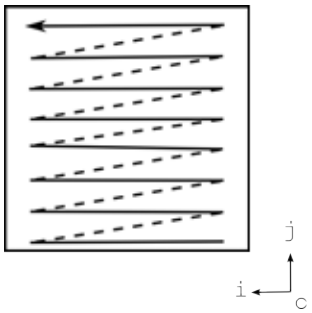


Figura 3.9: Permutación Omega

definida por el algoritmo 3.27.

Algoritmo 3.27 omega - Permutación corrimiento horizontal circular zigzagueante

Entrada: *segment*, *glide*
Salida: *segment*
1: $glide \leftarrow glide \ mod \ 64$
2: **si** $glide = 0$ **entonces**
3: **devolver** *segment*
4: **si no**
5: $newSegment \leftarrow segment >> glide \vee (segment \wedge (2^{glide} - 1)) << (64 - glide)$
6: **devolver** *newSegment*
7: **fin si**

3.5.10. ς – Sigma: Permutación corrimiento vertical circular zigzagueante

Ésta es una permutación de segmento cuyo objetivo es hacer un corrimiento circular de la magnitud indicada, en una trayectoria en zig-zag sobre las columnas.

DEFINICIÓN **3.12** *Sea la permutación ς que se muestra en la figura 3.10,*

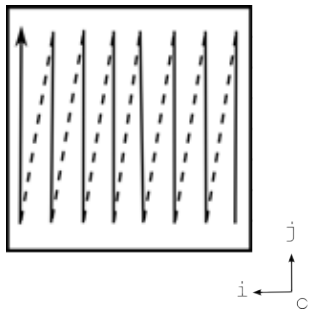


Figura 3.10: Permutación Sigma

definida por el algoritmo 3.28.

Algoritmo 3.28 sigma - Permutación corrimiento vertical circular zigzagueante

Entrada: *segment*, *glide*
Salida: *segment*
1: $glide \leftarrow glide \ mod \ 64$
2: **si** $glide = 0$ **entonces**
3: **devolver** *segment*
4: **si no**
5: $hGlide \leftarrow glide//8$
6: $vGlide \leftarrow glide \bmod 8$
7: $newSegment \leftarrow (segment << vGlide * 8) \wedge (2^{64} - 1)$
8: $tmp \leftarrow segment >> (8 - vGlide) * 8$
9: **para todo** $row \in [0, \ldots, vGlide]$ **hacer**
10: $aux \leftarrow (tmp >> row * 8) \wedge 0xFF$
11: $aux \leftarrow ((aux << 1) \vee (aux >> 7)) \wedge 0xFF$
12: $newSegment \leftarrow newSegment \vee aux << row * 8$
13: **fin para**
14: $segment \leftarrow newSegment$
15: $newSegment \leftarrow 0$
16: **para todo** $row \in [0, \ldots, 7]$ **hacer**
17: $aux \leftarrow (tmp >> row * 8) \wedge 0xFF$
18: $aux \leftarrow ((aux << hGlide) \vee (aux >> (8 - hGlide))) \wedge 0xFF$
19: $newSegment \leftarrow newSegment \vee aux << row * 8$
20: **fin para**
21: **devolver** *newSegment*
22: **fin si**

3.5.11. ξ – Xi: Permutación corrimiento horizontal circular reptante

Ésta es una permutación de segmento cuyo objetivo es hacer un corrimiento circular de la magnitud indicada, en una trayectoria similar al recorrido de una serpiente sobre las filas.

Definición **3.13** *Sea la permutación ξ que se muestra en la figura 3.11,*

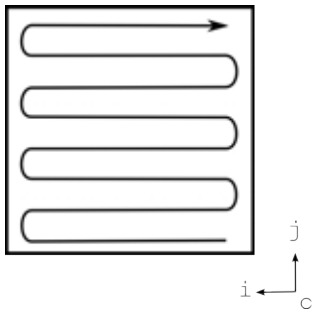


Figura 3.11: Permutación Xi

definida por el algoritmo 3.29.

Algoritmo 3.29 xi - Permutación corrimiento horizontal circular reptante

Entrada: *segment*, *glide*
Salida: *segment*
1: $glide \leftarrow glide \ mod \ 64$
2: **si** $glide = 0$ **entonces**
3: **devolver** *segment*
4: **si no**
5: **devolver** $auxXi(omega(auxXi(segment), glide))$
6: **fin si**

Algoritmo 3.30 auxXi - Ordena segmento en patrón xi

Entrada: *segment*
Salida: *segment*
1: $newSegment \leftarrow segment \wedge 0x00FF00FF00FF00FF$
2: **para todo** $glide \in [1, 3, 5, 7]$ **hacer**
3: $newSegment \leftarrow newSegment \vee (segment >> (glide * 8) \wedge 0xFF) << (7 - glide + 1) \ mod \ 8 * 8$
4: **fin para**
5: **devolver** *newSegment*

3.5.12. υ – ípsilon: Permutación corrimiento vertical circular reptante

Ésta es una permutación de segmento cuyo objetivo es hacer un corrimiento circular de la magnitud indicada, en una trayectoria similar al recorrido de una serpiente sobre las columnas.

Definición **3.14** *Sea la permutación υ que se muestra en la figura 3.12,*

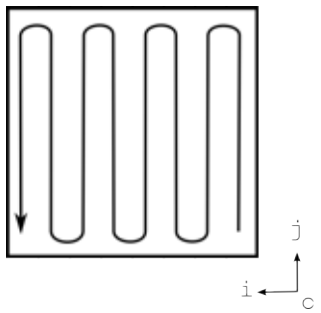


Figura 3.12: Permutación ípsilon

definida por el algoritmo 3.31.

Algoritmo 3.31 ipsilon - Permutación corrimiento vertical circular reptante

Entrada: *segment*, *glide*
Salida: *segment*
1: $glide \leftarrow glide \ mod \ 64$
2: **si** $glide = 0$ **entonces**
3: **devolver** *segment*
4: **si no**
5: **devolver** $auxIpsilon(sigma(auxIpsilon(segment), glide))$
6: **fin si**

Algoritmo 3.32 auxIpsilon - Algoritmo auxiliar de permutación corrimiento vertical circular reptante

Entrada: *segment*
Salida: *segment*
1: $newSegment \leftarrow segment \wedge 0x5555555555555555$
2: **para todo** $row \in [0, \ldots, 7]$ **hacer**
3: $newSegment \leftarrow newSegment \vee ((segment >> row * 8) \wedge 0xAA) << (7 - row) * 8$
4: **fin para**
5: **devolver** *newSegment*

Capítulo 4

Resultados

4.1. Análisis del GFBPP

La validación de la calidad de los flujos se medirá con ayuda de algunas de las pruebas de aleatoriedad propuestas por el NIST[10] y la entropía de la información de Shannon[12], para que finalmente se promedien todas estas evaluaciones obteniendo un único valor de aleatoriedad (lo que se consideró suficiente para este propósito, pero discutible, ya que se puede argumentar que promediar resultados conduce a la perdida de información y no refleja todo comportamiento fielmente), principalmente porque será la semilla quien defina los estados iniciales y todos los subsecuentes.

Como lo que se buscó es poder afirmar que "casi" cualquier conjunto de reglas podría generar un flujo pseudo-aleatorio, se determinó construir un AC con reglas dinámicas y pocas células, de tal manera que fácilmente se pueda perder aleatoriedad, ya sea por la repetición de ciclos o por el tamaño en si mismo y porque así es más fácil estimar como se comportará un AC con mayor cantidad de células y un flujo mucho más grande, lo cual puede ser inviable por lo costoso del cálculo con recursos de hardware limitados. Entonces, definiremos los GFBPP con los siguientes parámetros:

- $|S| = 16$
- $\Delta = 0b1 << 8 = 256$, *en los AC es común que el estado inicial sea una sola célula central activa*
- $|G| = 8 + \frac{1024 * 1024}{2} = 524,296$, ya que omitimos las primeras $\frac{|S|}{2} = 8$ generaciones, por tanto tendremos un flujo de 1 Mb
- $\forall \Psi(|\Psi| = [5, 16])$

La selección de cada conjunto de reglas para cada célula, se realizó individualmente con ayuda de la función random bajo los siguientes criterios:

1. Criterio de control concatenando las salidas de la función random.

2. Usar las reglas conocidas, pertenecientes a la *Clase III de Wolfram* $\{30, 75, 86, 89, 101, 135, 149\}$.

3. Usar las reglas cortas sugeridas por Seredynski(2003) [11] $\{30, 90, 105, 150, 165, 86, 101, 153, 39\}$.

4. Usar indiscriminadamente las reglas omitiendo únicamente la 0 y 255. La hipótesis es que aunque la lógica nos indique que las reglas de *Clase I de Wolfram*, van a sabotear la aleatoriedad del flujo resultante, el hecho de que en cada generación se varíe la regla utilizada y que para cuando se complete el ciclo del conjunto Ψ muy probablemente la vecindad habrán sido influenciada por otras reglas, le aportará más a la aleatoriedad de lo que le resta, gracias a la diversidad agregada y la propagación del caos.

Por cuestiones de tiempo y el alto procesamiento requerido sólo se analizaron los últimos 512 kb del flujo con las pruebas de aleatoriedad, lo que representa las ultimas $262,144$ generaciones, obteniendo los resultados de la tabla 4.1 y la *figura 4.1*.

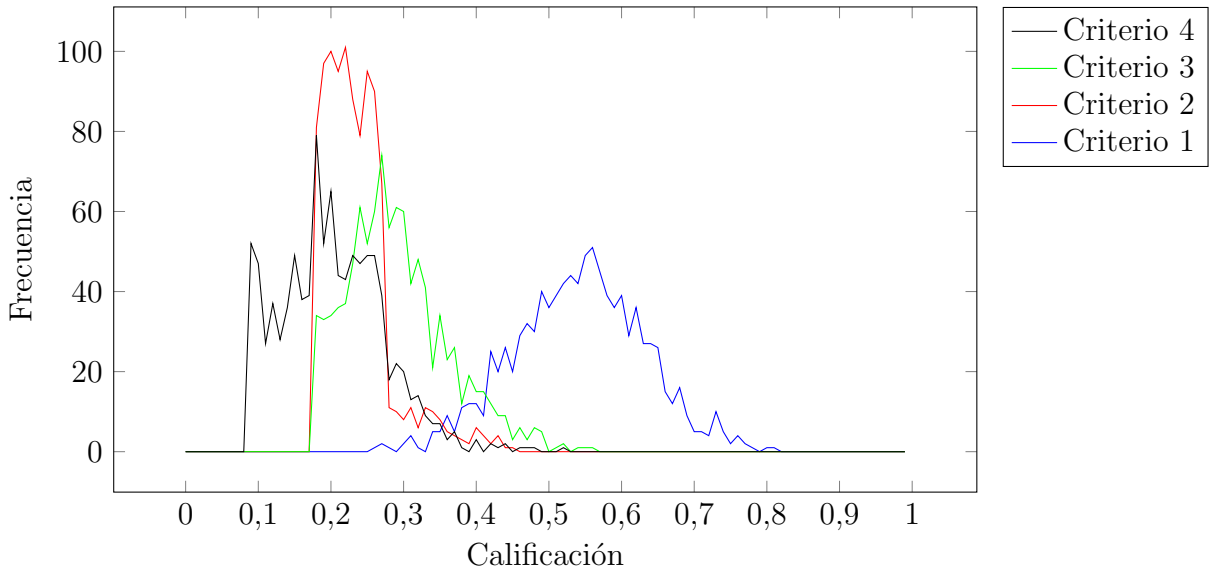


Figura 4.1: Distribución de resultados de pruebas de aleatoriedad

Criterio	**Alias**	**Promedio**	**Simulaciones**
1	Control (función random)	0.541	1000
2	Reglas conocidas	0.236	1000
3	Sugeridas[11]	0.290	1000
4	Indiscriminadamente	0.202	1000

Tabla 4.1: Comparación de resultados de pruebas de aleatoriedad promediados

En base a los datos de la tabla 4.1 y la tabla 4.2 se pueden sacar algunas conclusiones previas[1] sobre las reglas elegidas:

1. Con ningún criterio se alcanza un flujo pseudo-aleatorio óptimo
2. Los flujos generados con los criterios 2, 3, y 4 perdieron calidad rápidamente, como era de esperarse, por la limitada cantidad de células, lo cual nos permite ver como se comportan en general los criterios con más células en flujos muy grandes
3. Los criterios 2, 3 y 4 aún con sus matices y cualidades, generan flujos pseudo-aleatorios de baja calidad sí se comparan con el criterio de control. Por sí solos no son muy buenos como método de cifrado único, pero usados como un elemento secundario dentro de un cifrado mas elaborado, pueden ser muy útiles
4. El criterio 4 tiene un desempeño inferior al resto de propuestas
5. El resultado mas interesante es el de la entropía de la información de Shannon[12], ya que nos indica que a pesar de los malos resultados en general, el equilibrio de 1's y 0's se mantiene constante, lo que sugiere que el GFBPP aunque no tiene una alta aleatoriedad, parece comportarse como un sistema complejo

Test	**Criterio 1**	**Criterio 2**	**Criterio 3**	**Criterio 4**
shannonentropy	1.000	1.000	1.000	0.999
monobitfrequencytest	0.498	0.054	0.113	0.011
blockfrequencytest	0.496	1.000	1.000	0.574
runstest	0.516	0.007	0.029	0.017
longestrunones10000	0.488	0.001	0.073	0.039
binarymatrixranktest	0.516	0.438	0.426	0.419
spectraltest	0.484	0.000	0.000	0.002
nonoverlappingtemplatematchingtest	0.507	0.010	0.046	0.054
overlappingtemplatematchingtest	0.463	0.003	0.031	0.013
maurersuniversalstatistictest	0.486	0.006	0.326	0.077
cumultativesumstest	0.495	0.078	0.149	0.014

Tabla 4.2: Desglose por prueba aplicada de resultados promedio de pruebas de aleatoriedad

4.2. Vectores de prueba de permutaciones

En esta sección se mostrarán tablas que corresponden a los valores que deberán obtenerse al procesar los datos con los parámetros y la permutación indicada, de manera que facilite la implementación.

[1]Las conclusiones finales se abordarán en el capitulo correspondiente

Operación	**Vector**
$X \leftarrow$	0xAD99B1D3B1103401
$X \leftarrow \iota(X, 1)$	0xD6CCD8E9D8081A80
$X \leftarrow \iota(X, 1)$	0x6B666CF46C040D40
$X \leftarrow \iota(X, 1)$	0xB533367A36028620
$X \leftarrow \iota(X, 1)$	0xDA991B3D1B014310
$X \leftarrow \iota(X, 1)$	0x6DCC8D9E8D80A108
$X \leftarrow \iota(X, 1)$	0xB666C64FC640D004
$X \leftarrow \iota(X, 1)$	0x5B3363A763206802
$X \leftarrow \iota(X, 1)$	0xAD99B1D3B1103401
$X \leftarrow$	0x50F6E64A10465061
$X \leftarrow \iota(X, 1)$	0x287B7325082328B0
$X \leftarrow \iota(X, 1)$	0x14BDB99204911458
$X \leftarrow \iota(X, 1)$	0x0ADEDC4902C80A2C
$X \leftarrow \iota(X, 1)$	0x056F6EA401640516
$X \leftarrow \iota(X, 1)$	0x82B737528032820B
$X \leftarrow \iota(X, 1)$	0x41DB9B2940194185
$X \leftarrow \iota(X, 1)$	0xA0EDCD94208CA0C2
$X \leftarrow \iota(X, 1)$	0x50F6E64A10465061
$X \leftarrow$	0x725236026A30DBD5
$X \leftarrow \iota(X, 1)$	0x39291B013518EDEA
$X \leftarrow \iota(X, 1)$	0x9C948D809A0CF675
$X \leftarrow \iota(X, 1)$	0x4E4AC6404D067BBA
$X \leftarrow \iota(X, 1)$	0x27256320A603BD5D
$X \leftarrow \iota(X, 1)$	0x9392B1105381DEAE
$X \leftarrow \iota(X, 1)$	0xC949D808A9C06F57
$X \leftarrow \iota(X, 1)$	0xE4A46C04D460B7AB
$X \leftarrow \iota(X, 1)$	0x725236026A30DBD5
$X \leftarrow$	0xB7AE6CFC8558B435
$X \leftarrow \iota(X, 1)$	0xDB57367EC22C5A9A
$X \leftarrow \iota(X, 1)$	0xEDAB1B3F61162D4D
$X \leftarrow \iota(X, 1)$	0xF6D58D9FB00B96A6
$X \leftarrow \iota(X, 1)$	0x7BEAC6CF58854B53
$X \leftarrow \iota(X, 1)$	0xBD7563E72CC2A5A9
$X \leftarrow \iota(X, 1)$	0xDEBAB1F31661D2D4
$X \leftarrow \iota(X, 1)$	0x6F5DD8F90BB0696A
$X \leftarrow \iota(X, 1)$	0xB7AE6CFC8558B435

Tabla 4.3: Vectores de prueba para *iota*

Operación	Vector
$X \leftarrow$	0xAD99B1D3B1103401
$X \leftarrow \rho(X, 1)$	0x01AD99B1D3B11034
$X \leftarrow \rho(X, 1)$	0x3401AD99B1D3B110
$X \leftarrow \rho(X, 1)$	0x103401AD99B1D3B1
$X \leftarrow \rho(X, 1)$	0xB1103401AD99B1D3
$X \leftarrow \rho(X, 1)$	0xD3B1103401AD99B1
$X \leftarrow \rho(X, 1)$	0xB1D3B1103401AD99
$X \leftarrow \rho(X, 1)$	0x99B1D3B1103401AD
$X \leftarrow \rho(X, 1)$	0xAD99B1D3B1103401
$X \leftarrow$	0x50F6E64A10465061
$X \leftarrow \rho(X, 1)$	0x6150F6E64A104650
$X \leftarrow \rho(X, 1)$	0x506150F6E64A1046
$X \leftarrow \rho(X, 1)$	0x46506150F6E64A10
$X \leftarrow \rho(X, 1)$	0x1046506150F6E64A
$X \leftarrow \rho(X, 1)$	0x4A1046506150F6E6
$X \leftarrow \rho(X, 1)$	0xE64A1046506150F6
$X \leftarrow \rho(X, 1)$	0xF6E64A1046506150
$X \leftarrow \rho(X, 1)$	0x50F6E64A10465061
$X \leftarrow$	0x725236026A30DBD5
$X \leftarrow \rho(X, 1)$	0xD5725236026A30DB
$X \leftarrow \rho(X, 1)$	0xDBD5725236026A30
$X \leftarrow \rho(X, 1)$	0x30DBD5725236026A
$X \leftarrow \rho(X, 1)$	0x6A30DBD572523602
$X \leftarrow \rho(X, 1)$	0x026A30DBD5725236
$X \leftarrow \rho(X, 1)$	0x36026A30DBD57252
$X \leftarrow \rho(X, 1)$	0x5236026A30DBD572
$X \leftarrow \rho(X, 1)$	0x725236026A30DBD5
$X \leftarrow$	0xB7AE6CFC8558B435
$X \leftarrow \rho(X, 1)$	0x35B7AE6CFC8558B4
$X \leftarrow \rho(X, 1)$	0xB435B7AE6CFC8558
$X \leftarrow \rho(X, 1)$	0x58B435B7AE6CFC85
$X \leftarrow \rho(X, 1)$	0x8558B435B7AE6CFC
$X \leftarrow \rho(X, 1)$	0xFC8558B435B7AE6C
$X \leftarrow \rho(X, 1)$	0x6CFC8558B435B7AE
$X \leftarrow \rho(X, 1)$	0xAE6CFC8558B435B7
$X \leftarrow \rho(X, 1)$	0xB7AE6CFC8558B435

Tabla 4.4: Vectores de prueba para *rho*

Operación	**Vector**
$X \leftarrow$	0xAD99B1D3B1103401
$X \leftarrow \omega(X, 8)$	0x01AD99B1D3B11034
$X \leftarrow \omega(X, 8)$	0x3401AD99B1D3B110
$X \leftarrow \omega(X, 8)$	0x103401AD99B1D3B1
$X \leftarrow \omega(X, 8)$	0xB1103401AD99B1D3
$X \leftarrow \omega(X, 8)$	0xD3B1103401AD99B1
$X \leftarrow \omega(X, 8)$	0xB1D3B1103401AD99
$X \leftarrow \omega(X, 8)$	0x99B1D3B1103401AD
$X \leftarrow \omega(X, 8)$	0xAD99B1D3B1103401
$X \leftarrow$	0x50F6E64A10465061
$X \leftarrow \omega(X, 8)$	0x6150F6E64A104650
$X \leftarrow \omega(X, 8)$	0x506150F6E64A1046
$X \leftarrow \omega(X, 8)$	0x46506150F6E64A10
$X \leftarrow \omega(X, 8)$	0x1046506150F6E64A
$X \leftarrow \omega(X, 8)$	0x4A1046506150F6E6
$X \leftarrow \omega(X, 8)$	0xE64A1046506150F6
$X \leftarrow \omega(X, 8)$	0xF6E64A1046506150
$X \leftarrow \omega(X, 8)$	0x50F6E64A10465061
$X \leftarrow$	0x725236026A30DBD5
$X \leftarrow \omega(X, 8)$	0xD5725236026A30DB
$X \leftarrow \omega(X, 8)$	0xDBD5725236026A30
$X \leftarrow \omega(X, 8)$	0x30DBD5725236026A
$X \leftarrow \omega(X, 8)$	0x6A30DBD572523602
$X \leftarrow \omega(X, 8)$	0x026A30DBD5725236
$X \leftarrow \omega(X, 8)$	0x36026A30DBD57252
$X \leftarrow \omega(X, 8)$	0x5236026A30DBD572
$X \leftarrow \omega(X, 8)$	0x725236026A30DBD5
$X \leftarrow$	0xB7AE6CFC8558B435
$X \leftarrow \omega(X, 8)$	0x35B7AE6CFC8558B4
$X \leftarrow \omega(X, 8)$	0xB435B7AE6CFC8558
$X \leftarrow \omega(X, 8)$	0x58B435B7AE6CFC85
$X \leftarrow \omega(X, 8)$	0x8558B435B7AE6CFC
$X \leftarrow \omega(X, 8)$	0xFC8558B435B7AE6C
$X \leftarrow \omega(X, 8)$	0x6CFC8558B435B7AE
$X \leftarrow \omega(X, 8)$	0xAE6CFC8558B435B7
$X \leftarrow \omega(X, 8)$	0xB7AE6CFC8558B435

Tabla 4.5: Vectores de prueba para *omega*

Operación	**Vector**
$X \leftarrow$	0xAD99B1D3B1103401
$X \leftarrow \varsigma(X, 8)$	0x5B3363A763206802
$X \leftarrow \varsigma(X, 8)$	0xB666C64FC640D004
$X \leftarrow \varsigma(X, 8)$	0x6DCC8D9E8D80A108
$X \leftarrow \varsigma(X, 8)$	0xDA991B3D1B014310
$X \leftarrow \varsigma(X, 8)$	0xB533367A36028620
$X \leftarrow \varsigma(X, 8)$	0x6B666CF46C040D40
$X \leftarrow \varsigma(X, 8)$	0xD6CCD8E9D8081A80
$X \leftarrow \varsigma(X, 8)$	0xAD99B1D3B1103401
$X \leftarrow$	0x50F6E64A10465061
$X \leftarrow \varsigma(X, 8)$	0xA0EDCD94208CA0C2
$X \leftarrow \varsigma(X, 8)$	0x41DB9B2940194185
$X \leftarrow \varsigma(X, 8)$	0x82B737528032820B
$X \leftarrow \varsigma(X, 8)$	0x056F6EA401640516
$X \leftarrow \varsigma(X, 8)$	0x0ADEDC4902C80A2C
$X \leftarrow \varsigma(X, 8)$	0x14BDB99204911458
$X \leftarrow \varsigma(X, 8)$	0x287B7325082328B0
$X \leftarrow \varsigma(X, 8)$	0x50F6E64A10465061
$X \leftarrow$	0x725236026A30DBD5
$X \leftarrow \varsigma(X, 8)$	0xE4A46C04D460B7AB
$X \leftarrow \varsigma(X, 8)$	0xC949D808A9C06F57
$X \leftarrow \varsigma(X, 8)$	0x9392B1105381DEAE
$X \leftarrow \varsigma(X, 8)$	0x27256320A603BD5D
$X \leftarrow \varsigma(X, 8)$	0x4E4AC6404D067BBA
$X \leftarrow \varsigma(X, 8)$	0x9C948D809A0CF675
$X \leftarrow \varsigma(X, 8)$	0x39291B013518EDEA
$X \leftarrow \varsigma(X, 8)$	0x725236026A30DBD5
$X \leftarrow$	0xB7AE6CFC8558B435
$X \leftarrow \varsigma(X, 8)$	0x6F5DD8F90BB0696A
$X \leftarrow \varsigma(X, 8)$	0xDEBAB1F31661D2D4
$X \leftarrow \varsigma(X, 8)$	0xBD7563E72CC2A5A9
$X \leftarrow \varsigma(X, 8)$	0x7BEAC6CF58854B53
$X \leftarrow \varsigma(X, 8)$	0xF6D58D9FB00B96A6
$X \leftarrow \varsigma(X, 8)$	0xEDAB1B3F61162D4D
$X \leftarrow \varsigma(X, 8)$	0xDB57367EC22C5A9A
$X \leftarrow \varsigma(X, 8)$	0xB7AE6CFC8558B435

Tabla 4.6: Vectores de prueba para *sigma*

Operación	**Vector**
$X \leftarrow$	0xAD99B1D3B1103401
$X \leftarrow \xi(X, 8)$	0x1034D3B199B101AD
$X \leftarrow \xi(X, 8)$	0xB101B19934D3AD10
$X \leftarrow \xi(X, 8)$	0xD3AD993401B110B1
$X \leftarrow \xi(X, 8)$	0xB1103401AD99B1D3
$X \leftarrow \xi(X, 8)$	0x99B101AD1034D3B1
$X \leftarrow \xi(X, 8)$	0x34D3AD10B101B199
$X \leftarrow \xi(X, 8)$	0x01B110B1D3AD9934
$X \leftarrow \xi(X, 8)$	0xAD99B1D3B1103401
$X \leftarrow$	0x50F6E64A10465061
$X \leftarrow \xi(X, 8)$	0x46504A10F6E66150
$X \leftarrow \xi(X, 8)$	0xE66110F6504A5046
$X \leftarrow \xi(X, 8)$	0x4A50F650611046E6
$X \leftarrow \xi(X, 8)$	0x1046506150F6E64A
$X \leftarrow \xi(X, 8)$	0xF6E6615046504A10
$X \leftarrow \xi(X, 8)$	0x504A5046E66110F6
$X \leftarrow \xi(X, 8)$	0x611046E64A50F650
$X \leftarrow \xi(X, 8)$	0x50F6E64A10465061
$X \leftarrow$	0x725236026A30DBD5
$X \leftarrow \xi(X, 8)$	0x30DB026A5236D572
$X \leftarrow \xi(X, 8)$	0x36D56A52DB027230
$X \leftarrow \xi(X, 8)$	0x027252DBD56A3036
$X \leftarrow \xi(X, 8)$	0x6A30DBD572523602
$X \leftarrow \xi(X, 8)$	0x5236D57230DB026A
$X \leftarrow \xi(X, 8)$	0xDB02723036D56A52
$X \leftarrow \xi(X, 8)$	0xD56A3036027252DB
$X \leftarrow \xi(X, 8)$	0x725236026A30DBD5
$X \leftarrow$	0xB7AE6CFC8558B435
$X \leftarrow \xi(X, 8)$	0x58B4FC85AE6C35B7
$X \leftarrow \xi(X, 8)$	0x6C3585AEB4FCB758
$X \leftarrow \xi(X, 8)$	0xFCB7AEB43585586C
$X \leftarrow \xi(X, 8)$	0x8558B435B7AE6CFC
$X \leftarrow \xi(X, 8)$	0xAE6C35B758B4FC85
$X \leftarrow \xi(X, 8)$	0xB4FCB7586C3585AE
$X \leftarrow \xi(X, 8)$	0x3585586CFCB7AEB4
$X \leftarrow \xi(X, 8)$	0xB7AE6CFC8558B435

Tabla 4.7: Vectores de prueba para *xi*

Operación	Vector
$X \leftarrow$	0xAD99B1D3B1103401
$X \leftarrow v(X, 8)$	0x02682063A763335B
$X \leftarrow v(X, 8)$	0xB666C64FC640D004
$X \leftarrow v(X, 8)$	0x08A1808D9E8DCC6D
$X \leftarrow v(X, 8)$	0xDA991B3D1B014310
$X \leftarrow v(X, 8)$	0x208602367A3633B5
$X \leftarrow v(X, 8)$	0x6B666CF46C040D40
$X \leftarrow v(X, 8)$	0x801A08D8E9D8CCD6
$X \leftarrow v(X, 8)$	0xAD99B1D3B1103401
$X \leftarrow$	0x50F6E64A10465061
$X \leftarrow v(X, 8)$	0xC2A08C2094CDEDA0
$X \leftarrow v(X, 8)$	0x41DB9B2940194185
$X \leftarrow v(X, 8)$	0x0B8232805237B782
$X \leftarrow v(X, 8)$	0x056F6EA401640516
$X \leftarrow v(X, 8)$	0x2C0AC80249DCDE0A
$X \leftarrow v(X, 8)$	0x14BDB99204911458
$X \leftarrow v(X, 8)$	0xB028230825737B28
$X \leftarrow v(X, 8)$	0x50F6E64A10465061
$X \leftarrow$	0x725236026A30DBD5
$X \leftarrow v(X, 8)$	0xABB760D4046CA4E4
$X \leftarrow v(X, 8)$	0xC949D808A9C06F57
$X \leftarrow v(X, 8)$	0xAEDE815310B19293
$X \leftarrow v(X, 8)$	0x27256320A603BD5D
$X \leftarrow v(X, 8)$	0xBA7B064D40C64A4E
$X \leftarrow v(X, 8)$	0x9C948D809A0CF675
$X \leftarrow v(X, 8)$	0xEAED1835011B2939
$X \leftarrow v(X, 8)$	0x725236026A30DBD5
$X \leftarrow$	0xB7AE6CFC8558B435
$X \leftarrow v(X, 8)$	0x6A69B00BF9D85D6F
$X \leftarrow v(X, 8)$	0xDEBAB1F31661D2D4
$X \leftarrow v(X, 8)$	0xA9A5C22CE76375BD
$X \leftarrow v(X, 8)$	0x7BEAC6CF58854B53
$X \leftarrow v(X, 8)$	0xA6960BB09F8DD5F6
$X \leftarrow v(X, 8)$	0xEDAB1B3F61162D4D
$X \leftarrow v(X, 8)$	0x9A5A2CC27E3657DB
$X \leftarrow v(X, 8)$	0xB7AE6CFC8558B435

Tabla 4.8: Vectores de prueba para *ipsilon*

Operación	Vector
$X \leftarrow$	0x3EA74E5776AB19C7005A6E85CBD0C057D88CAB633FBCBF59BB67AA6EED2A026EDB29F5117E6D5D6FF68467027FCAE69A0B7C1C201101E45D6F0F3BFCB71D0821
$X \leftarrow \nu(X)$	0xC757596E6F9A5D2119C0BF025DE6E408ABD0BC2A6DCA011D76CB3FED7E7F11B75785636E110220FC4E6EABAAF5671C3BA75A8C6729847C0F3E00D8BBDBF60B6F
$X \leftarrow \nu(X)$	0x21081DB7FC3B0F6F5DE40111201C7C0B9AE6CA7F026784F66F5D6D7E11F529DB6E022AED6EAA67BB59BFBC3F63AB8CD857C0D0CB856E5A00C719AB76574EA73E
$X \leftarrow \nu(X)$	0x6F0BF6DBBBD8003E0F7C8429678C5AA73B1C67F5AAAB6E4EFC2002116E638557B7117F7EED3FCB761D01CA6D2ABCD0AB08E4E65D02BFC019215D9A6F6E5957C7
$X \leftarrow \nu(X)$	0x3EA74E5776AB19C7005A6E85CBD0C057D88CAB633FBCBF59BB67AA6EED2A026EDB29F5117E6D5D6FF68467027FCAE69A0B7C1C201101E45D6F0F3BFCB71D0821
$X \leftarrow 2\nu(X)$	0x21081DB7FC3B0F6F5DE40111201C7C0B9AE6CA7F026784F66F5D6D7E11F529DB6E022AED6EAA67BB59BFBC3F63AB8CD857C0D0CB856E5A00C719AB76574EA73E
$X \leftarrow 2\nu(X)$	0x3EA74E5776AB19C7005A6E85CBD0C057D88CAB633FBCBF59BB67AA6EED2A026EDB29F5117E6D5D6FF68467027FCAE69A0B7C1C201101E45D6F0F3BFCB71D0821
$X \leftarrow 2\nu(X)$	0x21081DB7FC3B0F6F5DE40111201C7C0B9AE6CA7F026784F66F5D6D7E11F529DB6E022AED6EAA67BB59BFBC3F63AB8CD857C0D0CB856E5A00C719AB76574EA73E
$X \leftarrow 3\nu(X)$	0xC757596E6F9A5D2119C0BF025DE6E408ABD0BC2A6DCA011D76CB3FED7E7F11B75785636E110220FC4E6EABAAF5671C3BA75A8C6729847C0F3E00D8BBDBF60B6F
$X \leftarrow 3\nu(X)$	0x3EA74E5776AB19C7005A6E85CBD0C057D88CAB633FBCBF59BB67AA6EED2A026EDB29F5117E6D5D6FF68467027FCAE69A0B7C1C201101E45D6F0F3BFCB71D0821
$X \leftarrow 3\nu(X)$	0x6F0BF6DBBBD8003E0F7C8429678C5AA73B1C67F5AAAB6E4EFC2002116E638557B7117F7EED3FCB761D01CA6D2ABCD0AB08E4E65D02BFC019215D9A6F6E5957C7
$X \leftarrow 3\nu(X)$	0x21081DB7FC3B0F6F5DE40111201C7C0B9AE6CA7F026784F66F5D6D7E11F529DB6E022AED6EAA67BB59BFBC3F63AB8CD857C0D0CB856E5A00C719AB76574EA73E
$X \leftarrow 3\nu(X)$	0xC757596E6F9A5D2119C0BF025DE6E408ABD0BC2A6DCA011D76CB3FED7E7F11B75785636E110220FC4E6EABAAF5671C3BA75A8C6729847C0F3E00D8BBDBF60B6F
$X \leftarrow$	0x415EF58F2C98188F35AC4B784E72BE4BBE35BA293EFDE562571B3556C1F067FDC6D96ED6E5C171B8DD561416146C8213D2E932A59E317114834EAEE3B080C730
$X \leftarrow \nu(X)$	0x8F4B62FDB813143018BEE567718271C79872FDF0C16C31802C4E3EC1E5149EB08F782956D616A5E3F54BBA356E1432AE5EAC351BD956E94E4135BE57C6DDD283
$X \leftarrow \nu(X)$	0x30C780B0E3AE4E831471319EA532E9D213826C14161456DDB871C1E5D66ED9C6FD67F0C156351B5762E5FD3E29BA35BE4BBE724E784BAC358F18982C8FF55E41
$X \leftarrow \nu(X)$	0x83D2DDC657BE35414EE956D91B35AC5EAE32146E35BA4BF5E3A516D65629788FB09E14E5C13E4E2C80316CC1F0FD7298C771827167E5BE18301413B8FD624B8F
$X \leftarrow \nu(X)$	0x415EF58F2C98188F35AC4B784E72BE4BBE35BA293EFDE562571B3556C1F067FDC6D96ED6E5C171B8DD561416146C8213D2E932A59E317114834EAEE3B080C730
$X \leftarrow 2\nu(X)$	0x30C780B0E3AE4E831471319EA532E9D213826C14161456DDB871C1E5D66ED9C6FD67F0C156351B5762E5FD3E29BA35BE4BBE724E784BAC358F18982C8FF55E41
$X \leftarrow 2\nu(X)$	0x415EF58F2C98188F35AC4B784E72BE4BBE35BA293EFDE562571B3556C1F067FDC6D96ED6E5C171B8DD561416146C8213D2E932A59E317114834EAEE3B080C730
$X \leftarrow 2\nu(X)$	0x30C780B0E3AE4E831471319EA532E9D213826C14161456DDB871C1E5D66ED9C6FD67F0C156351B5762E5FD3E29BA35BE4BBE724E784BAC358F18982C8FF55E41
$X \leftarrow 3\nu(X)$	0x8F4B62FDB813143018BEE567718271C79872FDF0C16C31802C4E3EC1E5149EB08F782956D616A5E3F54BBA356E1432AE5EAC351BD956E94E4135BE57C6DDD283
$X \leftarrow 3\nu(X)$	0x415EF58F2C98188F35AC4B784E72BE4BBE35BA293EFDE562571B3556C1F067FDC6D96ED6E5C171B8DD561416146C8213D2E932A59E317114834EAEE3B080C730
$X \leftarrow 3\nu(X)$	0x83D2DDC657BE35414EE956D91B35AC5EAE32146E35BA4BF5E3A516D65629788FB09E14E5C13E4E2C80316CC1F0FD7298C771827167E5BE18301413B8FD624B8F
$X \leftarrow 3\nu(X)$	0x30C780B0E3AE4E831471319EA532E9D213826C14161456DDB871C1E5D66ED9C6FD67F0C156351B5762E5FD3E29BA35BE4BBE724E784BAC358F18982C8FF55E41
$X \leftarrow 3\nu(X)$	0x8F4B62FDB813143018BEE567718271C79872FDF0C16C31802C4E3EC1E5149EB08F782956D616A5E3F54BBA356E1432AE5EAC351BD956E94E4135BE57C6DDD283
$X \leftarrow$	0xE9CA4CC76B8260F2F2C6A12BC6CB6F9F856C27BF76411EF33E003B7DA01E185A7AB546F063E4ABE32311961AEB512E5BE27B632D8747AD1DBA6F1D08A7BED018
$X \leftarrow \nu(X)$	0xF29FF35AE35B1D18606F1E18AB2EADD082CB411EE45147BE6BC676A063EB87A7C72BBF7DF01A2D084CA1273B4696631DCAC66C00B5117B6FE9F2853E7A23E2BA
$X \leftarrow \nu(X)$	0x18D0BEA7081D6FBA1DAD47872D637BE25B2E51EB1A961123E3ABE463F046B57A5A181EA07D3B003EF31E4176BF276C859F6FCBC62BA1C6F2F260826BC74CCAE9
$X \leftarrow \nu(X)$	0xBAE2237A3E85F2E96F7B11B5006CC6CA1D6396463B27A14C082D1AF07DBF2BC7A787EB63A076C66BBE4751E41E41CB82D0AD2EAB181E6F60181D5BE35AF39FF2
$X \leftarrow \nu(X)$	0xE9CA4CC76B8260F2F2C6A12BC6CB6F9F856C27BF76411EF33E003B7DA01E185A7AB546F063E4ABE32311961AEB512E5BE27B632D8747AD1DBA6F1D08A7BED018
$X \leftarrow 2\nu(X)$	0x18D0BEA7081D6FBA1DAD47872D637BE25B2E51EB1A961123E3ABE463F046B57A5A181EA07D3B003EF31E4176BF276C859F6FCBC62BA1C6F2F260826BC74CCAE9
$X \leftarrow 2\nu(X)$	0xE9CA4CC76B8260F2F2C6A12BC6CB6F9F856C27BF76411EF33E003B7DA01E185A7AB546F063E4ABE32311961AEB512E5BE27B632D8747AD1DBA6F1D08A7BED018
$X \leftarrow 2\nu(X)$	0x18D0BEA7081D6FBA1DAD47872D637BE25B2E51EB1A961123E3ABE463F046B57A5A181EA07D3B003EF31E4176BF276C859F6FCBC62BA1C6F2F260826BC74CCAE9
$X \leftarrow 3\nu(X)$	0xF29FF35AE35B1D18606F1E18AB2EADD082CB411EE45147BE6BC676A063EB87A7C72BBF7DF01A2D084CA1273B4696631DCAC66C00B5117B6FE9F2853E7A23E2BA
$X \leftarrow 3\nu(X)$	0xE9CA4CC76B8260F2F2C6A12BC6CB6F9F856C27BF76411EF33E003B7DA01E185A7AB546F063E4ABE32311961AEB512E5BE27B632D8747AD1DBA6F1D08A7BED018
$X \leftarrow 3\nu(X)$	0xBAE2237A3E85F2E96F7B11B5006CC6CA1D6396463B27A14C082D1AF07DBF2BC7A787EB63A076C66BBE4751E41E41CB82D0AD2EAB181E6F60181D5BE35AF39FF2
$X \leftarrow 3\nu(X)$	0x18D0BEA7081D6FBA1DAD47872D637BE25B2E51EB1A961123E3ABE463F046B57A5A181EA07D3B003EF31E4176BF276C859F6FCBC62BA1C6F2F260826BC74CCAE9
$X \leftarrow 3\nu(X)$	0xF29FF35AE35B1D18606F1E18AB2EADD082CB411EE45147BE6BC676A063EB87A7C72BBF7DF01A2D084CA1273B4696631DCAC66C00B5117B6FE9F2853E7A23E2BA

Tabla 4.9: Vectores de prueba para *ni*

Operación	Vector
$X \leftarrow$	0x3EA74E5776AB19C7005A6E85CBD0C057D88CAB633FBCBF59BB67AA6EED2A026EDB29F5117E6D5D6FF68467027FCAE69A0B7C1C201101E45D6F0F3BFCB71D0821
$X \leftarrow \eta(X)$	0x3C251C828A276621B44A338D3B32725FA959BECCBD1D64983D42D091F5861566DDD6CF883EBD957CA1ED738BBD94745BF98BAF2DB7292C3BD899B417EED115D7
$X \leftarrow \eta(X)$	0xF6F0DC3FEDB81084D03E3804888027BA6F21E640FE536759DB94AF887EB6BAF6DDE65576B75440761B31D5C6FC3DFD9A005A76A1D30B03EA7CE572EA6ED598E3
$X \leftarrow \eta(X)$	0x1B992DE8778BA8EB9FD1F5B4ED9434DC85B7CED1BD292EDABB6BF3117CBDA93EBC420B89AF61A866959A7D33BDB826192D52CCB1DC4C4EFA3CA4384151E46684
$X \leftarrow \eta(X)$	0x3EA74E5776AB19C7005A6E85CBD0C057D88CAB633FBCBF59BB67AA6EED2A026EDB29F5117E6D5D6FF68467027FCAE69A0B7C1C201101E45D6F0F3BFCB71D0821
$X \leftarrow 2\eta(X)$	0xF6F0DC3FEDB81084D03E3804888027BA6F21E640FE536759DB94AF887EB6BAF6DDE65576B75440761B31D5C6FC3DFD9A005A76A1D30B03EA7CE572EA6ED598E3
$X \leftarrow 2\eta(X)$	0x3EA74E5776AB19C7005A6E85CBD0C057D88CAB633FBCBF59BB67AA6EED2A026EDB29F5117E6D5D6FF68467027FCAE69A0B7C1C201101E45D6F0F3BFCB71D0821
$X \leftarrow 2\eta(X)$	0xF6F0DC3FEDB81084D03E3804888027BA6F21E640FE536759DB94AF887EB6BAF6DDE65576B75440761B31D5C6FC3DFD9A005A76A1D30B03EA7CE572EA6ED598E3
$X \leftarrow 3\eta(X)$	0x3C251C828A276621B44A338D3B32725FA959BECCBD1D64983D42D091F5861566DDD6CF883EBD957CA1ED738BBD94745BF98BAF2DB7292C3BD899B417EED115D7
$X \leftarrow 3\eta(X)$	0x3EA74E5776AB19C7005A6E85CBD0C057D88CAB633FBCBF59BB67AA6EED2A026EDB29F5117E6D5D6FF68467027FCAE69A0B7C1C201101E45D6F0F3BFCB71D0821
$X \leftarrow 3\eta(X)$	0x1B992DE8778BA8EB9FD1F5B4ED9434DC85B7CED1BD292EDABB6BF3117CBDA93EBC420B89AF61A866959A7D33BDB826192D52CCB1DC4C4EFA3CA4384151E46684
$X \leftarrow 3\eta(X)$	0xF6F0DC3FEDB81084D03E3804888027BA6F21E640FE536759DB94AF887EB6BAF6DDE65576B75440761B31D5C6FC3DFD9A005A76A1D30B03EA7CE572EA6ED598E3
$X \leftarrow 3\eta(X)$	0x3C251C828A276621B44A338D3B32725FA959BECCBD1D64983D42D091F5861566DDD6CF883EBD957CA1ED738BBD94745BF98BAF2DB7292C3BD899B417EED115D7
$X \leftarrow$	0x415EF58F2C98188F35AC4B784E72BE4BBE35BA293EFDE562571B3556C1F067FDC6D96ED6E5C171B8DD561416146C8213D2E932A59E317114834EAEE3B080C730
$X \leftarrow \eta(X)$	0xF45285D1D89DA61979F1139A1A3EDC0E0646DDC6956E5E9C6E3D6D3AE44F53F824DB96074725031B3EA7B97977248E49DCA9D6B94602AA27AB5C0BC51854DC2B
$X \leftarrow \eta(X)$	0xC17275C70D01E30C4B974CA5798C8E28BB6A2868283641C8639B766BA7838E1DEAD8AC6A830FE6BF7DAC5D947CBFA746AC35D21E724E7DD2827AAFF1341918F1
$X \leftarrow \eta(X)$	0xD53AD0A3182A3BD43B956B9D624055E47CE59D9EEE24719224DB69E0E2A4C0D876BCB65C27F2CA1F6062BB63A9767A399E8FC859587C3B702F4AA18B1BB96598
$X \leftarrow \eta(X)$	0x415EF58F2C98188F35AC4B784E72BE4BBE35BA293EFDE562571B3556C1F067FDC6D96ED6E5C171B8DD561416146C8213D2E932A59E317114834EAEE3B080C730
$X \leftarrow 2\eta(X)$	0xC17275C70D01E30C4B974CA5798C8E28BB6A2868283641C8639B766BA7838E1DEAD8AC6A830FE6BF7DAC5D947CBFA746AC35D21E724E7DD2827AAFF1341918F1
$X \leftarrow 2\eta(X)$	0x415EF58F2C98188F35AC4B784E72BE4BBE35BA293EFDE562571B3556C1F067FDC6D96ED6E5C171B8DD561416146C8213D2E932A59E317114834EAEE3B080C730
$X \leftarrow 2\eta(X)$	0xC17275C70D01E30C4B974CA5798C8E28BB6A2868283641C8639B766BA7838E1DEAD8AC6A830FE6BF7DAC5D947CBFA746AC35D21E724E7DD2827AAFF1341918F1
$X \leftarrow 3\eta(X)$	0xF45285D1D89DA61979F1139A1A3EDC0E0646DDC6956E5E9C6E3D6D3AE44F53F824DB96074725031B3EA7B97977248E49DCA9D6B94602AA27AB5C0BC51854DC2B
$X \leftarrow 3\eta(X)$	0x415EF58F2C98188F35AC4B784E72BE4BBE35BA293EFDE562571B3556C1F067FDC6D96ED6E5C171B8DD561416146C8213D2E932A59E317114834EAEE3B080C730
$X \leftarrow 3\eta(X)$	0xD53AD0A3182A3BD43B956B9D624055E47CE59D9EEE24719224DB69E0E2A4C0D876BCB65C27F2CA1F6062BB63A9767A399E8FC859587C3B702F4AA18B1BB96598
$X \leftarrow 3\eta(X)$	0xC17275C70D01E30C4B974CA5798C8E28BB6A2868283641C8639B766BA7838E1DEAD8AC6A830FE6BF7DAC5D947CBFA746AC35D21E724E7DD2827AAFF1341918F1
$X \leftarrow 3\eta(X)$	0xF45285D1D89DA61979F1139A1A3EDC0E0646DDC6956E5E9C6E3D6D3AE44F53F824DB96074725031B3EA7B97977248E49DCA9D6B94602AA27AB5C0BC51854DC2B
$X \leftarrow$	0xE9CA4CC76B8260F2F2C6A12BC6CB6F9F856C27BF76411EF33E003B7DA01E185A7AB546F063E4ABE32311961AEB512E5BE27B632D8747AD1DBA6F1D08A7BED018
$X \leftarrow \eta(X)$	0xC7132215EA93D01753C751193776833DFBD44E5EBD9073159A70A83C04A88CEF99C589EE218A7EEA0C96B54DC6D86642FAC37C27F7CB363F25F0CE4FF1665276
$X \leftarrow \eta(X)$	0x5DF6B810E57D0B1847DEC6B4E1E2B5B8C4886958D78A74DA5EAD620FC627D5C77C00DCBE0578185AA136E4FD6E8278CF4F6385D463D3F6F9975332E3D641064F
$X \leftarrow \eta(X)$	0xA40F73F28F664A6E5FC33EE4EFD36CFC3069ADB2631B664299A3917784517E57590E153C201531F7DF2B727ABD09CEA8CAE38A98EC6EC1BCE3C844A857C90BE8
$X \leftarrow \eta(X)$	0xE9CA4CC76B8260F2F2C6A12BC6CB6F9F856C27BF76411EF33E003B7DA01E185A7AB546F063E4ABE32311961AEB512E5BE27B632D8747AD1DBA6F1D08A7BED018
$X \leftarrow 2\eta(X)$	0x5DF6B810E57D0B1847DEC6B4E1E2B5B8C4886958D78A74DA5EAD620FC627D5C77C00DCBE0578185AA136E4FD6E8278CF4F6385D463D3F6F9975332E3D641064F
$X \leftarrow 2\eta(X)$	0xE9CA4CC76B8260F2F2C6A12BC6CB6F9F856C27BF76411EF33E003B7DA01E185A7AB546F063E4ABE32311961AEB512E5BE27B632D8747AD1DBA6F1D08A7BED018
$X \leftarrow 2\eta(X)$	0x5DF6B810E57D0B1847DEC6B4E1E2B5B8C4886958D78A74DA5EAD620FC627D5C77C00DCBE0578185AA136E4FD6E8278CF4F6385D463D3F6F9975332E3D641064F
$X \leftarrow 3\eta(X)$	0xC7132215EA93D01753C751193776833DFBD44E5EBD9073159A70A83C04A88CEF99C589EE218A7EEA0C96B54DC6D86642FAC37C27F7CB363F25F0CE4FF1665276
$X \leftarrow 3\eta(X)$	0xE9CA4CC76B8260F2F2C6A12BC6CB6F9F856C27BF76411EF33E003B7DA01E185A7AB546F063E4ABE32311961AEB512E5BE27B632D8747AD1DBA6F1D08A7BED018
$X \leftarrow 3\eta(X)$	0xA40F73F28F664A6E5FC33EE4EFD36CFC3069ADB2631B664299A3917784517E57590E153C201531F7DF2B727ABD09CEA8CAE38A98EC6EC1BCE3C844A857C90BE8
$X \leftarrow 3\eta(X)$	0x5DF6B810E57D0B1847DEC6B4E1E2B5B8C4886958D78A74DA5EAD620FC627D5C77C00DCBE0578185AA136E4FD6E8278CF4F6385D463D3F6F9975332E3D641064F
$X \leftarrow 3\eta(X)$	0xC7132215EA93D01753C751193776833DFBD44E5EBD9073159A70A83C04A88CEF99C589EE218A7EEA0C96B54DC6D86642FAC37C27F7CB363F25F0CE4FF1665276

Tabla 4.10: Vectores de prueba para *eta*

Operación	Vector
$X \leftarrow$	0x2CD20C9DF528CDED6A49CF01A18DADE8597B38FDFC7AE1BEC9033E72DA3F29BF665332F512164EEACF07B1A8491CA1269BC729C207A2136F5B635779F963BC8C
$X \leftarrow \zeta(X, 0x0F)$	0x23DD0392FA27C2E26546C00EAE82A2E7567437F2F375EEB1C60C317DD53026B0695C3DFA1D1941E5C008BEA74613AE2994C826CD08AD1C60546C5876F66CB383
$X \leftarrow \zeta(X, 0x0F)$	0x2CD20C9DF528CDED6A49CF01A18DADE8597B38FDFC7AE1BEC9033E72DA3F29BF665332F512164EEACF07B1A8491CA1269BC729C207A2136F5B635779F963BC8C
$X \leftarrow \zeta(X, 0xF0)$	0xDC22FC6D05D83D1D9AB93FF1517D5D18A98BC80D0C8A114E39F3CE822ACFD94F96A3C205E2E6BE1A3FF74158B9EC51D66B37D932F752E39FAB93A78909934C7C
$X \leftarrow \zeta(X, 0xF0)$	0x2CD20C9DF528CDED6A49CF01A18DADE8597B38FDFC7AE1BEC9033E72DA3F29BF665332F512164EEACF07B1A8491CA1269BC729C207A2136F5B635779F963BC8C
$X \leftarrow$	0xF01320119D340FD0C2E2D7B2AB1057B97A0ED468945124F1725BBD53776D8A68DDF94ED553452A26AE292C346F83A2CA3406845483574B0A4F43C25BA5CE6D40
$X \leftarrow \zeta(X, 0x0F)$	0xFF1C2F1E923B00DFCDEDD8BDA41F58B67501DB679B5E2BFE7D54B25C78628567D2F641DA5C4A2529A126233B608CADC53B098B5B8C584405404CCD54AAC1624F
$X \leftarrow \zeta(X, 0x0F)$	0xF01320119D340FD0C2E2D7B2AB1057B97A0ED468945124F1725BBD53776D8A68DDF94ED553452A26AE292C346F83A2CA3406845483574B0A4F43C25BA5CE6D40
$X \leftarrow \zeta(X, 0xF0)$	0x00E3D0E16DC4FF20321227425BE0A7498AFE249864A1D40182AB4DA3879D7A982D09BE25A3B5DAD65ED9DCC49F73523AC4F674A473A7BBFABFB332AB553E9DB0
$X \leftarrow \zeta(X, 0xF0)$	0xF01320119D340FD0C2E2D7B2AB1057B97A0ED468945124F1725BBD53776D8A68DDF94ED553452A26AE292C346F83A2CA3406845483574B0A4F43C25BA5CE6D40
$X \leftarrow$	0x9BAB3805844346DFA5933234B17E1989818E54B72F757C2BFAC93A447A9F55C2BC1DCF693979C5E2C352D40431767FE384A3BE6218A2756ADD64883CF728169C
$X \leftarrow \zeta(X, 0x0F)$	0x94A4370A8B4C49D0AA9C3D3BBE7116868E815BB8207A7324F5C6354B75905ACDB312C0663676CAEDCC5DDB0B3E7970EC8BACB16D17AD7A65D26B8733F8271993
$X \leftarrow \zeta(X, 0x0F)$	0x9BAB3805844346DFA5933234B17E1989818E54B72F757C2BFAC93A447A9F55C2BC1DCF693979C5E2C352D40431767FE384A3BE6218A2756ADD64883CF728169C
$X \leftarrow \zeta(X, 0xF0)$	0x6B5BC8F574B3B62F5563C2C4418EE979717EA447DF858CDB0A39CAB48A6FA5324CED3F99C989351233A224F4C1868F1374534E92E852859A2D9478CC07D8E66C
$X \leftarrow \zeta(X, 0xF0)$	0x9BAB3805844346DFA5933234B17E1989818E54B72F757C2BFAC93A447A9F55C2BC1DCF693979C5E2C352D40431767FE384A3BE6218A2756ADD64883CF728169C
$X \leftarrow$	0x2F9CE53F9644039A1B058065FA5B29556193C60E62AC4B30CD99991B16743BC39CCCF7CB09D65812505F989664CB5D7B980C1C5E5160AE4BB16F3ADA25EA9634
$X \leftarrow \zeta(X, 0x0F)$	0x2093EA30994B0C95140A8F6AF554265A6E9CC9016DA3443FC2969614197B34CC93C3F8C406D9571D5F5097996BC45274970313515E6FA144BE6035D52AE5993B
$X \leftarrow \zeta(X, 0x0F)$	0x2F9CE53F9644039A1B058065FA5B29556193C60E62AC4B30CD99991B16743BC39CCCF7CB09D65812505F989664CB5D7B980C1C5E5160AE4BB16F3ADA25EA9634
$X \leftarrow \zeta(X, 0xF0)$	0xDF6C15CF66B4F36AEBF570950AABD9A5916336FE925CBBC03D6969EBE684CB336C3C073BF926A8E2A0AF6866943BAD8B68FCECAEA1905EBB419FCA2AD51A66C4
$X \leftarrow \zeta(X, 0xF0)$	0x2F9CE53F9644039A1B058065FA5B29556193C60E62AC4B30CD99991B16743BC39CCCF7CB09D65812505F989664CB5D7B980C1C5E5160AE4BB16F3ADA25EA9634
$X \leftarrow$	0xDEBA34D0DFFEF5204CEBE2638400A225EDB497EBD604F2734A4D84A25BA1EF5994740A3492CD08FFC76A22EDFB84EEB10D9F22EFCAD1118AC9AFB4F682806F85
$X \leftarrow \zeta(X, 0x0F)$	0xD1B53BDFD0F1FA2F43E4ED6C8B0FAD2AE2BB98E4D90BFD7C45428BAD54AEE0569B7B053B9DC207F0C8652DE2F48BE1BE02902DE0C5DE1E85C6A0BBF98D8F608A
$X \leftarrow \zeta(X, 0x0F)$	0xDEBA34D0DFFEF5204CEBE2638400A225EDB497EBD604F2734A4D84A25BA1EF5994740A3492CD08FFC76A22EDFB84EEB10D9F22EFCAD1118AC9AFB4F682806F85
$X \leftarrow \zeta(X, 0xF0)$	0x2E4AC4202F0E05D0BC1B129374F052D51D44671B26F40283BABD7452AB511FA96484FAC4623DF80F379AD21D0B741E41FD6FD21F3A21E17A395F440672709F75
$X \leftarrow \zeta(X, 0xF0)$	0xDEBA34D0DFFEF5204CEBE2638400A225EDB497EBD604F2734A4D84A25BA1EF5994740A3492CD08FFC76A22EDFB84EEB10D9F22EFCAD1118AC9AFB4F682806F85
$X \leftarrow$	0x0018AE2752F96CF9244BB74B6657DCAB87026D42E80DD6FA870B8BE294E4F45064B8A96E40D7780A5A85F758DE84F0EC1DE3D0AB6B67A26E1C3F1685646D1461
$X \leftarrow \zeta(X, 0x0F)$	0x0F17A1285DF663F62B44B8446958D3A4880D624DE702D9F5880484ED9BEBFB5F6BB7A6614FD87705558AF857D18BFFE312ECDFA46468AD611330198A6B621B6E
$X \leftarrow \zeta(X, 0x0F)$	0x0018AE2752F96CF9244BB74B6657DCAB87026D42E80DD6FA870B8BE294E4F45064B8A96E40D7780A5A85F758DE84F0EC1DE3D0AB6B67A26E1C3F1685646D1461
$X \leftarrow \zeta(X, 0xF0)$	0xF0E85ED7A2099C09D4BB47BB96A72C5B77F29DB218FD260A77FB7B12641404A09448599EB02788FAAA7507A82E74001CED13205B9B97529EECCFE675949DE491
$X \leftarrow \zeta(X, 0xF0)$	0x0018AE2752F96CF9244BB74B6657DCAB87026D42E80DD6FA870B8BE294E4F45064B8A96E40D7780A5A85F758DE84F0EC1DE3D0AB6B67A26E1C3F1685646D1461
$X \leftarrow$	0x44BF551CC2C0071D3C137B80820881EE06F7FA4E6D3A6850FA88957A786794A9084B97D716EA094F17DD9739110BD40D2A0B047321239FEAF92FE84FEA804558
$X \leftarrow \zeta(X, 0x0F)$	0x4BB05A13CDCF0812331C748F8D078EE109F8F5416235675FF5879A7577689BA6074498D819E5064018D298361E04DB0225040B7C2E2C90E5F620E740E58F4A57
$X \leftarrow \zeta(X, 0x0F)$	0x44BF551CC2C0071D3C137B80820881EE06F7FA4E6D3A6850FA88957A786794A9084B97D716EA094F17DD9739110BD40D2A0B047321239FEAF92FE84FEA804558
$X \leftarrow \zeta(X, 0xF0)$	0xB44FA5EC3230F7EDCCE38B7072F8711EF6070ABE9DCA98A00A78658A88976459F8BB6727E61AF9BFE72D67C9E1FB24FDDAFBF483D1D36F1A09DF18BF1A70B5A8
$X \leftarrow \zeta(X, 0xF0)$	0x44BF551CC2C0071D3C137B80820881EE06F7FA4E6D3A6850FA88957A786794A9084B97D716EA094F17DD9739110BD40D2A0B047321239FEAF92FE84FEA804558
$X \leftarrow$	0xC48842ECE0B10CCC1BA0A0A88BCEF93CB28C7D6428B205CA7C83B0DAF102E6FAE5ACCB89E941642EE912009B4E2CED0EA96418B3C6808DE1E2189E0D87B1739D
$X \leftarrow \zeta(X, 0x0F)$	0xCB874DE3EFBE03C314AFAFA784C1F633BD83726B27BD0AC5738CBFD5FE0DE9F5EAA3C486E64E6B21E61D0F944123E201A66B17BCC98F82EEED17910288BE7C92
$X \leftarrow \zeta(X, 0x0F)$	0xC48842ECE0B10CCC1BA0A0A88BCEF93CB28C7D6428B205CA7C83B0DAF102E6FAE5ACCB89E941642EE912009B4E2CED0EA96418B3C6808DE1E2189E0D87B1739D
$X \leftarrow \zeta(X, 0xF0)$	0x3478B21C1041FC3CEB5050587B3E09CC427C8D94D842F53A8C73402A01F2160A155C3B7919B194DE19E2F06BBEDC1DFE5994E84336707D1112E86EFD7741836D
$X \leftarrow \zeta(X, 0xF0)$	0xC48842ECE0B10CCC1BA0A0A88BCEF93CB28C7D6428B205CA7C83B0DAF102E6FAE5ACCB89E941642EE912009B4E2CED0EA96418B3C6808DE1E2189E0D87B1739D
$X \leftarrow$	0x958C5445011C6CD7565567262EA8E5125FD267809313E9E3982A38C6814D2180B937AC17D46C984F66D04EDC5EBA8447103CC66DB6E10CF7AB7B809EB325E059
$X \leftarrow \zeta(X, 0x0F)$	0x9A835B4A0E1363D8595A682921A7EA1D50DD688F9C1CE6EC972537C98E422E8FB638A318DB63974069DF41D351B58B481F33C962B9EE03F8A4748F91BC2AEF56
$X \leftarrow \zeta(X, 0x0F)$	0x958C5445011C6CD7565567262EA8E5125FD267809313E9E3982A38C6814D2180B937AC17D46C984F66D04EDC5EBA8447103CC66DB6E10CF7AB7B809EB325E059
$X \leftarrow \zeta(X, 0xF0)$	0x657CA4B5F1EC9C27A6A597D6DE5815E2AF22977063E3191368DAC83671BDD17049C75CE7249C68BF9620BE2CAE4A74B7E0CC369D4611FC075B8B706E43D510A9
$X \leftarrow \zeta(X, 0xF0)$	0x958C5445011C6CD7565567262EA8E5125FD267809313E9E3982A38C6814D2180B937AC17D46C984F66D04EDC5EBA8447103CC66DB6E10CF7AB7B809EB325E059
$X \leftarrow$	0xEB4032057154F3E0D8A0C749F9AAD8C0AFFB8B571F7FE0AD431500A698006BCCD6559B24F31E0180C44C60689DEC2364DAF2CBB2753FD955089A27C8ABFAEC52
$X \leftarrow \zeta(X, 0x0F)$	0xE44F3D0A7E5BFCEFD7AFC846F6A5D7CFA0F484581070EFA24C1A0FA9970F64C3D95A942BFC110E8FCB436F6792E32C6BD5FDC4BD7A30D65A079528C7A4F5E35D
$X \leftarrow \zeta(X, 0x0F)$	0xEB4032057154F3E0D8A0C749F9AAD8C0AFFB8B571F7FE0AD431500A698006BCCD6559B24F31E0180C44C60689DEC2364DAF2CBB2753FD955089A27C8ABFAEC52
$X \leftarrow \zeta(X, 0xF0)$	0x1BB0C2F581A40310285037B9095A28305F0B7BA7EF8F105DB3E5F05668F09B3C26A56BD403EEF17034BC90986D1CD3942A023B4285CF29A5F86AD7385B0A1CA2
$X \leftarrow \zeta(X, 0xF0)$	0xEB4032057154F3E0D8A0C749F9AAD8C0AFFB8B571F7FE0AD431500A698006BCCD6559B24F31E0180C44C60689DEC2364DAF2CBB2753FD955089A27C8ABFAEC52

Tabla 4.11: Vectores de prueba para *dseta*

Operación	Vector
$X \leftarrow$	0x2CD20C9DF528CDED6A49CF01A18DADE8597B38FDFC7AE1BEC9033E72DA3F29BF665332F512164EEACF07B1A8491CA1269BC729C207A2136F5B635779F963BC8C
$X \leftarrow \mu(X, 0x0F)$	0x2CD20C9D0AD732126A49CF015E725217597B38FD03851E41C9033E7225C0D640665332F5EDE9B115CF07B1A8B6E35ED99BC729C2F85DEC905B635779069C4373
$X \leftarrow \mu(X, 0x0F)$	0x2CD20C9DF528CDED6A49CF01A18DADE8597B38FDFC7AE1BEC9033E72DA3F29BF665332F512164EEACF07B1A8491CA1269BC729C207A2136F5B635779F963BC8C
$X \leftarrow \mu(X, 0xF0)$	0xD32DF362F528CDED95B630FEA18DADE8A684C702FC7AE1BE36FCC18DDA3F29BF99ACCD0A12164EEA30F84E57491CA1266438D63D07A2136FA49CA886F963BC8C
$X \leftarrow \mu(X, 0xF0)$	0x2CD20C9DF528CDED6A49CF01A18DADE8597B38FDFC7AE1BEC9033E72DA3F29BF665332F512164EEACF07B1A8491CA1269BC729C207A2136F5B635779F963BC8C
$X \leftarrow$	0xF01320119D340FD0C2E2D7B2AB1057B97A0ED468945124F1725BBD53776D8A68DDF94ED553452A26AE292C346F83A2CA3406845483574B0A4F43C25BA5CE6D40
$X \leftarrow \mu(X, 0x0F)$	0xF013201162CBF02FC2E2D7B254EFA8467A0ED4686BAEDB0E725BBD5388927597DDF94ED5ACBAD5D9AE292C34907C5D35340684547CA8B4F54F43C25B5A3192BF
$X \leftarrow \mu(X, 0x0F)$	0xF01320119D340FD0C2E2D7B2AB1057B97A0ED468945124F1725BBD53776D8A68DDF94ED553452A26AE292C346F83A2CA3406845483574B0A4F43C25BA5CE6D40
$X \leftarrow \mu(X, 0xF0)$	0x0FECDFEE9D340FD03D1D284DAB1057B985F12B97945124F18DA442AC776D8A682206B12A53452A2651D6D3CB6F83A2CACBF97BAB83574B0AB0BC3DA4A5CE6D40
$X \leftarrow \mu(X, 0xF0)$	0xF01320119D340FD0C2E2D7B2AB1057B97A0ED468945124F1725BBD53776D8A68DDF94ED553452A26AE292C346F83A2CA3406845483574B0A4F43C25BA5CE6D40
$X \leftarrow$	0x9BAB3805844346DFA5933234B17E1989818E54B72F757C2BFAC93A447A9F55C2BC1DCF693979C5E2C352D40431767FE384A3BE6218A2756ADD64883CF728169C
$X \leftarrow \mu(X, 0x0F)$	0x9BAB38057BBCB920A59332344E81E676818E54B7D08A83D4FAC93A448560AA3DBC1DCF69C6863A1DC352D404CE89801C84A3BE62E75D8A95DD64883C08D7E963
$X \leftarrow \mu(X, 0x0F)$	0x9BAB3805844346DFA5933234B17E1989818E54B72F757C2BFAC93A447A9F55C2BC1DCF693979C5E2C352D40431767FE384A3BE6218A2756ADD64883CF728169C
$X \leftarrow \mu(X, 0xF0)$	0x6454C7FA844346DF5A6CCDCBB17E19897E71AB482F757C2B0536C5BB7A9F55C243E230963979C5E23CAD2BFB31767FE37B5C419D18A2756A229B77C3F728169C
$X \leftarrow \mu(X, 0xF0)$	0x9BAB3805844346DFA5933234B17E1989818E54B72F757C2BFAC93A447A9F55C2BC1DCF693979C5E2C352D40431767FE384A3BE6218A2756ADD64883CF728169C
$X \leftarrow$	0x2F9CE53F9644039A1B058065FA5B29556193C60E62AC4B30CD99991B16743BC39CCCF7CB09D65812505F989664CB5D7B980C1C5E5160AE4BB16F3ADA25EA9634
$X \leftarrow \mu(X, 0x0F)$	0x2F9CE53F69BBFC651B05806505A4D6AA6193C60E9D53B4CFCD99991BE98BC43C9CCCF7CBF629A7ED505F98969B34A284980C1C5EAE9F51B4B16F3ADADA1569CB
$X \leftarrow \mu(X, 0x0F)$	0x2F9CE53F9644039A1B058065FA5B29556193C60E62AC4B30CD99991B16743BC39CCCF7CB09D65812505F989664CB5D7B980C1C5E5160AE4BB16F3ADA25EA9634
$X \leftarrow \mu(X, 0xF0)$	0xD0631AC09644039AE4FA7F9AFA5B29559E6C39F162AC4B30326666E416743BC36333083409D65812AFA0676964CB5D7B67F3E3A15160AE4B4E90C52525EA9634
$X \leftarrow \mu(X, 0xF0)$	0x2F9CE53F9644039A1B058065FA5B29556193C60E62AC4B30CD99991B16743BC39CCCF7CB09D65812505F989664CB5D7B980C1C5E5160AE4BB16F3ADA25EA9634
$X \leftarrow$	0xDEBA34D0DFFEF5204CEBE2638400A225EDB497EBD604F2734A4D84A25BA1EF5994740A3492CD08FFC76A22EDFB84EEB10D9F22EFCAD1118AC9AFB4F682806F85
$X \leftarrow \mu(X, 0x0F)$	0xDEBA34D020010ADF4CEBE2637BFF5DDAEDB497EB29FB0D8C4A4D84A2A45E10A694740A346D32F700C76A22ED047B114E0D9F22EF352EEE75C9AFB4F67D7F907A
$X \leftarrow \mu(X, 0x0F)$	0xDEBA34D0DFFEF5204CEBE2638400A225EDB497EBD604F2734A4D84A25BA1EF5994740A3492CD08FFC76A22EDFB84EEB10D9F22EFCAD1118AC9AFB4F682806F85
$X \leftarrow \mu(X, 0xF0)$	0x2145CB2FDFFEF520B3141D9C8400A225124B6814D604F273B5B27B5D5BA1EF596B8BF5CB92CD08FF3895DD12FB84EEB1F260DD10CAD1118A36504B0982806F85
$X \leftarrow \mu(X, 0xF0)$	0xDEBA34D0DFFEF5204CEBE2638400A225EDB497EBD604F2734A4D84A25BA1EF5994740A3492CD08FFC76A22EDFB84EEB10D9F22EFCAD1118AC9AFB4F682806F85
$X \leftarrow$	0x0018AE2752F96CF9244BB74B6657DCAB87026D42E80DD6FA870B8BE294E4F45064B8A96E40D7780A5A85F758DE84F0EC1DE3D0AB6B67A26E1C3F1685646D1461
$X \leftarrow \mu(X, 0x0F)$	0x0018AE27AD069306244BB74B99A8235487026D4217F22905870B8BE26B1B0BAF64B8A96EBF2887F55A85F758217B0F131DE3D0AB94985D911C3F16859B92EB9E
$X \leftarrow \mu(X, 0x0F)$	0x0018AE2752F96CF9244BB74B6657DCAB87026D42E80DD6FA870B8BE294E4F45064B8A96E40D7780A5A85F758DE84F0EC1DE3D0AB6B67A26E1C3F1685646D1461
$X \leftarrow \mu(X, 0xF0)$	0xFFE751D852F96CF9DBB448B46657DCAB78FD92BDE80DD6FA78F4741D94E4F4509B47569140D7780AA57A08A7DE84F0ECE21C2F546B67A26EE3C0E97A646D1461
$X \leftarrow \mu(X, 0xF0)$	0x0018AE2752F96CF9244BB74B6657DCAB87026D42E80DD6FA870B8BE294E4F45064B8A96E40D7780A5A85F758DE84F0EC1DE3D0AB6B67A26E1C3F1685646D1461
$X \leftarrow$	0x44BF551CC2C0071D3C137B80820881EE06F7FA4E6D3A6850FA88957A786794A9084B97D716EA094F17DD9739110BD40D2A0B047321239FEAF92FE84FEA804558
$X \leftarrow \mu(X, 0x0F)$	0x44BF551C3D3FF8E23C137B807DF77E1106F7FA4E92C597AFFA88957A87986B56084B97D7E915F6B017DD9739EEF42BF22A0B0473DEDC6015F92FE84F157FBAA7
$X \leftarrow \mu(X, 0x0F)$	0x44BF551CC2C0071D3C137B80820881EE06F7FA4E6D3A6850FA88957A786794A9084B97D716EA094F17DD9739110BD40D2A0B047321239FEAF92FE84FEA804558
$X \leftarrow \mu(X, 0xF0)$	0xBB40AAE3C2C0071DC3EC847F820881EEF90805B16D3A685005776A85786794A9F7B4682816EA094FE82268C6110BD40DD5F4FB8C21239FEA06D017B0EA804558
$X \leftarrow \mu(X, 0xF0)$	0x44BF551CC2C0071D3C137B80820881EE06F7FA4E6D3A6850FA88957A786794A9084B97D716EA094F17DD9739110BD40D2A0B047321239FEAF92FE84FEA804558
$X \leftarrow$	0xC48842ECE0B10CCC1BA0A0A88BCEF93CB28C7D6428B205CA7C83B0DAF102E6FAE5ACCB89E941642EE912009B4E2CED0EA96418B3C6808DE1E2189E0D87B1739D
$X \leftarrow \mu(X, 0x0F)$	0xC48842EC1F4EF3331BA0A0A8743106C3B28C7D64D74DFA357C83B0DA0EFD1905E5ACCB8916BE9BD1E912009BB1D312F1A96418B3397F721EE2189E0D784E8C62
$X \leftarrow \mu(X, 0x0F)$	0xC48842ECE0B10CCC1BA0A0A88BCEF93CB28C7D6428B205CA7C83B0DAF102E6FAE5ACCB89E941642EE912009B4E2CED0EA96418B3C6808DE1E2189E0D87B1739D
$X \leftarrow \mu(X, 0xF0)$	0x3B77BD13E0B10CCCE45F5F578BCEF93C4D73829B28B205CA837C4F25F102E6FA1A533476E941642E16EDFF644E2CED0E569BE74CC6808DE11DE761F287B1739D
$X \leftarrow \mu(X, 0xF0)$	0xC48842ECE0B10CCC1BA0A0A88BCEF93CB28C7D6428B205CA7C83B0DAF102E6FAE5ACCB89E941642EE912009B4E2CED0EA96418B3C6808DE1E2189E0D87B1739D
$X \leftarrow$	0x958C5445011C6CD7565567262EA8E5125FD267809313E9E3982A38C6814D2180B937AC17D46C984F66D04EDC5EBA8447103CC66DB6E10CF7AB7B809EB325E059
$X \leftarrow \mu(X, 0x0F)$	0x958C5445FEE3932856556726D1571AED5FD267806CEC161C982A38C67EB2DE7FB937AC172B9367B066D04EDCA1457BB8103CC66D491EF308AB7B809E4CDA1FA6
$X \leftarrow \mu(X, 0x0F)$	0x958C5445011C6CD7565567262EA8E5125FD267809313E9E3982A38C6814D2180B937AC17D46C984F66D04EDC5EBA8447103CC66DB6E10CF7AB7B809EB325E059
$X \leftarrow \mu(X, 0xF0)$	0x6A73ABBA011C6CD7A9AA98D92EA8E512A02D987F9313E9E367D5C739814D218046C853E8D46C984F992FB1235EBA8447EFC33992B6E10CF754847F61B325E059
$X \leftarrow \mu(X, 0xF0)$	0x958C5445011C6CD7565567262EA8E5125FD267809313E9E3982A38C6814D2180B937AC17D46C984F66D04EDC5EBA8447103CC66DB6E10CF7AB7B809EB325E059
$X \leftarrow$	0xEB4032057154F3E0D8A0C749F9AAD8C0AFFB8B571F7FE0AD431500A698006BCCD6559B24F31E0180C44C60689DEC2364DAF2CBB2753FD955089A27C8ABFAEC52
$X \leftarrow \mu(X, 0x0F)$	0xEB4032058EAB0C1FD8A0C7490655273FAFFB8B57E0801F52431500A667FF9433D6559B240CE1FE7FC44C60686213DC9BDAF2CBB28AC026AA089A27C8540513AD
$X \leftarrow \mu(X, 0x0F)$	0xEB4032057154F3E0D8A0C749F9AAD8C0AFFB8B571F7FE0AD431500A698006BCCD6559B24F31E0180C44C60689DEC2364DAF2CBB2753FD955089A27C8ABFAEC52
$X \leftarrow \mu(X, 0xF0)$	0x14BFCDFA7154F3E0275F38B6F9AAD8C0500474A81F7FE0ADBCEAFF5998006BCC29AA64DBF31E01803BB39F979DEC2364250D344D753FD955F765D837ABFAEC52
$X \leftarrow \mu(X, 0xF0)$	0xEB4032057154F3E0D8A0C749F9AAD8C0AFFB8B571F7FE0AD431500A698006BCCD6559B24F31E0180C44C60689DEC2364DAF2CBB2753FD955089A27C8ABFAEC52

Tabla 4.12: Vectores de prueba para *mu*

Operación	Vector
$X \leftarrow$	0xE7370249C2B061691D8F768569A8E62AA5AFA32DFA06067C1022ACAF17930592D145DB750D7B434D57A01DCBEA6399B4785E979931324CC4019194DFA4F1B1C8
$X \leftarrow \lambda(X, 0x0F)$	0xEE3E0443C4B062631B8F7C8A63A1EC25AAAFA62BF50C0C791024A9AF1E960A94D24AD77A0B77464B5EA01BC7E56693B8715D9E93323449C8029298DFA8F2B2C1
$X \leftarrow \lambda(X, 0x0F)$	0xED3D0846C8B06466178F798566A2E92AA5AFAC27FA0909731028A3AF1D9C0598D445DE75077E4C475DA017CEEA6C96B1725B9D96343843C1049491DFA1F4B4C2
$X \leftarrow \lambda(X, 0x0F)$	0xEB3B014CC1B0686C1E8F738A6CA4E325AAAFA92EF50303761021A6AF1B990A91D84ADD7A0E7D494E5BA01ECDE5699CB274579B9C383146C2089892DFA2F8B8C4
$X \leftarrow \lambda(X, 0x0F)$	0xE7370249C2B061691D8F768569A8E62AA5AFA32DFA06067C1022ACAF17930592D145DB750D7B434D57A01DCBEA6399B4785E979931324CC4019194DFA4F1B1C8
$X \leftarrow \lambda(X, 0xF0)$	0xD76702899270C1C92D1FE615C958D64A555F534DFA0606EC20425C5F27330532B185BBE50DEB838DA7502D9BDAC33974E8AE373961628C94013134BF54F17198
$X \leftarrow \lambda(X, 0xF0)$	0xB7C7021932E091994D2FD62599A8B68AA5AFA38DFA0606DC4082ACAF4763056271157BD50DDB131D57A04D3BBA9369E4D85E6769C1C21C340161647FA4F1E138
$X \leftarrow \lambda(X, 0xF0)$	0x7797022962D031398D4FB6453958761A555F531DFA0606BC80125C5F87C305C2E125EBB50DBB232DA7508D6B7A33C9D4B8AEC7C991922C6401C1C4EF54F1D168
$X \leftarrow \lambda(X, 0xF0)$	0xE7370249C2B061691D8F768569A8E62AA5AFA32DFA06067C1022ACAF17930592D145DB750D7B434D57A01DCBEA6399B4785E979931324CC4019194DFA4F1B1C8
$X \leftarrow$	0x9E73264EE5580E9A2CFEB68E9543FC50C0621290ADEB43D012AF467DCDADB305A88AD9F6368B4288C2C0CFCA64A55D357D9226EA997A7DBDDCD8C8F1BEFA6825
$X \leftarrow \lambda(X, 0x0F)$	0x9D762C4DEA510D9529FDBC8D9A46F950C0641490ABE746D014AF4C7BCBABB60AA185D3FC3C874481C4C0CFC568AA5B3A7B942CE593757BBBD9D1C1F2BDF5612A
$X \leftarrow \lambda(X, 0x0F)$	0x9B7C294BE5520B9A23FBB98B954CF350C0681890A7EE4CD018AF4977C7A7BC05A28AD6F9398E4882C8C0CFCA61A55735779829EA967A77B7D3D2C2F4BBFA6225
$X \leftarrow \lambda(X, 0x0F)$	0x97792347EA54079526F7B3879A49F650C0611190AEED49D011AF437ECEAEB90AA485DCF3338D4184C1C0CFC562AA5E3A7E9123E59C757EBED6D4C4F8B7F5642A
$X \leftarrow \lambda(X, 0x0F)$	0x9E73264EE5580E9A2CFEB68E9543FC50C0621290ADEB43D012AF467DCDADB305A88AD9F6368B4288C2C0CFCA64A55D357D9226EA997A7DBDDCD8C8F1BEFA6825
$X \leftarrow \lambda(X, 0xF0)$	0x3EE3468ED5A80E3A4CFE761E3583FCA090C222305DDB83B0225F86ED9D5D7305581AB9F6661B821892909F9AC455AD65ED3246DA39EAED7DBCB898F17EFAC845
$X \leftarrow \lambda(X, 0xF0)$	0x6ED3861EB5580E6A8CFEE62E6513FC5030924260ADBB137042AF16DD3DADE305A82A79F6C62B122832303F3A94A55DC5DD6286BA69DADDED7C7838F1EEFA9885
$X \leftarrow \lambda(X, 0xF0)$	0xCEB3162E75A80ECA1CFED64EC523FCA0603282C05D7B23E0825F26BD6D5DD305584AE9F6964B224862606F6A3455AD95BDC2167AC9BABDDDECE868F1DEFA3815
$X \leftarrow \lambda(X, 0xF0)$	0x9E73264EE5580E9A2CFEB68E9543FC50C0621290ADEB43D012AF467DCDADB305A88AD9F6368B4288C2C0CFCA64A55D357D9226EA997A7DBDDCD8C8F1BEFA6825
$X \leftarrow$	0x5327153A110CAC3916B8AA0679206F88CE42EE19401209D554EF6D5E0C5A84C59B64DB9ECDA33433D0AB2414D71A3A223E93ECA556E037693E890092380F9B0B
$X \leftarrow \lambda(X, 0x0F)$	0x562E1A351209A9331CB1A50C73206F81CD44ED13401403DA58EF6B5D095588CA9768D79DCBA63836D0A72818DE1535243D96E9AA5CE03E633D830094310F9707
$X \leftarrow \lambda(X, 0x0F)$	0x5C2D153A1403A33619B2AA0976206F82CB48EB16401806D551EF675B035A81C59E61DE9BC7AC313CD0AE2111DD1A3A283B9CE3A559E03D663B860098320F9E0E
$X \leftarrow \lambda(X, 0x0F)$	0x592B1A351806A63C13B4A5037C206F84C741E71C40110CDA52EF6E57065582CA9D62DD97CEA93239D0AD2212DB1535213799E6AA53E03B6C378C0091340F9D0D
$X \leftarrow \lambda(X, 0x0F)$	0x5327153A110CAC3916B8AA0679206F88CE42EE19401209D554EF6D5E0C5A84C59B64DB9ECDA33433D0AB2414D71A3A223E93ECA556E037693E890092380F9B0B
$X \leftarrow \lambda(X, 0xF0)$	0xA347256A210C5C6926785A06E940CF189E82DE29802209B5A4DFCDAE0CAA14953BC4BB3E9D536463B05B4424B72A6A426E33DC55A6D067C96E190032680F3B0B
$X \leftarrow \lambda(X, 0xF0)$	0x538745CA410CACC946E8AA06D9809F283E12BE491042097554BF9D5E0C5A24356B947B6E3DA3C4C370AB8444774ACA82CE63BCA556B0C799CE290062C80F6B0B
$X \leftarrow \lambda(X, 0xF0)$	0xA317859A810C5C9986D85A06B9103F486E227E89208209E5A47F3DAE0CAA4465CB34EBCE6D539493E05B1484E78A9A129EC37C55A67097399E4900C2980FCB0B
$X \leftarrow \lambda(X, 0xF0)$	0x5327153A110CAC3916B8AA0679206F88CE42EE19401209D554EF6D5E0C5A84C59B64DB9ECDA33433D0AB2414D71A3A223E93ECA556E037693E890092380F9B0B
$X \leftarrow$	0x152BA4C7D98181288246B5044FB431D341F8B55DF9C6D048FBB97B6B7009B1496F5C4F9A8769A74BFFFFAFA0E11FC27F1AFBB77DA6FABE2A27DA7628098C1C8B
$X \leftarrow \lambda(X, 0x0F)$	0x1A27A8CED3828221844CBA084FB832D642F1BA5BF3CCD041F7B377677003B2436F594F958E63AE47FFFFAFA0E21FC47F15F7BE7BACF5BD252ED57C2103891987
$X \leftarrow \lambda(X, 0x0F)$	0x152EA1CDD68484228849B5014FB134DC44F2B557F6C9D042FEB67E6E7006B4466F534F9A8D66AD4EFFFFAFA0E41FC87F1AFEBD77A9FABB2A2DDA79220683138E
$X \leftarrow \lambda(X, 0x0F)$	0x1A2DA2CBDC8888248143BA024FB238D948F4BA5EFCC3D044FDBC7D6D700CB84C6F564F958B6CAB4DFFFFAFA0E81FC17F15FDBB7EA3F5B7252BD573240C86168D
$X \leftarrow \lambda(X, 0x0F)$	0x152BA4C7D98181288246B5044FB431D341F8B55DF9C6D048FBB97B6B7009B1496F5C4F9A8769A74BFFFFAFA0E11FC27F1AFBB77DA6FABE2A27DA7628098C1C8B
$X \leftarrow \lambda(X, 0xF0)$	0x254B5497B9111148128675048F7461B381F875ADF996B088FB79EBCBE0097189CFAC8F3A17C9578BFFFF5F50D12F92EF2AFB77ED56FA7E4A47BAE648091C2C1B
$X \leftarrow \lambda(X, 0xF0)$	0x458BA43779212188221 6E5041FE4C17311F8E55DF9367018FBE9DB9BD009E1199F5C1F6A2799A71BFFFFAFA0B14F32DF4AFBE7DDA6FAEE8A877AD688092C4C2B
$X \leftarrow \lambda(X, 0xF0)$	0x851B5467E94141184226D5042FD491E321F8D5ADF966E028FBD9BB3BB009D1293FAC2FCA4739572BFFFF5F50718F62BF8AFBD7BD56FADE1A17EAB618094C8C4B
$X \leftarrow \lambda(X, 0xF0)$	0x152BA4C7D98181288246B5044FB431D341F8B55DF9C6D048FBB97B6B7009B1496F5C4F9A8769A74BFFFFAFA0E11FC27F1AFBB77DA6FABE2A27DA7628098C1C8B
$X \leftarrow$	0x0BD9D35C7C376F18CC185530A23BBEB8285104F4FE6224B11480612F800569DC0CF5C8EB3E2F5D4C9D34FE02CEB5AD45E5281FE7FF69A0A08D28AA7B88891DC3
$X \leftarrow \lambda(X, 0x0F)$	0x07D3D659793E6F11C9115A30A437BDB1215208F8FD6428B21880622F800A63D909FAC1E73D2F5B499B38FD04CDBAAB4AEA211FEEFF63A0A08B21A57781831BC6
$X \leftarrow \lambda(X, 0x0F)$	0x0ED6DC53733D6F12C3125530A83EBBB2225401F1FB6821B41180642F800566D303F5C2EE3B2F57439731FB08CBB5A745E5221FEDFF66A0A08722AA7E828617CC
$X \leftarrow \lambda(X, 0x0F)$	0x0DDCD956763B6F14C6145A30A13DB7B4245802F2F76122B81280682F800A6CD606FAC4ED372F5E469E32F701C7BAAE4AEA241FEBFF6CA0A08E24A57D848C1EC9
$X \leftarrow \lambda(X, 0x0F)$	0x0BD9D35C7C376F18CC185530A23BBEB8285104F4FE6224B11480612F800569DC0CF5C8EB3E2F5D4C9D34FE02CEB5AD45E5281FE7FF69A0A08D28AA7B88891DC3
$X \leftarrow \lambda(X, 0xF0)$	0x0BB9B3ACEC67CF289C28A560526B7E7848A104F4FEC24712410C14F1005C9BC0CF598DB6E4FAD8C3D64FE029E755D85D5482FD7FFC950501D485AEB18192D93
$X \leftarrow \lambda(X, 0xF0)$	0x0B79735CDCC79F483C4855C0A2CBEEE8885104F4FE9284E14420918F2005997C0CF538BBCE8F5D1C6DC4FE023EE5AD15B5884FB7FF99A0A02D88AADB28294D33
$X \leftarrow \lambda(X, 0xF0)$	0x0BE9E3ACBC973F886C88A590529BDED818A104F4FE3214D18440311F400539EC0CF5687B9E1FAD2CCD94FE026ED55D2575188F77FF3950504D185ABB48498D63
$X \leftarrow \lambda(X, 0xF0)$	0x0BD9D35C7C376F18CC185530A23BBEB8285104F4FE6224B11480612F800569DC0CF5C8EB3E2F5D4C9D34FE02CEB5AD45E5281FE7FF69A0A08D28AA7B88891DC3

Tabla 4.13: Vectores de prueba para *lambda*

Operación	Vector
$X \leftarrow$	0xE7370249C2B061691D8F768569A8E62AA5AFA32DFA06067C1022ACAF17930592D145DB750D7B434D57A01DCBEA6399B4785E979931324CC4019194DFA4F1B1C8
$X \leftarrow \varphi(X, 0x0F)$	0xE7370249B06169C21D8F7685A8E62A69A5AFA32D06067CFA1022ACAF93059217D145DB757B434D0D57A01DCB6399B4EA785E9799324CC431019194DFF1B1C8A4
$X \leftarrow \varphi(X, 0x0F)$	0xE73702496169C2B01D8F7685E62A69A8A5AFA32D067CFA061022ACAF05921793D145DB75434D0D7B57A01DCB99B4EA63785E97994CC43132019194DFB1C8A4F1
$X \leftarrow \varphi(X, 0x0F)$	0xE737024969C2B0611D8F76852A69A8E6A5AFA32D7CFA06061022ACAF92179305D145DB754D0D7B4357A01DCBB4EA6399785E9799C431324C019194DFC8A4F1B1
$X \leftarrow \varphi(X, 0x0F)$	0xE7370249C2B061691D8F768569A8E62AA5AFA32DFA06067C1022ACAF17930592D145DB750D7B434D57A01DCBEA6399B4785E979931324CC4019194DFA4F1B1C8
$X \leftarrow \varphi(X, 0xF0)$	0x370249E7C2B061698F76851D69A8E62AAFA32DA5FA06067C22ACAF101793059245DB75D10D7B434DA01DCB57EA6399B45E97997831324CC49194DF01A4F1B1C8
$X \leftarrow \varphi(X, 0xF0)$	0x0249E737C2B0616976851D8F69A8E62AA32DA5AFFA06067CACAF102217930592DB75D1450D7B434D1DCB57A0EA6399B49799785E31324CC494DF0191A4F1B1C8
$X \leftarrow \varphi(X, 0xF0)$	0x49E73702C2B06169851D8F7669A8E62A2DA5AFA3FA06067CAF1022AC1793059275D145DB0D7B434DCB57A01DEA6399B499785E9731324CC4DF019194A4F1B1C8
$X \leftarrow \varphi(X, 0xF0)$	0xE7370249C2B061691D8F768569A8E62AA5AFA32DFA06067C1022ACAF17930592D145DB750D7B434D57A01DCBEA6399B4785E979931324CC4019194DFA4F1B1C8
$X \leftarrow$	0x9E73264EE5580E9A2CFEB68E9543FC50C0621290ADEB43D012AF467DCDADB305A88AD9F6368B4288C2C0CFCA64A55D357D9226EA997A7DBDDCD8C8F1BEFA6825
$X \leftarrow \varphi(X, 0x0F)$	0x9E73264E580E9AE52CFEB68E43FC5095C0621290EB43D0AD12AF467DADB305CDA88AD9F68B428836C2C0CFCAA55D35647D9226EA7A7DBD99DCD8C8F1FA6825BE
$X \leftarrow \varphi(X, 0x0F)$	0x9E73264E0E9AE5582CFEB68EFC509543C062129043D0ADEB12AF467DB305CDADA88AD9F64288368BC2C0CFCA5D3564A57D9226EA7DBD997ADCD8C8F16825BEFA
$X \leftarrow \varphi(X, 0x0F)$	0x9E73264E9AE5580E2CFEB68E509543FCC0621290D0ADEB4312AF467D05CDADB3A88AD9F688368B42C2C0CFCA3564A55D7D9226EABD997A7DDCD8C8F125BEFA68
$X \leftarrow \varphi(X, 0x0F)$	0x9E73264EE5580E9A2CFEB68E9543FC50C0621290ADEB43D012AF467DCDADB305A88AD9F6368B4288C2C0CFCA64A55D357D9226EA997A7DBDDCD8C8F1BEFA6825
$X \leftarrow \varphi(X, 0xF0)$	0x73264E9EE5580E9AFEB68E2C9543FC50621290C0ADEB43D0AF467D12CDADB3058AD9F6A8368B4288C0CFCAC264A55D359226EA7D997A7DBDD8C8F1DCBEFA6825
$X \leftarrow \varphi(X, 0xF0)$	0x264E9E73E5580E9AB68E2CFE9543FC501290C062ADEB43D0467D12AFCDADB305D9F6A88A368B4288CFCAC2C064A55D3526EA7D92997A7DBDC8F1DCD8BEFA6825
$X \leftarrow \varphi(X, 0xF0)$	0x4E9E7326E5580E9A8E2CFEB69543FC5090C06212ADEB43D07D12AF46CDADB305F6A88AD9368B4288CAC2C0CF64A55D35EA7D9226997A7DBDF1DCD8C8BEFA6825
$X \leftarrow \varphi(X, 0xF0)$	0x9E73264EE5580E9A2CFEB68E9543FC50C0621290ADEB43D012AF467DCDADB305A88AD9F6368B4288C2C0CFCA64A55D357D9226EA997A7DBDDCD8C8F1BEFA6825
$X \leftarrow$	0x5327153A110CAC3916B8AA0679206F88CE42EE19401209D554EF6D5E0C5A84C59B64DB9ECDA33433D0AB2414D71A3A223E93ECA556E037693E890092380F9B0B
$X \leftarrow \varphi(X, 0x0F)$	0x5327153A0CAC391116B8AA06206F8879CE42EE191209D54054EF6D5E5A84C50C9B64DB9EA33433CDD0AB24141A3A22D73E93ECA5E03769563E8900920F9B0B38
$X \leftarrow \varphi(X, 0x0F)$	0x5327153AAC39110C16B8AA066F887920CE42EE1909D5401254EF6D5E84C50C5A9B64DB9E3433CDA3D0AB24143A22D71A3E93ECA5376956E03E8900929B0B380F
$X \leftarrow \varphi(X, 0x0F)$	0x5327153A39110CAC16B8AA068879206FCE42EE19D540120954EF6D5EC50C5A849B64DB9E33CDA334D0AB241422D71A3A3E93ECA56956E0373E8900920B380F9B
$X \leftarrow \varphi(X, 0x0F)$	0x5327153A110CAC3916B8AA0679206F88CE42EE19401209D554EF6D5E0C5A84C59B64DB9ECDA33433D0AB2414D71A3A223E93ECA556E037693E890092380F9B0B
$X \leftarrow \varphi(X, 0xF0)$	0x27153A53110CAC39B8AA061679206F8842EE19CE401209D5EF6D5E540C5A84C564DB9E9BCDA33433AB2414D0D71A3A2293ECA53E56E037698900923E380F9B0B
$X \leftarrow \varphi(X, 0xF0)$	0x153A5327110CAC39AA0616B879206F88EE19CE42401209D56D5E54EF0C5A84C5DB9E9B64CDA334332414D0ABD71A3A22ECA53E9356E0376900923E89380F9B0B
$X \leftarrow \varphi(X, 0xF0)$	0x3A532715110CAC390616B8AA79206F8819CE42EE401209D55E54EF6D0C5A84C59E9B64DBCDA3343314D0AB24D71A3A22A53E93EC56E03769923E8900380F9B0B
$X \leftarrow \varphi(X, 0xF0)$	0x5327153A110CAC3916B8AA0679206F88CE42EE19401209D554EF6D5E0C5A84C59B64DB9ECDA33433D0AB2414D71A3A223E93ECA556E037693E890092380F9B0B
$X \leftarrow$	0x152BA4C7D98181288246B5044FB431D341F8B55DF9C6D048FBB97B6B7009B1496F5C4F9A8769A74BFFFFAFA0E11FC27F1AFBB77DA6FABE2A27DA7628098C1C8B
$X \leftarrow \varphi(X, 0x0F)$	0x152BA4C7818128D98246B504B431D34F41F8B55DC6D048F9FBB97B6B09B149706F5C4F9A69A74B87FFFFAFA01FC27FE11AFBB77DFABE2AA627DA76288C1C8B09
$X \leftarrow \varphi(X, 0x0F)$	0x152BA4C78128D9818246B50431D34FB441F8B55DD048F9C6FBB97B6BB14970096F5C4F9AA74B8769FFFFAFA0C27FE11F1AFBB77DBE2AA6FA27DA76281C8B098C
$X \leftarrow \varphi(X, 0x0F)$	0x152BA4C728D981818246B504D34FB43141F8B55D48F9C6D0FBB97B6B497009B16F5C4F9A4B8769A7FFFFAFA07FE11FC21AFBB77D2AA6FABE27DA76288B098C1C
$X \leftarrow \varphi(X, 0x0F)$	0x152BA4C7D98181288246B5044FB431D341F8B55DF9C6D048FBB97B6B7009B1496F5C4F9A8769A74BFFFFAFA0E11FC27F1AFBB77DA6FABE2A27DA7628098C1C8B
$X \leftarrow \varphi(X, 0xF0)$	0x2BA4C715D981812846B504824FB431D3F8B55D41F9C6D048B97B6BFB7009B1495C4F9A6F8769A74BFFAFA0FFE11FC27FFBB77D1AA6FABE2ADA762827098C1C8B
$X \leftarrow \varphi(X, 0xF0)$	0xA4C7152BD9818128B50482464FB431D3B55D41F8F9C6D0487B6BFBB97009B1494F9A6F5C8769A74BAFA0FFFFE11FC27FB77D1AFBA6FABE2A762827DA098C1C8B
$X \leftarrow \varphi(X, 0xF0)$	0xC7152BA4D9818128048246B54FB431D35D41F8B5F9C6D0486BFBB97B7009B1499A6F5C4F8769A74BA0FFFFAFE11FC27F7D1AFBB7A6FABE2A2827DA76098C1C8B
$X \leftarrow \varphi(X, 0xF0)$	0x152BA4C7D98181288246B5044FB431D341F8B55DF9C6D048FBB97B6B7009B1496F5C4F9A8769A74BFFFFAFA0E11FC27F1AFBB77DA6FABE2A27DA7628098C1C8B
$X \leftarrow$	0x0BD9D35C7C376F18CC185530A23BBEB8285104F4FE6224B11480612F800569DC0CF5C8EB3E2F5D4C9D34FE02CEB5AD45E5281FE7FF69A0A08D28AA7B88891DC3
$X \leftarrow \varphi(X, 0x0F)$	0x0BD9D35C376F187CCC1855303BBEB8A2285104F46224B1FE1480612F0569DC800CF5C8EB2F5D4C3E9D34FE02B5AD45CEE5281FE769A0A0FF8D28AA7B891DC388
$X \leftarrow \varphi(X, 0x0F)$	0x0BD9D35C6F187C37CC185530BEB8A23B285104F424B1FE621480612F69DC80050CF5C8EB5D4C3E2F9D34FE02AD45CEB5E5281FE7A0A0FF698D28AA7B1DC38889
$X \leftarrow \varphi(X, 0x0F)$	0x0BD9D35C187C376FCC185530B8A23BBE285104F4B1FE62241480612FDC8005690CF5C8EB4C3E2F5D9D34FE0245CEB5ADE5281FE7A0FF69A08D28AA7BC388891D
$X \leftarrow \varphi(X, 0x0F)$	0x0BD9D35C7C376F18CC185530A23BBEB8285104F4FE6224B11480612F800569DC0CF5C8EB3E2F5D4C9D34FE02CEB5AD45E5281FE7FF69A0A08D28AA7B88891DC3
$X \leftarrow \varphi(X, 0xF0)$	0xD9D35C0B7C376F18185530CCA23BBEB85104F428FE6224B180612F14800569DCF5C8EB0C3E2F5D4C34FE029DCEB5AD45281FE7E5FF69A0A028AA7B8D88891DC3
$X \leftarrow \varphi(X, 0xF0)$	0xD35C0BD97C376F185530CC18A23BBEB804F42851FE6224B1612F1480800569DCC8EB0CF53E2F5D4CFE029D34CEB5AD451FE7E528FF69A0A0AA7B8D2888891DC3
$X \leftarrow \varphi(X, 0xF0)$	0x5C0BD9D37C376F1830CC1855A23BBEB8F4285104FE6224B12F148061800569DCEB0CF5C83E2F5D4C029D34FECEB5AD45E7E5281FFF69A0A07B8D28AA88891DC3
$X \leftarrow \varphi(X, 0xF0)$	0x0BD9D35C7C376F18CC185530A23BBEB8285104F4FE6224B11480612F800569DC0CF5C8EB3E2F5D4C9D34FE02CEB5AD45E5281FE7FF69A0A08D28AA7B88891DC3

Tabla 4.14: Vectores de prueba para *fi*

Capítulo 5

Discusión y Conclusiones

En este trabajo se pudo construir un algoritmo de cifrado que cumple con los objetivos de limitar el alcance de un ataque de fuerza bruta, ya que logró crear la posibilidad de controlar (hasta cierto punto) la cantidad aproximada de operaciones requeridas para el cifrado y descifrado. Lo que le da al usuario el control del tiempo aproximado que se requerirá para poder descifrar su información de acuerdo con sus necesidades.

Se logró un nivel de personalización para un sistema de cifrado que hasta donde tengo conocimiento no se había dado, ya que el usuario con sólo cambiar parámetros o el archivo llave generará un procedimiento de cifrado diferente, sin contar con el hecho de que puede ampliar el sistema agregando nuevos operadores de permutación o sistemas de cifrado.

En el caso de la implementación de extensiones embebiendo otros sistemas de cifrado, de una manera subjetiva se puede decir que es sencilla.

Como todo sistema de cifrado simétrico, no resuelve el problema del intercambio de llaves, pero el uso de sistemas asimétricos y buenas políticas de seguridad podría atenuar este problema.

El sistema propuesto está lejos de ser perfecto, de hecho la mayor parte de su fortaleza se puede adjudicar al usuario final, lo que lo hace falible, pero también le permitirá jugar un rol de mayor importancia en la seguridad criptográfica de su organización, convirtiéndolo en una especie de diseñador criptográfico.

La fortaleza de este sistema dependerá de diversos factores como siempre ha sido y seguirá siendo *"la elección de la llave"* k, pero también de la selección o diseño del *archivo llave* f_{k}, además de la personalización adicional que se pueda aportar, ya sea modificando la configuración, creando nuevas permutaciones y/o agregando existentes (AES, Twofish, RSA, $\cdots$, etc[1]), ya que este cifrado es fácilmente extensible como se vio en la *sección 3.4*[2], en este contexto el rol de un diseñador criptográfico, es fundamental, ya que se le otorgan muchas

[1] La recomendación es aprovechar el GFBPP, para pasar como llave el máximo número de bytes permitidos por el cifrado.

[2] Recordemos que en este caso no se extendió porque lo que se pretendió es analizar el algoritmo con permutaciones simples similares a las de un cubo de Rubik y se consideró que apoyarse en transformaciones más complejas, como lo son los algoritmos de cifrado, pondría en duda la fortaleza del algoritmo en general.

libertades.

Este sistema de doble llave k y f_{k} está diseñado para mantener cierto grado de seguridad, siempre y cuando al menos una de ellas sea desconocida, siendo nuevamente responsabilidad del usuario final y la organización el resguardo de las mismas. Algo interesante es el hecho de que el archivo llave f_{k}, puede generarse a partir de este GFBPP o cualquier otro, pero sí se usa como base el aquí propuesto en la *sección 3.2*, cada parámetro necesario, sería una llave en potencia, lo que lo convertiría en un sistema multi-llave y esto podría permitir nuevas aplicaciones.

En el caso del GFBPP, se concluye que es una herramienta que puede simplificar el diseño de generadores de flujos de bits pseudo-aleatorios, pero aún tiene mucho por mejorar antes de poder usarse, sí se deseará se podría utilizar otro tipo de generador de flujo pseudo-aleatorio como por ejemplo: un hash criptográfico usado en forma recursiva.

Como ya se dijo en la *sección 3.2* la visión simplista de promediar los resultados de las pruebas, sí bien facilita la elección automatizada de reglas, definitivamente no es infalible, pero desde mi muy particular punto de vista es suficiente, ya que la llave k define los estados iniciales y se puede establecer (sí se modifico **config.gfParam.initEvolutions**) la generación inicial desde la cual se comenzarán a producir los flujos de bits.

5.1. Trabajo futuro

Algo que quedaría pendiente en este trabajo es la optimización de algoritmos, lo cual no fue abordado, principalmente para que estos fuesen lo más claros y universales posibles, pero también para evitar que el debate se centrara en la calidad de dicha optimización y/o escenarios donde no serían óptimos.

Para el caso de los AC's como generadores de flujos pseudo-aleatorios, se han hecho múltiples propuestas para mejorar la selección de reglas, pero sería interesante aplicarlas a los AC's con reglas dinámicas como el GFBPP, expandirlo a AC's multi-dimensionales, además de utilizar técnicas de construcción similares a las usadas para interpretar el archivo llave f_{k}.

Finalmente, el GFBPP de acuerdo con los resultados obtenidos, tiene mucho por mejorar tanto en construcción, discriminación rápida de las mejores reglas y la determinación de la misma para la generación actual (véase el algoritmo 3.2) para obtener mejores resultados; además de la obvia limitación de los AC's cuando alcanzan el estado inicial, en la misma generación relativa del ciclo de reglas, ya que a partir de este momento se repetirá el ciclo (comportamiento no es deseado), lo que requerirá de perturbaciones irregulares para evitarse. También hay que considerar la posibilidad de replantearlo integrándolo con otras bases teóricas que puedan darle un comportamiento más caótico o con más aleatoriedad.

Apéndice A

Plantillas de cubos

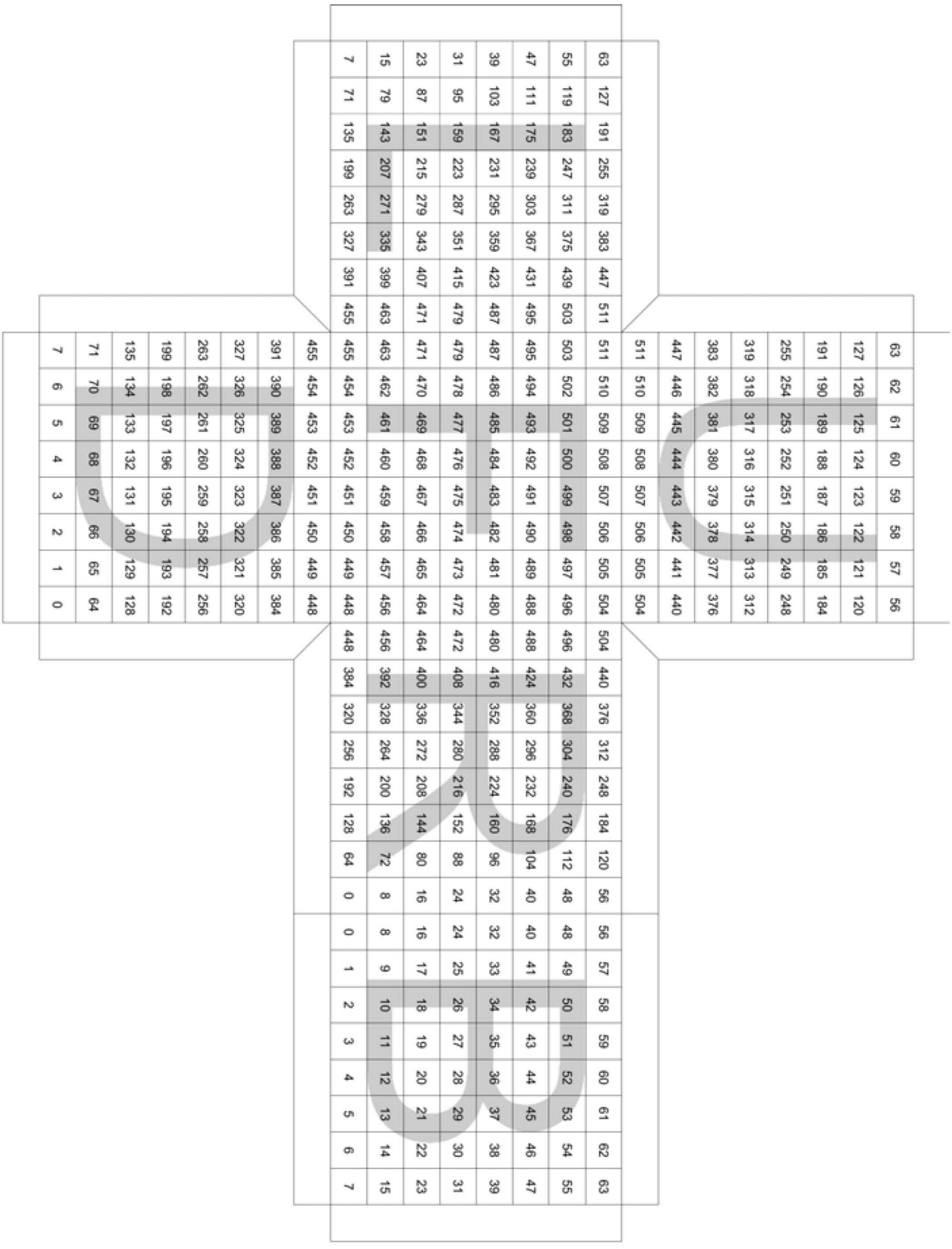


Figura A.1: Cubo recortable con posiciones

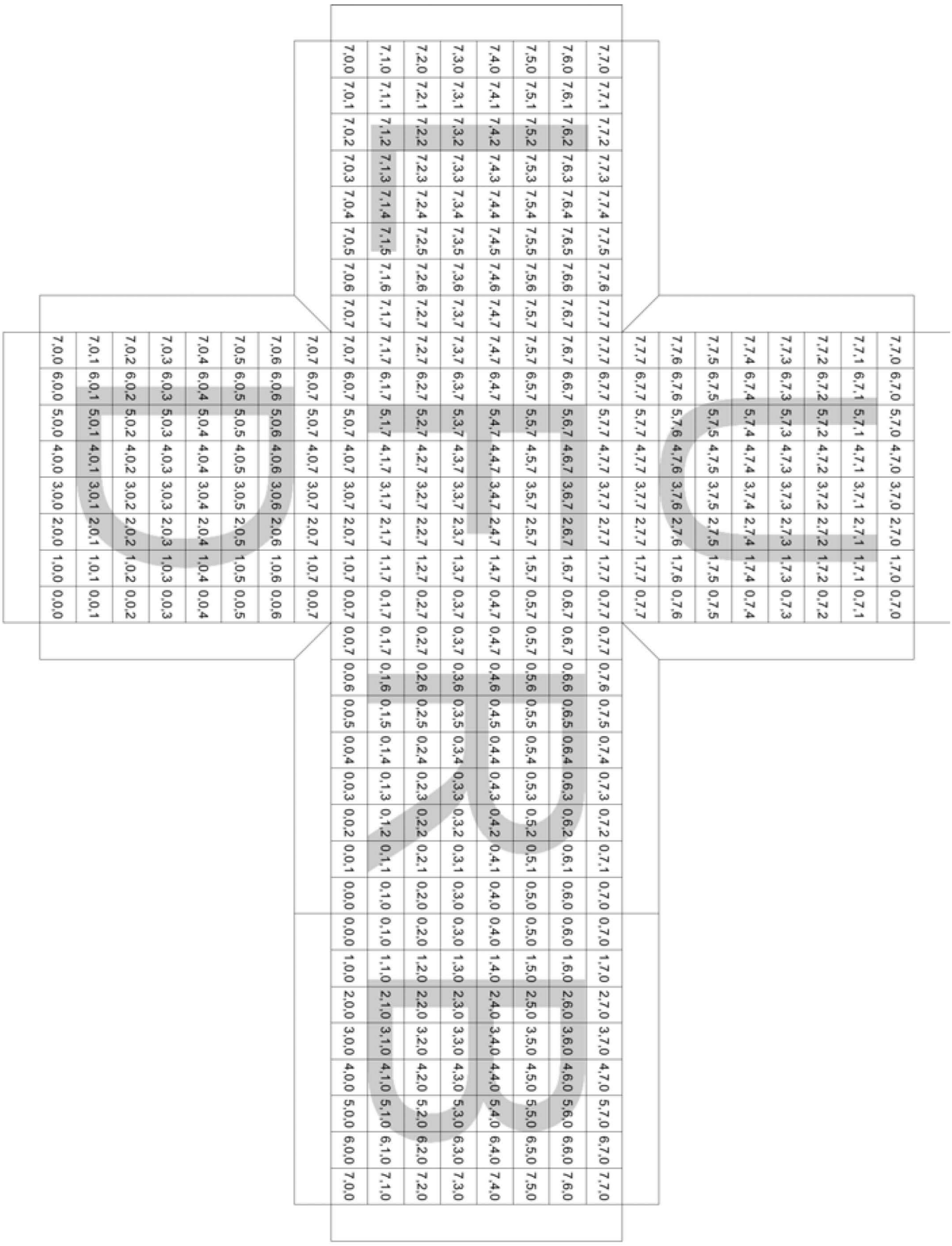


Figura A.2: Cubo recortable con coordenadas

Bibliografía

[1] Alexandroff, P. *An Introduction to the Theory of Groups*. Dover Publications, 1959.

[2] Benkiniouar, M. *Cellular automata for Cryptography*. IEEE, 2004.

[3] Britz, M. T. *Computer Forensics and Cyber Crime: An Introduction*. Pearson, 2013.

[4] Dubreil, P. *Teoría de grupos: Curso de iniciación*. Reverté, 1975.

[5] Flatley, J. L. Playstation 3 used to hack ssl, xbox used to play boogie bunnies. http://www.engadget.com/2008/12/30/hackers-playstation-3-make-ssl-much-less-secure. Recuperado el 16 de marzo de 2015.

[6] Menezes, A. *Handbook of Applied Cryptography*. CRC Press, 1996.

[7] Ortega Arjona, J. Programación concurrente, Noviembre 1996.

[8] Ortega Arjona, J. *Patterns for Parallel Software Design*. Wiley Software Patterns. Wiley, 2010.

[9] Rose, J. S. *A Course on Group Theory*. Dover Publications, 1978.

[10] Rukhin, A. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. NIST, 2010.

[11] Seredynski, F. *Secret Key Cryptography with Cellular Automata*. IEEE, 2003.

[12] Shannon, C. E. *A Mathematical Theory of Communication*. The Bell System Technical Journal, 1948.

[13] Shannon, C. E. *Communication Theory of Secrecy Systems*. Bell System Technical Journal, 1949.

[14] SPIEGEL. Prying eyes: Inside the nsa's war on internet security. http://www.spiegel.de/international/germany/inside-the-nsa-s-war-on-internet-security-a-1010361.html. Recuperado el 16 de marzo de 2015.

[15] Szczys, M. 25 gpus brute force 348 billion hashes per second to crack your passwords. http://hackaday.com/2012/12/06/25-gpus-brute-force-348-billion-hashes-per-second-to-crack-your-passwords. Recuperado el 16 de marzo de 2015.

[16] Wikipedia. Cifrado vernam. http://es.wikipedia.org/wiki/Cifrado_Vernam. Recuperado el 16 de marzo de 2015.

[17] Wikipedia. Ley de amdahl. http://es.wikipedia.org/wiki/Ley_de_Amdahl. Recuperado el 16 de marzo de 2015.

[18] Wikipedia. Sistema de lindenmayer. http://es.wikipedia.org/wiki/Sistema-L. Recuperado el 16 de marzo de 2015.

[19] Wolfram, S. Cellular automaton. http://mathworld.wolfram.com/CellularAutomaton.html. Recuperado el 16 de marzo de 2015.

[20] Wolfram, S. Elementary cellular automaton. http://mathworld.wolfram.com/ElementaryCellularAutomaton.html. Recuperado el 16 de marzo de 2015.

[21] Wolfram, S. *Theory and Application of Cellular Automata*. World Scientific, 1996.

[22] Zomaya, A. Y. *Secret Key Cryptography with Cellular Automata*. IEEE, 2004.

Listado de Acrónimos

AC Autómata Celular. VII, XI, 4, 7, 11–13, 15–17, 47, 64

ASIC Circuitos Integrados de Aplicación Específica. 10

GFBPP Generador de Flujos de Bits Pseudo-aleatorios Personalizable. XI, 4, 5, 7, 15, 16, 18, 20, 22, 23, 47, 49, 63, 64

GPU Graphics Processing Unit. 4

NIST National Institute of Standars and Technology. 47

UP Unidad de Procesamiento. 10

Listado de símbolos

$\wedge$ AND binario. 22, 26, 32–46

$\vee$ OR binario. 18, 22, 25, 26, 33–46

B Cara posterior sus variaciones según su posición son $\{B, \mathrm{B}, \mathrm{B}, \mathrm{B}\}$. 30

D Cara inferior sus variaciones según su posición son $\{D, \mathrm{D}, \mathrm{D}, \mathrm{D}\}$. 30

F Cara frontal sus variaciones según su posición son $\{F, \mathrm{F}, \mathrm{F}, \mathrm{F}\}$. 30

$f_{\mathbf{k}}$ Archivo llave. 3, 4, 18–21, 23, 63, 64

G Conjunto de generaciones del AC. 12

i Eje de coordenadas x. 31

j Eje de coordenadas y. 31

k Eje de coordenadas z. 31

$\mathbf{k}_x$ Llave del emisor. 8

k Clave, llave o contraseña. 4, 8, 9, 19, 20, 23, 63, 64

$\mathbf{k}_y$ Llave del receptor. 8

L Cara izquierda sus variaciones según su posición son $\{L, \mathrm{L}, \mathrm{L}, \mathrm{L}\}$. 30

m_t Mensaje transformado. 4, 8

m Mensaje. 4, 8

md Detalles del mensaje. 18, 20

P Punto definido en tres dimensiones. 31

Q Punto definido en dos dimensiones. 31

R Cara derecha sus variaciones según su posición son $\{R, \mathrm{R}, \mathrm{R}, \mathrm{R}\}$. 30

S Conjunto de células. 12

$s_{\mathbf{k}}$ Script llave. 3, 4, 19, 21, 23

$t^{-1}()$ Función de transformación inversa o descifrado. 8

$t_{s_{\mathbf{k}}}()$ Función de transformación, sobre el script llave. 4

$t()$ Función de transformación o cifrado. 3, 4, 8

U Cara superior sus variaciones según su posición son $\{U, \supset, \cap, \subset\}$. 30

V Conjunto de símbolos o alfabeto del AC. 12

υ ípsilon: Permutación corrimiento vertical circular reptante. VI, 15, 46, 55

Γ Gamma: Conjunto de condiciones de frontera del AC. 12

Δ Delta: Estado inicial del conjunto S. 12, 47

X Chi: Reintegración de segmento en cubo. 33

$\Theta()$ Theta: Es la configuración de la vecindad para cada $s \in S$. 12

Φ Phi: Es el conjunto de reglas para el conjunto S. 12

Ψ Psi: SubConjunto de reglas. 16, 47, 48

α alpha. Representación de cubo como un matriz de datos (8 x 8 x 8), o "**notación en bits**".. 31

β beta: Representación de segmento de cubo como un arreglo de datos de longitud 8, o "**notación en Bytes**". 31

γ gamma: Condición de frontera del AC. 12

ζ Dseta: Negación de columnas. VI, 15, 37, 58

χ chi: Extracción de segmento del cubo. 33

η eta: Permutación girar cubo horizontalmente. VI, 15, 36, 57

θ theta: Es la configuración de la vecindad para cada $s \in S$. 16

ι iota: Permutación corrimiento vertical circular independiente. VI, 15, 41, 50

λ lambda: Permutación de columnas. VI, 15, 32, 39, 60

μ mu: Negación de filas. VI, 15, 38, 59

ν ni: Permutación girar cubo verticalmente. VI, 15, 35, 56

ξ xi: Permutación corrimiento horizontal circular reptante. VI, 15, 45, 54

φ Fi: Permutación de filas. VI, 15, 32, 40, 61

ρ rho: Permutación corrimiento horizontal circular independiente. VI, 15, 42, 51

ς varsigma: Permutación corrimiento horizontal circular zigzagueante. VI, 15, 44, 53

ω omega: Permutación corrimiento vertical circular zigzagueante. VI, 15, 43, 52

Glosario

Es un cifrado de flujo en el que el texto en claro se combina, mediante la operación XOR, con un flujo de datos aleatorio o pseudoaleatorio del mismo tamaño, para generar un texto cifrado. . Artículo en *Wikipedia*, *La enciclopedia libre*. Recuperado el 16 de marzo de 2015 <http://es.wikipedia.org/wiki/Cifrado_Vernam>.

C

cadena crítica Es la secuencia de precedencias y elementos terminales dependientes de recursos que evitan que un proyecto, al que se le dan recursos limitados, pueda ser completado en un tiempo menor. Artículo en *Wikipedia*, *La enciclopedia libre*. Recuperado el 16 de marzo de 2015 <http://es.wikipedia.org/wiki/Gesti%C3%B3n_de_Proyectos_por_Cadena_Cr%C3%ADtica>, p. 10.

cómputo forense Es la aplicación de técnicas científicas y analíticas especializadas a infraestructura tecnológica que permiten identificar, preservar, analizar y presentar datos que sean válidos dentro de un proceso legal. Dichas técnicas incluyen reconstruir el bien informático, examinar datos residuales, autenticar datos y explicar las características técnicas del uso aplicado a los datos y bienes informáticos. Artículo en *Wikipedia*, *La enciclopedia libre*. Recuperado el 16 de marzo de 2015 <http://es.wikipedia.org/wiki/C%C3%B3mputo_forense>, p. 9.

G

granularidad Es un indicador de la cantidad de procesamiento que cada unidad de procesamiento puede ejecutar independientemente, en relación a el tiempo que ocupa para intercambiar información con otra unidad de procesamiento[7], p. 10.

granularidad fina Medida cualitativa de procesamiento en la que se realizan pocas instrucciones entre las comunicaciones de las unidades de procesamiento. La relación de tiempo de procesamiento entre el tiempo de comunicación es baja[7], p. 10.

granularidad gruesa Medida cualitativa de procesamiento en la que dos o mas procesos dependientes con poca sincronización. La relación de tiempo de procesamiento entre el tiempo de comunicación es alta[7], p. 11.

L

Ley de Amdahl La mejora obtenida en el rendimiento de un sistema debido a la alteración de uno de sus componentes está limitada por la fracción de tiempo que se utiliza dicho componente, p. 3.

M

magic numbers En cómputo forense, son los patrones característicos de los diferentes formatos de archivos, estos pueden ser intencionales o no, pero su principal aplicación es identificar el tipo de archivo recuperado, pero del cual se desconoce su tipo. Veáse `<http://en.wikipedia.org/wiki/Magic_number_%28programming%29>`, p. 20.

mensaje en claro Mensaje sin cifrar, p. 8.

mensaje transformado Mensaje cifrado, p. 8.

P

patrón La solución a un problema de diseño en un contexto dado C. Alexander (1977). En el caso del patrón de software es la relación entre un problema(función) y una solución(forma) que se da en un contexto existente[8], p. 11.

procesamiento cooperativo Dos o mas procesos son cooperativos sí se comunican entre sí y requieren algún tipo de sincronización para evitar corromper sus datos[7], p. 10.

procesamiento disjunto Dos o mas procesos son disjuntos sí no se comunican entre sí[7], p. 10.

procesamiento distribuido ídem que proceso paralelo, pero en el que las unidades de procesamiento se encuentran diferentes equipos[7], p. 10.

procesamiento paralelo Conjunto de dos o más procesos secuenciales, que se ejecutan simultáneamente, cooperando entre sí, para lograr un objetivo común[7], p. 10.

proceso Manipulación del contenido de la memoria por acción del una unidad de procesamiento, p. 10.

R

ruta crítica O método de la ruta crítica o del camino crítico es un algoritmo utilizado para el cálculo de tiempos y plazos en la planificación de proyectos. Un proyecto puede tener varias rutas críticas paralelas. Artículo en *Wikipedia*, *La enciclopedia libre*. Recuperado el 16 de marzo de 2015 `<http://es.wikipedia.org/wiki/M%C3%A9todo_de_la_ruta_cr%C3%ADtica>`.

T

texto en claro Texto sin cifrar, p. 9.