Universidad Autónoma Metropolitana
Unidad Iztapalapa
División de Ciencias Básicas e Ingeniería

Evaluación de rendimiento del Algoritmo de Dispersión de Información sobre los campos finitos $GF(2^8)$ *y* $GF(2^{16})$

Presenta:
Betzayda Deyanira Velázquez Méndez

Para obtener el grado de:
Maestra en Ciencias y Tecnologías de la Información

Asesor:
Dr. Ricardo Marcelín Jiménez
Profesor Titular "C"

Presidente	Secretario	Vocal
Dr. Francisco de Asís López Fuentes	Dr. Leonardo Palacios Luengas	M.C. José Ignacio Castillo Velázquez

Ciudad de México 2019

Resumen

El algoritmo de dispersión de información (IDA) es una transformación lineal sobre un campo finito, que convierte un archivo de $|F|$ bytes en n archivos de $\frac{|F|}{m}$ bytes, denominados dispersos, tales que es posible reconstruir el original, a partir de cualesquiera m dispersos.

En este trabajo se evalúa el funcionamiento de un par de implementaciones de IDA que operan sobre $GF(2^8)$ y $GF(2^{16})$, respectivamente. Ambas soportan un conjunto de parámetros $2 < m \leq 28$ y $3 < n \leq 30$.

Esta evaluación determina el desempeño de cada implementación como una función del campo finito sobre el que opera, sus parámetros (m, n), así como las condiciones de trabajo sobre las cuales se despliega, esto es, el sistema operativo, el hardware de base y la gestión de la memoria.

Además se evalúa el impacto de IDA con cada campo finito tomando en cuenta el sistema operativo, el hardware de procesamiento y la implementación en memoria, para apreciar un panorama general sobre el comportamiento de la velocidad de procesamiento de acuerdo a las características que necesite el usuario.

Los resultados muestran que a pesar del sistema operativo, el hardware utilizado o la implementación en memoria, siempre hay un aumento de velocidad utilizando el campo $GF(2^{16})$ en comparación con el campo $GF(2^8)$.

También encontramos que el tamaño de los elementos de trabajo no es más que una forma de controlar las operaciones de entrada y salida, que resultan tener el impacto más sensible en el rendimiento de IDA.

Tabla de Contenido

Capítulo 1

Marco Introductorio

Resumen

En este capítulo se describe el protocolo de investigación, el cual se centra en la descripción del problema a investigar, trabajo relacionado, así como de la metodología con que se aborda su solución.

1.1. Introducción

La información se ha convertido en un activo estratégico de las organizaciones, cuyo volumen y tasas de crecimiento obligan a revisar los mecanismos para su adecuada gestión. La gestión de la información es entendida como las herramientas para su almacenamiento, consulta, transporte e incluso, generación de conocimiento [1].

Según el índice global de Cisco, para el 2021, la capacidad de almacenamiento en los centros de datos crecerá 2.6 Zettabytes en comparación con 663 Exabytes en 2016, un crecimiento de casi 4 veces mayor, lo que implica reconsiderar los mecanismos de almacenamiento de la información [2]. En este sentido, es indispensable pensar en 2 requerimientos básicos que deben cuidarse en los sistemas diseñados para tal propósito: la alta disponibilidad y la escalabilidad.

[1] En términos generales, cubre todos los aspectos de la producción, coordinación, almacenamiento, recuperación y difusión de la información, independientemente del formato o la fuente, y sugiere un aspecto organizativo que impartirá cierto grado de valor agregado a la información [1].

La alta disponibilidad se consigue garantizando que el sistema preste servicio aún si varios de sus componentes salieran de operación, es decir, se consigue utilizando recursos redundantes. En este sentido, la alta disponibilidad va de la mano de la confiabilidad y la tolerancia a fallas. En cuanto a la escalabilidad se refiere, esta propiedad garantiza que un sistema puede crecer en sus capacidades sin perder calidad en el soporte de los servicios que ofrece.

El algoritmo de dispersión de información [3], es una interesante alternativa para generar información redundante y almacenarla, de tal forma que, aún si alguno de los dispositivos de almacenamiento falla, sería posible reconstruir con un mínimo disponible la información inicial.

Por consiguiente, la optimización del algoritmo de dispersión de información, es un tema de relevancia y trascendencia en el uso y aprovechamiento de las tecnologías de la información, debido a que se obtendría una relación costo/beneficio que mejora los recursos de tiempo y almacenamiento de información en grandes organizaciones que diariamente trabajan cantidades masivas de datos.

1.2. Definición del problema

Muchos proyectos de almacenamiento de alto desempeño como EMC [4], Babel [5] o almacenamiento en la nube [6] han utilizado el Algoritmo de Dispersión de Información (IDA) por las posibilidades que ofrece este algoritmo para configurar una estrategia de almacenamiento tolerante a fallas, con diferentes grados de redundancia en la información y un uso eficiente del almacenamiento.

Actualmente, se ha desarrollado una primera versión del algoritmo que funciona sobre el campo $GF(2^8)$, esto es, el conjunto de las cadenas de 8 bits [7]. Aun cuando la implementación en software del algoritmo es relativamente rápida, se requiere acelerar el proceso si se piensa usar en aplicaciones de mayor demanda, como el caso de los servidores de almacenamiento. En la sección 2 se explicará el concepto de GF.

En este sentido, el problema de investigación es mejorar el tiempo de procesamiento de IDA utilizando campos finitos de orden mayor al de $GF(2^8)$ para utilizarlo en aplicaciones de alto desempeño.

1.3. Justificación

Un criterio para comparar los tipos de redundancia es por la cantidad de fallas que puede tolerar. Existen 3 tipos de redundancia: (a) de equipos o física, (b) de tiempo, y (c) de información. La redundancia física (a) utiliza un conjunto de equipos de respaldo, esta nos permite tener una tolerancia muy grande, sin embargo, el conjunto de recursos en exceso puede ser muy costoso.

La redundancia (b) de tiempo es repetir una operación o procedimiento hasta tener la certeza que se ha completado con éxito o que debe abortarse. La redundancia de información (c) se basa en codificar la información con un exceso de dígitos binarios tales que, si alguno de estos dígitos se perdieran, sería posible recuperar el mensaje original con los dígitos restantes [8].

Debido a que las empresas generan grandes volúmenes de información, es posible optimizar la redundancia consumiendo menos recursos para garantizar la alta disponibilidad. Esto se consigue con la redundancia de información, específicamente con IDA. Asimismo, este algoritmo es utilizado en diversas aplicaciones como la transmisión de información tolerante a fallos, así como en la comunicación entre procesadores de forma paralela, entre otros [9].

Dado un archivo F que consta de $|F|$ bits, IDA lo transforma en n archivos llamados dispersos donde cada uno de estos es de tamaño $\frac{|F|}{m}$, donde $1 < m < n$, y tales que bastan cualesquiera m dispersos para recuperar a F [3]. Por ejemplo, en un algoritmo IDA con parámetros $n = 5$ y $m = 3$ se transformaría al archivo F en 5 dispersos, cada uno de $\frac{1}{3}$ de $|F|$ tal que bastarían cualesquiera 3 de ellos para recuperar a F. En comparación con la redundancia basada en copias o réplicas, el costo de exceso de información es del 66 % y con una tolerancia de 2 fallas.

Observemos que IDA ofrece la misma tolerancia a fallas a un costo mucho menor como lo muestra Zhao [10] que compara el rendimiento de IDA con métodos actuales de replicación o redundancia física y como resultado de estudio, se demuestra que IDA obtiene mejor rendimiento pudiendo tolerar el mismo número de fallas con una cantidad sensiblemente menor de información redundante.

A pesar de que IDA ofrece diferentes puntos de equilibrio entre su costo y beneficio, la complejidad computacional del algoritmo obliga a buscar implementaciones que mejoren el costo en tiempo. Por consiguiente, este proyecto se enfocará a mejorar la velocidad de IDA al usar un campo finito de un orden superior a (2^8), debido a que las computadoras actuales nos permiten trabajar con palabras de datos de 16, 32 o hasta 64 dígitos simultáneos.

1.4. Objetivos

1.4.1. Objetivo general

Implementar el algoritmo de dispersión de información (IDA) sobre un campo finito de orden superior al de $GF(2^8)$ para aumentar la velocidad en aplicaciones de alto desempeño.

1.4.2. Objetivos específicos

- Analizar el funcionamiento de la versión actual de IDA.
- Construir una versión de IDA sobre $GF(2^8)$ que soporte un conjunto amplio de parámetros de operación (n, m).
- Elaborar una versión de IDA sobre $GF(2^{16})$ para un conjunto amplio de parámetros de operación (n, m).
- Determinar el impacto de las condiciones de operación sobre el desempeño de las diferentes instancias del algoritmo.

1.5. Preguntas de investigación

- ¿Cómo funciona IDA sobre los campos finitos $GF(2^8)$ y $GF(2^{16})$?
- ¿Afecta aumentar el orden del campo a $GF(2^{16})$ en el algoritmo IDA?
- ¿Cómo aumenta la velocidad de IDA en comparación con la versión anterior?
- ¿Cómo es el comportamiento de IDA en diferentes condiciones de operación?

1.6. Variables de investigación

Las variables de investigación son un término para hacer referencia a los cambios que se evaluarán en la investigación. Estas se dividen en variables independientes y variables dependientes, de las cuales, la manipulación de la variable independiente causa efecto en la o las variables dependientes. En esta investigación las variables serán las siguientes:
Variables independientes:

- Campo Finito $GF(2^8)$
- Campo Finito $GF(2^{16})$
- Parámetros (n, m)
- Equipo de cómputo
- Tamaño de archivo
- Sistema Operativo

Variables dependientes:

- Tiempo de dispersión.
- Tiempo de recuperación.

El número de elementos de un campo, es llamado el orden del campo [11].Un campo finito se definirá con detalle en el capítulo 2.

n: Número de dispersos totales.

m: Umbral de recuperación o número mínimo con el cual se puede recuperar un archivo [3].

Tiempo de dispersión: Tiempo que tarda en generar el número de dispersos n [12].

Tiempo de recuperación: Tiempo que tarda en restaurar un archivo a partir de m dispersos [13].

1.7. Limitaciones

El proyecto estará limitado por las siguientes condiciones:

- Tamaño del campo:
  1. $GF(2^8)$
  2. $GF(2^{16})$
- Tamaño del archivo:
  1. Mínimo 1 MB
  2. Máximo 1024 MB[2]
- Intervalo de n y m
  1. n=[3,30]
  2. m=[2,28]

1.8. Hipótesis

Si utilizamos cantidades bits mayores a un byte para el procesamiento del algoritmo, es decir, crecemos el orden del campo de trabajo de $GF(2^8)$ a $GF(2^{16})$, entonces disminuirá el tiempo de procesamiento del algoritmo de dispersión de información de manera significativa.

[2]Equivalente a aproximadamente 1 hora de video de 720x1280, a 30fps

1.9. Metodología

Se realizará un estudio cuantitativo-correlacional, y se determinará cuál es la ganancia en tiempo que causa el incremento de orden del campo finito en el algoritmo de dispersión de información.

Paso 1. Análisis sobre el funcionamiento de la versión actual de IDA.

Actividades:

a) Revisión de la literatura de IDA.
b) Estudio del funcionamiento de IDA.
c) Estudio del funcionamiento sobre campos finitos.
d) Estudio del funcionamiento sobre campos finitos extendidos.
e) Implementación de operaciones básicas para IDA.

Paso 2. Construcción de IDA con n y m sobre el campo $GF(2^8)$.

Actividades:

a) Desarrollo de una biblioteca de aritmética sobre campos finitos.
b) Desarrollo de una biblioteca de álgebra de matrices sobre campos finitos.
c) Integración de ambas bibliotecas con n y m sobre $GF(2^8)$.
d) Elaboración del protocolo de investigación, así como del estado del arte.
e) Reportar en forma de artículo los avances obtenidos.

Paso 3. Elaboración de IDA con n y m sobre $GF(2^{16})$.

Actividades:

a) Modificación de la biblioteca de aritmética sobre campos finitos.
b) Modificación de la biblioteca de álgebra de matrices sobre campos finitos.
c) Integración de ambas bibliotecas con n y m sobre $GF(2^{16})$.

Paso 4. Determinación del impacto de las condiciones de operación de IDA sobre $GF(2^8)$ y $GF(2^{16})$.

Actividades:

a) Determinación del tiempo que tarda IDA sobre $GF(2^8)$ y $GF(2^{16})$.

b) Comparación de la versión actual contra la versión optimizada sobre $GF(2^{16})$.

c) Determinación del tiempo de IDA sobre $GF(2^8)$ y $GF(2^{16})$ con diferentes sistemas operativos.

d) Determinación del tiempo de IDA sobre $GF(2^8)$ y $GF(2^{16})$ con diferente hardware de procesamiento.

e) Determinación del tiempo de procesamiento entre implementaciones realizadas sobre disco y memoria.

Paso 5. Comparación sobre los experimentos realizados

Actividades:

a) Comparar cada una de las condiciones de operación de IDA sobre los distintos campos.

b) Idónea Comunicación de Resultados (ICR).

Recursos:

- Material bibliográfico
- Compilador gcc
- Computadora de 64 bits
- Computadora de 32 bits con diferente hardware a la anterior
- Sistema Operativo Linux y Windows

Actividad	Proyecto de Investigación								
	Trimestre I			Trimestre II			Trimestre III		
Análisis sobre el funcionamiento de la versión actual de IDA.									
Revisión de la literatura de IDA									
Estudio del funcionamiento de IDA.									
Estudio del funcionamiento sobre campos finitos.									
Estudio del funcionamiento sobre campos finitos extendidos.									
Implementación de operaciones básicas para IDA.									
Construcción de IDA con cualesquiera *n* y *m* sobre el campo $GF(2^8)$.									
Desarrollo de una biblioteca de aritmética sobre campos finitos.									
Desarrollo de una biblioteca de álgebra de matrices sobre campos finitos.									
Integración de ambas bibliotecas con cualesquiera *n* y *m* sobre $GF(2^8)$.									
Elaboración del protocolo de investigación, así como del estado del arte.									
Reportar en forma de artículo los avances obtenidos.									
Elaboración de IDA con cualesquiera *n* y *m* sobre $GF(2^{16})$.									
Modificación a biblioteca de aritmética sobre campos finitos.									
Modificación a biblioteca de álgebra de matrices sobre campos finitos.									
Integración de ambas bibliotecas con cualesquiera *n* y *m* sobre $GF(2^{16})$.									
Reportar en forma de artículo los avances obtenidos.									
Determinación del impacto de las condiciones de operación de IDA sobre $GF(2^8)$ y $GF(2^{16})$									
Determinación del tiempo que tarda IDA sobre $GF(2^8)$ y $GF(2^{16})$.									
Comparación de la versión actual contra la versión optimizada sobre $GF(2^{16})$.									
Determinación del tiempo de IDA sobre $GF(2^8)$ y $GF(2^{16})$ con diferentes sistemas operativos.									
Determinación del tiempo de IDA sobre $GF(2^8)$ y $GF(2^{16})$ con diferente hardware de procesamiento.									
Determinación del tiempo de procesamiento entre implementaciones realizadas sobre disco y memoria.									
Comparación sobre los experimentos realizados									
Contrastar cada una de las condiciones de operación de IDA sobre los distintos campos.									
Idónea comunicación de resultados.									

Figura 1.1: Planeación de Actividades

1.10. Antecedentes

El algoritmo IDA es utilizado en diversas aplicaciones para redes de computadoras, sistemas distribuidos, y en sistemas de almacenamiento. A continuación se mencionan algunas de las investigaciones relacionadas con IDA.

Tanenbaum menciona que la técnica clave para la tolerancia es la redundancia. Especifica que para los sistemas de archivos distribuidos, básicamente hay dos tipos de métodos para obtener la redundancia: replicación y codificación. La codificación consiste en agregar bits extra a los datos para detectar errores y puedan ser corregidos [8].

En 1989, Michael O. Rabin presentó el algoritmo de dispersión de información *(IDA: Information Dispersal Algorithm)*, el cual utiliza la redundancia por codificación. Dado un archivo F que consta de $|F|$ bits se transforma en n archivos llamados dispersos, donde cada uno de estos dispersos es de tamaño $\frac{|F|}{m}$, cumpliendo la relación $1 < m < n$, y tales que, bastan cualesquiera m dispersos para recuperar a F [3].

Sin profundizar sobre los diversos temas que implica IDA, Plank presentó una explicación completa y una implementación detallada sobre cómo realizar la codificación de la redundancia de información para los programadores de sistemas con conocimientos mínimos en el álgebra o teoría de campos finitos, para mejorar la confiabilidad en sistemas similares a RAID, sin necesidad de consultar referencias externas [14].

Alnafoosi contribuyó con un marco conceptual integrado para evaluar y ayudar a seleccionar la solución de almacenamiento óptima para los requisitos de múltiples variables como: la capacidad, escalabilidad, presupuesto, carga de trabajo, seguridad, privacidad, disponibilidad, fiabilidad, entre otros. Además, examina IDA, utilizando este marco y es el primero en examinar cuatro diferentes tecnologías de almacenamiento de datos masivos (Big Data) utilizando este marco [15].

La siguiente sección se divide en dos apartados. El primer apartado se refiere a las aplicaciones, que describen la utilidad de IDA para diversos propósitos, por ejemplo, transmisión eficiente de la información, tiempo de procesamiento, entre otros. El segundo apartado se refiere a las investigaciones, que describe el estudio del algoritmo como seguridad de la información, rendimiento del algoritmo en sistemas de almacenamiento, tecnología de almacenamiento, entre otros.

1.10.1. Aplicaciones

En 1990 Rabin anunció las posibles aplicaciones del algoritmo de dispersión de información (IDA), para el almacenamiento seguro y fiable de la información, en redes

de computadoras e incluso en discos individuales, tolerancia a fallos, transmisión eficiente de la información en redes, y la comunicación entre procesadores para cómputo en paralelo. También propuso aplicaciones al problema de la consistencia y disponibilidad de datos en sistemas distribuidos y a un algoritmo distribuido de concordancia de patrones [9].

Igualmente, en el libro titulado *"Information Dispersal and Parallel Computation"* se hace referencia a IDA para realizar un encaminamiento paralelo tolerante a fallos. Para esto, se divide el paquete en piezas por medio de IDA, posteriormente los paquetes son distribuidos por la red en paralelo a sus destinos. Debido a que puede existir pérdida de información, el algoritmo de dispersión de información es de gran importancia, ya que es posible recuperar el archivo con algunas de sus piezas, esto evita la retransmisión de información y por lo tanto evita tener demasiado tráfico en la red [16].

Posteriormente, Hung-Min descubrió características que ayudan a reducir el tiempo de procesamiento para IDA y propone un método para la determinación de las variables óptimas del algoritmo enfocado a la comunicación de redes no fiables, tomando en cuenta los caminos disponibles [17]. Asimismo, Marvin analizó la cantidad de redundancia necesaria en la utilización de IDA en redes de computadoras, tomando en cuenta el tráfico en la red, la latencia así como el tamaño del archivo para maximizar la probabilidad de éxito, enfocándose al soporte de la calidad del servicio (QoS) [18].

Por su parte, Cheikh, propuso un protocolo utilizando IDA para la seguridad en MapReduce, se basan en la idea de dispersar cuidadosamente los trozos de un archivo, de tal manera que un atacante no pueda reconstruir el archivo con un solo nodo atacado, de tal manera que tendría que colaborar con otros nodos para obtener el archivo, no obstante algunos nodos se encontrarán en una zona segura, por consiguiente, no se podrá recuperar el archivo en MapReduce [19].

Otra aplicación que presentó Afianian es un método para mejorar IDA mediante el pre-procesamiento eficiente antes de la dispersión del algoritmo. Cabe mencionar que este método está enfocado a la computación en la nube móvil, por consiguiente se emplea un mecanismo de seguridad de baja energía, para el procesamiento móvil [20].

Según Kheng Kok la mayoría del almacenamiento en la nube que utilizan las corporaciones emplean la replicación como método de almacenamiento, por consiguiente, en este trabajo describe un método alternativo para almacenar los datos de la nube que se basan en IDA y el intercambio secreto de claves para tratar los aspectos de seguridad y disponibilidad de los datos [6].

Recientemente, el artículo realizado por Zhi-ting Yu y otros colaboradores proponen un modelo de cifrado llamado SIDA, el cual está basado en IDA y en el cifrado de múltiples capas. Utilizan IDA como un mecanismo de seguridad, es decir, para que su atacante no

pueda obtener los datos originales debido a que los datos se encuentran transformados y segmentados. Todas las operaciones de bytes en el algoritmo de dispersión de información están sobre el campo finito $GF(2^8)$ [21].

Como aplicación reciente Quezada y Marcelín presentaron un sistema de distribución de información que puede ser construido con equipos o dispositivos comerciales de bajo costo, en el cual los usuarios pueden tener fácil acceso a ese tipo de equipo. Este sistema implementa el algoritmo de dispersión de información para realizar la distribución de la información [7].

1.10.2. Investigaciones con IDA

A continuación se describen investigaciones relacionadas con IDA. Tal es el caso de la comparación que realiza Weatherspoon sobre los códigos de borrado como el algoritmo IDA, en el cual indica que estos tienen un orden menor de ancho de banda y de almacenamiento en comparación con los sistemas de replicación. Para realizar la comparación, construye una infraestructura de almacenamiento distribuido tolerante a fallos [22].

Asimismo, Hernández muestra una comparación entre los métodos de redundancia y el algoritmo de dispersión de información sobre el almacenamiento en la nube privada e híbrida. Alguno de los dispersos que IDA transforma, son almacenados en nube, esto con el fin de evitar recuperar el archivo completo desde una zona privada del sistema de almacenamiento [23].

Un artículo realizado por Zhao presentó un estudio sobre el rendimiento de IDA, donde se implementan dos sistemas de almacenamiento y se muestran que los métodos basados en IDA superan a los métodos de replicación de datos (REP). Zhao muestra que IDA es más rápido con implementación en hardware adicional, como es el caso de las tarjetas gráficas Nvidia y procesadores de múltiples núcleos como los Intel Xeon pero no se ha trabajado los campos finitos como en $GF(2^{16})$ para dicho algoritmo.

El mismo trabajo, muestra la comparación del rendimiento de los algoritmos de las dos técnicas más utilizadas para la redundancia de la información, esto trae como consecuencia que IDA sigue dando buenos resultados. El rendimiento de IDA y el algoritmo REP es semejante cuando el número de fallas de nodos son pocos, en cambio a medida que los nodos presentan más fallas, IDA obtiene mejores resultados que REP [10].

Además, Lee también analizó IDA para aplicarlo en una nueva técnica sobre un centro de datos en la nube para el almacenamiento con el propósito de mantener copias de seguridad confidenciales de los archivos de datos. El análisis se realiza comparando los algoritmos de replicas de información [24].

A pesar de que IDA demuestra ser un buen candidato en el rendimiento, no es tan seguro como el algoritmo de Shamir. Se podría pensar que IDA proporciona seguridad debido a la distribución de sus dispersos, es decir, los dispersos obtenidos por el algoritmo IDA pueden almacenarse en diferentes localidades, de tal manera que si algún intruso llegase a obtener un disperso de la información, no podría recuperar el archivo completo, pero sí una parte de él. No obstante, si el intruso llegase a obtener número mayor o igual al número del umbral de recuperación m, entonces se podría recuperar el archivo completo.

Con lo anterior, Ermoakiva presenta una arquitectura segura para el intercambio de registros médicos en Alemania con entorno en la nube, la cual proporciona disponibilidad, confidencialidad e integridad de los registros. En su arquitectura, implementan y realizan experimentos de rendimiento entre IDA y el esquema de compartición de secretos de Shamir. Sus resultados muestran que, el esquema de compartición de secretos de Shamir, funciona más lento que IDA, tanto en la dispersión como en la reconstrucción de la información, esto es debido al cifrado que realiza el algoritmo de compartición de secretos de Shamir [25].

En este sentido, IDA no es el algoritmo indicado si nos referimos estrictamente a la seguridad de la información, específicamente, en el cifrado de los datos. Por consiguiente el algoritmo de Shamir, obtiene otros beneficios de seguridad a pesar del rendimiento en comparación con IDA.

Por otra parte, existen compañías que también trabajan con IDA, tal es el caso de la empresa Cleversafe, junto con Hewlett Packard reportan un nuevo modelo de almacenamiento de datos utilizando IDA optimizado, el cual puede trabajar a escala de petabytes y a un costo menor en comparación de sistemas tradicionales de almacenamiento, además de ofrecer la escalabilidad necesaria para el crecimiento de una organización. Cabe mencionar que Cleversafe utiliza algoritmos de dispersión de información patentados [26].

Adicionalmente la empresa IBM construye una tecnología llamada "IBM Cloud Object Storage"basado en el algoritmo de dispersión de información, la cual ofrece soluciones para empresas que necesitan almacenar gran volumen de datos y recalcan la conveniencia de una tecnología diferente como el almacenamiento de dispersión de información. IBM Cloud Object Storage utiliza tecnología desarrollada por la empresa Cleversafe, que fue adquirida por IBM en 2015 [27].

La ventaja de utilizar IDA es que funciona sobre cualquier tecnología de almacenamiento como discos magnéticos, discos de estado sólido e incluso se podría pensar en futuro en moléculas de ADN, ya que IDA es independiente del dispositivo de almacenamiento a utilizar, es decir, aunque al paso de los años la tecnología avance, IDA aún podría implementarse.

Capítulo 2

Marco Teórico

Resumen

El objetivo principal de este capítulo es dar un panorama general de los conceptos teóricos en los cuales se fundamenta el algoritmo IDA como los campos finitos de extensión, así como los métodos para la construcción y manipulación de la matriz de Vandermonde y a su vez, un método para la inversión de la misma. Por último, se describe IDA en el proceso de dispersión así como el proceso de recuperación.

2.1. Fundamentos teóricos

2.1.1. Estructuras algebraicas

Una estructura algebraica es un objeto matemático que consiste en un conjunto no vacío C con operaciones definidas sobre él. Algunas estructuras cumplen las siguientes propiedades [28]:

1. Operación Binaria

   Definición 2.1.1 *Para cualquier $a, b \in C$ y $\star$ un operador :*

   $$\star : (a, b) \rightarrow a \star b \in C$$

2. Propiedad Asociativa

   Definición 2.1.2 *Para cualquier $a, b, c \in C$ y $\star$ un operador :*

$$\star : (a, b, c) \to (a \star b) \star c = a \star (b \star c)$$

3. Propiedad Conmutativa

   Definición 2.1.3 *Para cualquier $a, b \in C$ y $\star$ un operador :*

$$\star : (a, b) \to a \star b = b \star a$$

4. Propiedad Distributiva

   Definición 2.1.4 *Para cualquier $a, b, c \in C$ y $\star$, $\diamond$ operadores :*

$$\star, \diamond : (a, b, c) \to a \diamond (b \star c) = (a \diamond b) \star (a \diamond c)$$

5. Elemento Neutro

   Definición 2.1.5 *Existe un elemento $e \in C$ llamado elemento neutro tal que, dado un operador $\star$, para cualquier $a \in C$.*

$$\star : (a, e) \to a \star e = e \star a = a$$

6. Elemento Inverso o Elemento Simétrico

   Definición 2.1.6 *Para cualquier $a \in C$, dado un operador $\star$, existe un elemento a^{-1} llamado elemento inverso de a, tal que:*

$$\star : (a, a^{-1}) \to a \star a^{-1} = e$$

Existen diferentes tipos de estructuras que se diferencian por la composición de algunas de las propiedades anteriores. Enseguida, detallaremos algunas de éstas.

Grupo

Definición 2.1.7 *Un grupo $(G, \star)$ es un conjunto no vacío G dotado con un operador binario $\star$ y cumple con las propiedades:*

- *P. Asociativa*
- *Elemento Neutro*
- *Elemento Inverso*

Definición 2.1.8 *Un grupo $(G, \star)$ es* ***abeliano*** *o* ***conmutativo*** *si es un grupo y además, incluye la propiedad conmutativa*

Anillo

Definición 2.1.9 *Un Anillo $(A, \star, \diamond)$ es un conjunto no vacío A con operadores $\star, \diamond$ que cumple las propiedades:*

- *$(A, \star)$ sea un grupo abeliano*
- *Para el operador $(A, \diamond)$*
  - *P. Asociativa*
- *Para ambos operadores $\star, \diamond$*
  - *P. Distributiva*

Campo

Definición 2.1.10 *Un campo $(K, \star, \diamond)$ es un conjunto no vacío K con operadores $\star, \diamond$ que cumple con las propiedades:*

- *$(K, \star)$ sea un grupo abeliano*
- *$(K - e, \diamond)$ sea un grupo abeliano, donde e es un elemento neutro*
- *P. Distributiva*

Estructura Algebraica	Operador	Operación Binaria	P. Asociativa	P. Conmutativa	P. Distributiva	Elemento Neutro	Elemento Inverso
Grupo	$\star$	✓	✓			✓	✓
Grupo Abeliano	$\star$	✓	✓	✓		✓	✓
Anillo	$\star$	✓	✓	✓		✓	✓
	$\diamond$	✓	✓		✓		
Campo	$\star$	✓	✓	✓		✓	✓
	$\diamond$	✓	✓	✓	✓		✓

Tabla 2.1: Estructuras algebraicas y sus propiedades

Campos finitos

Un campo finito F es un conjunto finito de números enteros módulo p, donde p es un primo, cerrado bajo las operaciones de suma (+) y multiplicación (*); contiene 2 elementos: llamado neutro aditivo o '0' y neutro multiplicativo también llamado '1'.

Para todo número x en el campo, existe un número $-x$ llamado inverso aditivo, tal que

$$x + (-x) = 0 \tag{2.1}$$

Para todo número x distinto de 0, existe un número x^{-1} llamado inverso multiplicativo tal que

$$x * (x^{-1}) = 1 \tag{2.2}$$

Existen tantos campos finitos como números primos. Para todo primo p, los enteros módulo p forman un campo de p elementos, denotado por F_p.

Ejemplo 2.1.1 *Sea $p = 5$, los elementos de F_5 son $\{0, 1, 2, 3, 4\}$ bajo las operaciones de suma y multiplicación módulo 5. Haciendo uso de la tabla 2.2 y 2.3 sobre F_5 respectivamente, se calculan las siguientes operaciones aritméticas:*

$$
\begin{aligned}
&\textit{Suma}: && 4+3=2\\
&\textit{Inverso aditivo}: && -2=3\\
&\textit{Multiplicación}: && 2\times 4=3\\
&\textit{Inverso multiplicativo}: && 3^{-1}=2
\end{aligned}
$$

$+$	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

Tabla 2.2: Adición en F_5

$\times$	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

Tabla 2.3: Multiplicación en F_5

Ejemplo 2.1.2 *Sea $p=2$, los elementos de F_2 son $\{0,1\}$ bajo las operaciones de suma y multiplicación módulo 2. Haciendo uso de la tabla 2.4 y 2.5 sobre F_2 respectivamente, se calculan las siguientes operaciones aritméticas:*

$$
\begin{aligned}
&\textit{Suma}: && 1+1=0\\
&\textit{Inverso aditivo}: && -1=1\\
&\textit{Multiplicación}: && 1\times 1=1\\
&\textit{Inverso multiplicativo}: && 1^{-1}=1
\end{aligned}
$$

+	0	1
0	0	1
1	1	0

Tabla 2.4: Adición en F_2

×	0	1
0	0	0
1	0	1

Tabla 2.5: Multiplicación en F_2

2.2. Campos finitos de extensión

Además de los campos finitos de tipo F_p, es posible construir otros conjuntos que satisfagan las propiedades de un campo finito, estos se denominan campos finitos de extensión y suelen denotarse como $GF(p^n)$[1], siendo p un primo y n un entero. Se dice en este caso, que el campo es de característica p y de orden p^n. La elección del campo depende del problema a resolver , siendo los campos $GF(2^8)$ y $GF(2^{16})$ los que se consideran útiles para los alcances de este trabajo.

2.2.1. Polinomios generadores y campos de extensión

El conjunto de polinomios $GF_2[x]$ con coeficientes en F_2 son expresiones de la forma:

$$a_0 + a_1 x + \ldots + a_n x^n, \tag{2.3}$$

donde los $a_j \in F_2, n \geq 1$ es un entero. Si se sabe que $\pi(x)$ es un polinomio irreducible, es decir, aquellos que no pueden descomponerse como el producto de dos polinomios en $GF_2[x]$ de grados menores que n, es posible construir un campo con las clases residuales módulo el polinomio $\pi(x)$. A este campo de clases residuales se le denomina "extensión del campo F ", siendo $\pi(x)$ el polinomio generador del campo.

Con cualquier polinomio $f(x) \in GF_2[x]$ de grados menores a n podemos asociar una $n - tupla$ binaria. Inversamente, con cualquier $n - tupla$ binaria puede ser interpretada

[1]Galois Field, llamado así por Évariste Galois

como un polinomio en $GF_2[x]$, es decir, la $n-tupla$ $(a_0, a_1, a_2, \cdots, a_{n-1})$, corresponde al polinomio:

$$f(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}, \tag{2.4}$$

donde el grado de $f(x)$ es a lo más $n-1$ [29].

Ejemplo 2.2.1 *El polinomio* $1x^3 + 0x^2 + 1x + 1 = x^3 + x + 1 \in GF_2[x]$ *corresponde a la 4-tupla de coeficientes binarios (1011).*

Definición 2.2.1 *Sea GF un campo de extensión y* $\alpha \in GF$ *, tal que* $\pi(x)$ *evaluado en* α *es igual a 0. Se dice entonces que* α *es un cero del polinomio irreducible generador de GF.*

Ejemplo 2.2.2 *Construir* $GF(2^4)$ *generado por el polinomio irreducible* $\pi(x) = x^4 + x + 1$.

Si $\alpha \in GF(2^4)$ *es un cero de* $\pi(x)$*, esto es, por definición 2.2.1 tenemos que:*

$$f(\alpha) = \alpha^4 + \alpha + 1 = 0 \tag{2.5}$$

$$\therefore \alpha^4 = \alpha + 1 \tag{2.6}$$

Ahora vamos a asociar cada tupla binaria de 4 bits, con un polinomio en α*, y una potencia de* α*:*

(0000)	0
(0001)	1
(0010)	α
(0100)	α^2
(1000)	α^3

Podemos calcular α^5 *a partir de* $f(x)$*, de la siguiente manera*

$$\begin{array}{r|l} & \alpha \\ \hline \alpha^4 + \alpha + 1 & \alpha^5 \\ & -\alpha^5 - \alpha^2 - \alpha \\ \hline & -\alpha^2 - \alpha \end{array} \tag{2.7}$$

Tomando en cuenta las operaciones de la tabla 2.4 obtenemos que $-\alpha = \alpha$

$$\therefore \quad \alpha^5 = \alpha^2 + \alpha \tag{2.8}$$

De otra manera, utilizando 2.6 podemos generar las potencias de α

$$\begin{aligned} \alpha^5 &= (\alpha)\alpha^4 = \alpha(\alpha+1) = \alpha^2 + \alpha \\ \alpha^6 &= (\alpha)\alpha^5 = \alpha(\alpha^2+\alpha) = \alpha^3 + \alpha^2 \\ \alpha^7 &= (\alpha)\alpha^6 = \alpha(\alpha^3+\alpha^2) = \alpha^4 + \alpha^3 = \alpha^3 + \alpha + 1 \end{aligned} \tag{2.9}$$

Obteniendo el campo completo en la Tabla 2.6

4-tuplas binarias	Polinomial	Exponencial
(0000)	0	-
(0001)	1	α^0
(0010)	α	α^1
(0100)	α^2	α^2
(1000)	α^3	α^3
(0011)	$\alpha+1$	α^4
(0110)	$\alpha^2+\alpha$	α^5
(1100)	$\alpha^3+\alpha^2$	α^6
(1011)	$\alpha^3+\alpha+1$	α^7
(0101)	α^2+1	α^8
(1010)	$\alpha^3+\alpha$	α^9
(0111)	$\alpha^2+\alpha+1$	α^{10}
(1110)	$\alpha^3+\alpha^2+\alpha$	α^{11}
(1111)	$\alpha^3+\alpha^2+\alpha+1$	α^{12}
(1101)	$\alpha^3+\alpha^2+1$	α^{13}
(1001)	α^3+1	α^{14}

Tabla 2.6: LD: Representaciones de los elementos en $GF(2^4)$

Sea a_0, a_1, a_2, a_3 una 4-tupla binaria, decimos que el logaritmo discreto de ésta, denotado como LD de a_0, a_1, a_2, a_3, es igual al exponente i de α que aparece en la última columna en el renglón de a_0, a_1, a_2, a_3. Alternativamente el antilogaritmo de i, denotado como AD, corresponde con la tupla binaria de la primera columna, en el renglón que corresponde al exponente i.

4-tuplas binarias	Posición	Exponencial
(0000)	0	-
(0001)	1	α^0
(0010)	2	α^1
(0100)	3	α^4
(1000)	4	α^2
(0011)	5	α^8
(0110)	6	α^5
(1100)	7	α^{10}
(1011)	8	α^3
(0101)	9	α^{14}
(1010)	10	α^9
(0111)	11	α^7
(1110)	12	α^6
(1111)	13	α^3
(1101)	14	α^{11}
(1001)	15	α^{12}

Tabla 2.7: AD: Representaciones de los elementos en $GF(2^4)$

2.2.2. Aritmética sobre campos finitos de extensión

Las operaciones algebraicas como suma, resta, multiplicación y división sobre un campo finito de extensión se aclararán con un ejemplo.

Ejemplo 2.2.3 *Construir las operaciones algebraicas de suma, resta, multiplicación y división implicadas por los polinomios* $\alpha^3+\alpha^2+\alpha+1$ *y* $\alpha^3+\alpha^2+\alpha \in GF(2^4)$

- *Suma: Tomando en cuenta las operaciones de la tabla 2.4 obtenemos que* $1+1=0$*, por tanto, si sumamos los coeficientes que corresponden al mismo grado de* α*, entonces conseguimos:*

$$(\alpha^3+\alpha^2+\alpha+1)+(\alpha^3+\alpha^2+\alpha)=1 \tag{2.10}$$

- *Resta: Tomando en cuenta las operaciones de la tabla 2.4 obtenemos que el inverso aditivo de 1 , es decir -1 , es el mismo uno, por tanto:*

$$(\alpha^3+\alpha^2+\alpha)-(\alpha^3+\alpha^2+\alpha+1)=-1=1 \tag{2.11}$$

Interpretando los polinomios $\alpha^3+\alpha^2+\alpha+1$ *y* $\alpha^3+\alpha^2+\alpha$ *como una* $4-$*tupla binaria se puede realizar las operaciones suma y resta con la operación binaria xor* $\oplus$*.*

Operación	*Polinomio*	*Binario*
Suma	$(\alpha^3+\alpha^2+\alpha+1)$	1111
$+$	$(\alpha^3+\alpha^2+\alpha+0)$	1110
$\oplus$	1	0001
Resta	$(\alpha^3+\alpha^2+\alpha+1)$	1111
$-$	$(\alpha^3+\alpha^2+\alpha+0)$	1110
$\oplus$	1	
		0001

- *Multiplicación y División: Realizamos las operaciones correspondientes con respecto a la tabla de multiplicación 2.5.*

$$(\alpha^3+\alpha^2+\alpha+1)\times(\alpha^3+\alpha^2+\alpha)=\alpha^6+\alpha^4+\alpha^3+\alpha \tag{2.12}$$

$$\begin{array}{cccccccccccccc}
 & & & & & & \alpha^3 & + & \alpha^2 & + & \alpha & + & 1 \\
\times & & & & & & & & \alpha^3 & + & \alpha^2 & + & \alpha \\
\hline
 & & & & & & \alpha^4 & + & \alpha^3 & + & \alpha^2 & + & \alpha \\
+ & & & & \alpha^5 & + & \alpha^4 & + & \alpha^3 & + & \alpha^2 & + & \\
 & & \alpha^6 & + & \alpha^5 & + & \alpha^4 & + & \alpha^3 & & & & \\
\hline
 & & \alpha^6 & & & + & \alpha^4 & + & \alpha^3 & & & & \alpha
\end{array} \tag{2.13}$$

Observamos que el resultado de la ecuación 2.12 no se encuentra dentro de $GF(2^4)$*, definición 2.1.1, por consiguiente:*

$$\begin{array}{r|l}
 & \alpha^2 \quad +1 \\
\hline
\alpha^4+\alpha+1 & \alpha^6+\alpha^4+\alpha^3 \quad +\alpha \\
 & -\alpha^6 \quad -\alpha^3-\alpha^2 \\
\hline
 & \alpha^4 \quad -\alpha^2+\alpha \\
 & -\alpha^4 \quad -\alpha-1 \\
\hline
 & -\alpha^2 \quad -1
\end{array} \tag{2.14}$$

Si consideramos el residuo de la división, podemos observar que $\alpha^2+1 \in GF(2^4)$ *y tomando en cuenta las operaciones de la tabla de adición 2.4 obtenemos que* $-\alpha=\alpha \ \therefore$

$$(\alpha^3+\alpha^2+\alpha+1)\times(\alpha^3+\alpha^2+\alpha)=\alpha^2+1 \tag{2.15}$$

Utilizando el método de logaritmo discreto, podemos realizar la operación inversa para realizar las multiplicaciones e inversos multiplicativos.

Ejemplo 2.2.4 *Realizar las operaciones de multiplicación y división usando las tablas 2.6 y 2.7.*

Multiplicación:

Decimal	*Operación*	*Binario*	*Exponencial*
10	$*$	(1010)	α^9
13		(1101)	α^{13}

$$\alpha^9 * \alpha^{13} = \alpha^{22} \tag{2.16}$$

Sin embargo, α^{22} *no está dentro del campo . A pesar de ello, se puede resolver utilizando:*

$$\alpha^{22\,\%15} = \alpha^7 = 1011 = 11_{dec} \tag{2.17}$$

puesto que el grupo multiplicativo $GF(2^4)-0$, es de orden 15. Esto trae como consecuencia que α^7 pertenece al campo.

Por otra parte, se puede obtener fácilmente utilizando las tablas LD y AD como:

$$LD[(AD[10] + AD[13])\,\%15] = 11_{dec} = 1011$$

(2.18)

División:

Decimal	*Operación*	*Binario*	*Exponencial*
10	$^{-}1$	(1010)	α^9
13		(1101)	α^{13}

$$(\alpha^9)^{-1} * (\alpha^{13}) = \alpha^{9+15} * \alpha^{13} = \alpha^{24} + \alpha^{13} = \alpha^{37\,\%15} = \alpha^7 = 1011 = 11_{dec} \tag{2.19}$$

Esto se puede obtener fácilmente como:

$$LD[(AD[10] + 15 - AD[13])\,\%15] \tag{2.20}$$

Sin embargo, las potencias de α comenzarán a repetirse porque el campo GF es un campo finito, y su orden es un número finito [29].

2.2.3. Matriz de vandermonde

Definición 2.2.2 *Una matriz de Vandermonde*[2] $\mathbb{V}$ *es una matriz de tamaño* $n \times m$ *en la cual cada elemento de* $\mathbb{V}$ *consiste en:*

$$v_{i,j} = x_i^{j-1} \tag{2.21}$$

Donde i, j son subíndices para indicar su posición; además $\mathbb{V} = v_{i,j}, 1 \leq i \leq n, 1 \leq j \leq m$ [30] y para nuestro caso, x pertenece al campo finito $GF(2^8)$ o $GF(2^{16})$.

Ejemplo 2.2.5 *Una matriz de Vandermonde* $\mathbb{V}$ *sobre los enteros, tendría la siguiente forma:*

$$\mathbb{V} = \begin{bmatrix} 1^0 & 1^1 & \cdots & 1^{m-1} \\ 2^0 & 2^1 & \cdots & 2^{m-1} \\ 3^0 & 3^1 & \cdots & 3^{m-1} \\ \vdots & & & \\ m^0 & m^1 & \cdots & m^{m-1} \\ \vdots & & & \\ n^0 & n^1 & \cdots & n^{m-1} \end{bmatrix} \tag{2.22}$$

Teorema 2.2.1 *Una matriz de Vandermonde* V *es invertible si y solo si todos los renglones* v_i *son diferentes entre sí [31].*

La ventaja de utilizar una matriz de Vandermonde en IDA es que puede ser parametrizable para cualquier n y m, es decir, se puede configurar IDA para cualesquiera n y m y por el teorema 2.2.1 al reconstruir la información, los renglones de la matriz de Vandermonde de los m dispersos que "sobreviven"tienen inversa.

[2] Llamada así en honor al matemático francés Alexandre-Théophile Vandermonde.

2.2.4. Matriz inversa por método de Gauss-Jordan

El método de Gauss-Jordan es un algoritmo utilizado en álgebra lineal para encontrar la matriz inversa. Se forma una matriz aumentada a partir de la matriz a invertir $\mathbb{A}$ de tamaño $m \times m$ y la matriz identidad $\mathbb{I}$ como se muestra a continuación:

$$[\mathbb{A}|\mathbb{I}] \tag{2.23}$$

$$\left[\begin{array}{cccc|cccc} a_{0,0} & a_{0,1} & \cdots & a_{0,m-1} & 1 & 0 & \cdots & 0 \\ a_{1,0} & a_{1,1} & \cdots & a_{1,m-1} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,m-1} & 0 & 0 & \cdots & 1 \end{array}\right] \tag{2.24}$$

Para obtener la matriz inversa de $\mathbb{A}$, denotada como $\mathbb{A}^{-1}$, se debe manipular la matriz extendida utilizando operaciones elementales de Gauss, de manera que en su mitad izquierda se consiga la matriz identidad $\mathbb{I}$. Entonces, en la mitad derecha habremos construido $\mathbb{A}^{-1}$ [32].

$$[\mathbb{I}|\mathbb{A}^{-1}] \tag{2.25}$$

Las operaciones elementales de Gauss se listan a continuación:

- Sumar o restar dos renglones de la matriz aumentada
- Multiplicar un renglón de la matriz aumentada por una constante
- Dividir un renglón de la matriz aumentada por una constante

2.3. Algoritmo de Dispersión de Información (IDA)

Describiremos el funcionamiento de IDA en dos partes, primeramente la dispersión de la información y seguidamente describiremos la recuperación de la información.

2.3.1. Funcionamiento del algoritmo IDA

Sea F un archivo que consta de $|F|$ dígitos binarios, este archivo se transforma en n archivos, llamados "dispersos" cada uno de los cuales es de tamaño $\frac{|F|}{m}$, donde $1 < m < n$, y tales que bastan cualesquiera m dispersos para recuperar a F [3]. Se puede observar en este caso que el costo de exceso de información (factor de estiramiento) será (n/m)–1 y las fallas que pueden tolerarse serán $n - m$.

2.3.2. Dispersión

Para poder realizar la transformación del archivo F, se requiere del uso de una matriz $\mathbb{A}$ llamada matriz de dispersión de tamaño n x m. Su requisito de construcción es que cualesquiera m de sus renglones formen una submatriz cuadrada que es invertible. También es importante mencionar que los elementos que conforman la matriz, pertenecen a un campo finito.

$$\begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,m-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,m-1} \end{bmatrix} \tag{2.26}$$

Se considera que el archivo F está formado por una sucesión de vectores $\overline{b_1}, \overline{b_2}, \ldots, \overline{b_l}$ sobre un campo finito GF. Cada vector consta de m componentes i.e. tiene m dimensiones.

$$F = (b_0, b_1, b_2, b_3, \cdots, b_{m-1}), (b_m, b_{m+1}, b_{m+2}, \cdots, b_{2m-1}), \cdots, (b_{(l-1)m}, b_{(l-1)m+1}, ..., b_{(l-1)m+m-1}) \tag{2.27}$$

Mediante una transformación lineal, cada vector $\overline{b_i}$ se transforma en un vector $\overline{d_i}$ de n dimensiones, esto es:

$$\mathbb{A} \cdot \bar{b_i} = \bar{d_i} \tag{2.28}$$

Nótese que si F no es múltiplo de m, entonces el último vector $\overline{b_l}$ tiene que completarse con r ceros. El j-ésimo componente de cada vector forma un archivo D_j llamado disperso, que incluye además, el número r y el j-ésimo renglón de $\mathbb{A}$. Como se muestra a continuación:

$$D_{j-1} = (r, a_{j-1,0}, a_{j-1,m-1}, d_{0,j-1}, ..., d_{l-1,j-1}) \tag{2.29}$$

con $0 \leq D_j < n - 1$.

2.3.3. Recuperación

Supóngase ahora que, del vector $\overline{d_i}$ se pierden k de sus coordenadas, tales que $k = n-m$, o bien, que se seleccionan m de sus componentes, en posiciones arbitrarias pero conocidas para formar el vector $\overline{c_i}$

Esto es equivalente a suponer que $\overline{d_i}$ se obtiene de $\overline{b_i}$ a través de la transformación lineal:

$$\mathbb{B} \cdot \overline{b_i} = \overline{c_i} \tag{2.30}$$

Donde a su vez $\mathbb{B}$ se construye seleccionando los m renglones de $\mathbb{A}$ en las mismas posiciones de las componentes de $\overline{d_i}$ que participan en $\overline{c_i}$. Por el requisito de la construcción, los m renglones de $\mathbb{B}$ son linealmente independientes y, por tanto, es invertible. En consecuencia, $\overline{b_i}$ puede reconstruirse a partir de $\overline{c_i}$ y $\mathbb{B}$:

$$\overline{b_i} = \mathbb{B}^{-1} \cdot \overline{c_i} \tag{2.31}$$

2.4. Operaciones con IDA

Presentamos a continuación un análisis simplificado de las operaciones asociadas con las tareas de dispersión y recuperación. Supongamos un conjunto de condiciones de trabajo como el que se presenta en la tabla 2.8.

Variable	Descripción
F	Longitud del archivo en bytes
k	8 o 16 bits
n	Número de dispersos
m	Umbral o tamaño del vector $\overline{b}$
$\overline{b}$	Vector de m elementos de F
$\overline{d}$	Vector de n elementos producidos de $\mathbb{A}$ y $\overline{b}$
$\overline{c}$	Vector de m elementos
$\mathbb{A}$	Matriz de dispersión de tamaño n x m
$\mathbb{B}$	Matriz de recuperación de tamaño m x m

Tabla 2.8: Variables de IDA

Suponiendo que F es múltiplo de m y además $k = 8$, entonces F, así como las matrices y vectores que intervienen en los procesos de dispersión y recuperación se asumen formados por elementos que se codifican con 8 bits cada uno.

2.4.1. Proceso de dispersión con $k = 8$

Lectura

La cantidad de vectores $\overline{b}$ que se adquieren de F puede expresarse como:

$$L_1 = \frac{F}{m} \tag{2.32}$$

Dicho de otra forma, F se descompone en L_1 vectores de m entradas cada uno.

Sea $\overline{b}$, un vector obtenido al leer F, como se indica en el paso anterior. Este vector se multiplica por la matriz $\mathbb{A}$. Por cada renglón de $\mathbb{A}$, este producto da lugar a m multiplicaciones y $m - 1$ sumas.

Dado que $\mathbb{A}$ tiene n renglones, podemos decir que se realizan n veces m productos y $m - 1$ sumas para cada vector $\overline{b}$, es decir:

$$\begin{aligned} n(m) \quad & productos \\ n(m-1) \quad & sumas \end{aligned} \tag{2.33}$$

Dado que las operaciones en la ecuación 2.33 se realizan por cada vector $\overline{b}$ entonces para todo el proceso de dispersión se realizan L_1 veces. Además si consideramos la cantidad de

vectores que se adquieren de $|F|$ tenemos que: $n(m)L_1 = n(m)\frac{|F|}{m} = n|F|$ productos y $n(m-1)L_1 = n(m-1)\frac{|F|}{m}$ cuando m es muy grande $\sim n|F|$ sumas.

Escritura

Al realizar las operaciones entre el vector $\overline{b}$ y los n renglones de la matriz $\mathbb{A}$, se genera un vector llamado $\overline{d}$ de n elementos, el cual contiene la información redundante para distribuirse en n archivos llamados dispersos. Cada elemento del vector $\overline{d}$ se escribe en un disperso diferente, teniendo como consecuencia n escrituras por cada vector $\overline{d}$. Así pues, la cantidad de escrituras que se realizan para todos los vectores $\overline{d}$ generados son:

$$nL_1 \tag{2.34}$$

Debido a que la cantidad de vectores $\overline{d}$ es la misma que los vectores $\overline{b}$, entonces , la cantidad de vectores $\overline{d}$ que se generan está dada por L_1.

2.4.2. Proceso de recuperación con $k = 8$

La complejidad para obtener la matriz $\mathbb{B}^{-1}$ es de orden m^3 [33].

Lectura

Para el proceso de lectura, se toma un elemento de 8 bits de cada uno de los m dispersos "sobrevivientes", formando el vector $\overline{c}$ de m elementos. En consecuencia, realizamos m lecturas por cada vector $\overline{c}$. Tomando en cuenta que la cantidad de vectores $\overline{c}$ que se formarán serán la misma cantidad de vectores $\overline{d}$ que se escribieron en el proceso de dispersión, podemos expresar que las operaciones de lectura son:

$$mL_1 \tag{2.35}$$

Ya que la cantidad de vectores $\overline{d}$ esta dada por L_1.

La operación entre un renglón de la matriz de recuperación $\mathbb{B}^{-1}$ y el vector $\overline{c}$ requiere m productos y $m-1$ sumas, de modo que al realizar la operación entre todos los renglones de la matriz $\mathbb{B}^{-1}$ tenemos:

$$\begin{aligned} m(m) &= m^2 \qquad productos \\ m(m-1) &\qquad sumas \end{aligned} \tag{2.36}$$

Puesto que las operaciones en la ecuación 2.36 se realizan para cada vector $\overline{c}$, en consecuencia se efectúa L_1 veces. Si consideramos la cantidad de vectores que se adquieren de $|F|$ tenemos que: $m(m)L_1 = m(m)\frac{|F|}{m} = m|F|$ productos y $m(m-1)L_1 = m(m-1)\frac{|F|}{m}$ cuando m es muy grande $\sim m|F|$

Escritura

Después de haber efectuado las operaciones entre la matriz $\mathbb{B}^{-1}$ y el vector $\overline{c}$, se produce el vector $\overline{b}$, el cual contiene m entradas de la información de F , de manera que se realizan L_1 escrituras de m elementos de $\overline{b}$ de un byte sobre el archivo de recuperación.

$$mL_1 \tag{2.37}$$

2.4.3. Proceso de dispersión con $k = 16$

En cuanto para $k = 16$, las matrices y vectores que participan en el proceso de dispersión y recuperación son formados por elementos de 2 bytes o 16 bits.

Lectura

Si sabemos que para $k = 8$ cada elemento ocupa 8 bits y $L_1 = \frac{F}{m}$, podemos decir que con $k = 16$ cada elemento ocupa 16 bits, reduciendo la cantidad de vectores $\overline{b}$ que se adquieren de F, por tanto puede expresarse como:

$$L_2 = \frac{1}{2}L_1 \tag{2.38}$$

Lo que significa que F se descompone en L_2 vectores de m entradas cada uno.

Sabemos que las operaciones entre un renglón de la matriz $\mathbb{A}$ y un vector $\overline{b}$ de m elementos está dado por m multiplicaciones y $m-1$ sumas de manera que las operaciones asociadas a toda la matriz $\mathbb{A}$ son:

$$\begin{aligned} n(m) &\qquad productos \\ n(m-1) &\qquad sumas \end{aligned} \tag{2.39}$$

Como resultado obtenemos que las operaciones en la ecuación 2.39 se ejecutan tantas veces vectores $\overline{b}$ existan, es decir, se ejecutan L_2 veces. Si consideramos la cantidad de vectores que se adquieren de $|F|$ tenemos que: $n(m)L_2 = n(m)\frac{1}{2}\frac{|F|}{m} = n\frac{|F|}{2}$ productos y $n(m-1)L_2 = n(m-1)\frac{1}{2}\frac{|F|}{m}$ cuando m es muy grande $\sim n\frac{|F|}{2}$

Escritura

Al efectuar las operaciones en la ecuación 2.39 se realiza la transformación de la información, generando un vector $\overline{d}$ de n entradas, en donde cada una de las entradas son escritas en n archivos llamados dispersos, por tanto, las operaciones de escritura son n veces por cada vector $\overline{d}$. Además, la cantidad de vectores $\overline{d}$ es la misma que L_2. Así pues, las operaciones de escritura son:

$$nL_2 \tag{2.40}$$

2.4.4. Proceso de recuperación con $k = 16$

La complejidad de invertir la matriz $\mathbb{B}^{-1}$ es de orden m^3 [33], a continuación describimos como se realizan los procesos de lectura y escritura para este escenario.

Lectura

De cada m dispersos se realiza una lectura de 16 bits, formando el vector $\overline{c}$ de m elementos, sin embargo, sabemos que tenemos $\frac{1}{2}L_1$ vectores, es decir, menos vectores que con $k = 8$ por lo tanto, las lecturas en el proceso de recuperación son:

$$mL_2 \tag{2.41}$$

Tenemos que por cada renglón realizamos m productos y $(m-1)$ sumas, así pues, las operaciones entre el vector $\overline{c}$ y la matriz $\mathbb{B}^{-1}$ son:

$$\begin{aligned} m(m) = m^2 \quad & productos \\ m(m-1) \quad & sumas \end{aligned} \tag{2.42}$$

ya que la matriz $\mathbb{B}^{-1}$ es de tamaño m x m. Además, las operaciones en la ecuación 2.42 se efectúan L_2 veces. Además si consideramos la cantidad de vectores que se adquieren de $|F|$ tenemos que: $m(m)L_2 = m(m)\frac{1}{2}\frac{|F|}{m} = m\frac{|F|}{2}$ productos y $m(m-1)L_2 = m(m-1)\frac{1}{2}\frac{|F|}{m}$ cuando m es muy grande $\sim m\frac{|F|}{2}$.

Escritura

Como sabemos, por cada vector $\bar{c}$ procesado en ecuación 2.42, se recupera un vector $\bar{b}$ del archivo original, por lo tanto, las escrituras que se efectúan para recuperar el archivo original son:

$$mL_2 \tag{2.43}$$

En la tabla 2.9 se presentan de forma comparativa las operaciones de IDA con k=8 y k=16.

	k = 8	**k = 16**
Dispersión		
Lectura	$L_1 = \frac{F}{m}$	$L_2 = \frac{1}{2}L_1$
Operaciones aritméticas		
Adiciones Productos	$\sim n\|F\|$ $n\|F\|$	$\sim n\frac{\|F\|}{2}$ $n\frac{\|F\|}{2}$
Escritura	nL_1	nL_2
Recuperación		
Lectura	mL_1	mL_2
Operaciones aritméticas		
Adiciones Productos	$\sim m\|F\|$ $m\|F\|$	$\sim m\frac{\|F\|}{2}$ $m\frac{\|F\|}{2}$
Escritura	mL_1	mL_2

Tabla 2.9: Comparación de operaciones entre IDA8 e IDA16

Capítulo 3

Implementación de IDA sobre $GF(2^8)$ y $GF(2^{16})$

Resumen

En este capítulo nos concentramos en la implementación de IDA, programado en lenguaje C, teniendo en cuenta las particularidades del lenguaje, interpretación de los polinomios primitivos, los casos a considerar para generar las tablas LD y AD, tratamiento del relleno, así como los problemas encontrados y solucionados en la implementación del algoritmo.

Por lo que se refiere al manejo de matrices y vectores utilizados en la implementación de IDA, fueron creados de forma dinámica, para proporcionar flexibilidad en la combinación de parámetros n y m.

En lo que resta de este capítulo, utilizaremos la convención que se observa en tabla 3.1. Además en la figura 3.1 se muestra un diagrama para ejemplificar el proceso de dispersión.

3.1. Implementación del proceso de dispersión

El proceso de dispersión se realiza proporcionando como parámetros el número de dispersos N, el umbral o número mínimo con el cual podemos recuperar el archivo M, el número k, que define el orden del campo ($GF(2^k)$), donde k puede ser 8 o 16, y el nombre del archivo.

Símbolo teórico	Símbolo en Código	Descripción
n	N	Número de dispersos
m	M	Umbral
$GF(2^8)$ ó $GF(2^{16})$	k	8 ó 16
F	*nombre_archivo*	Nombre del archivo a dispersar
LA	Tabla_Alphas	Tabla de logaritmo discreto
LD	LAlphas	Tabla de antilogaritmo discreto
Matriz $\mathbb{A}$	MatrizTransf	Matriz de transformación
Matriz $\mathbb{B}$	MatrizTransf	Matriz de reconstrucción en el proceso de recuperación

Tabla 3.1: Nomenclatura para utilizar en la implementación

```
1    N=atoi(argv[1]);
2    M=atoi(argv[2]);
3    k=atoi(argv[3]);
4    campo=pow(2,k)-1;
```

Listado 3.1: Parámetros N, M, k y campo

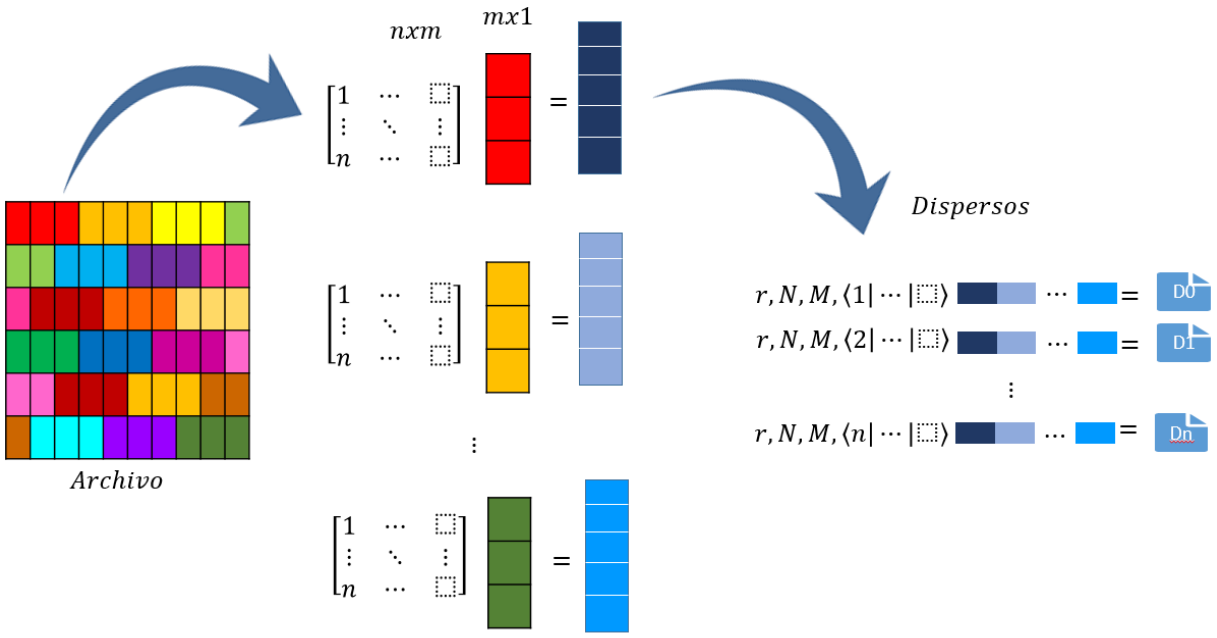

Figura 3.1: Diagrama del proceso de dispersión

El campo $GF(2^8)$ contiene a todas las cadenas de 8 dígitos binarios, el cual es generado a partir de su polinomio primitivo (ver sección 2.2.1):

$$\pi(x) = x^8 + x^6 + x^5 + x^4 + 1 \tag{3.1}$$

representado por el número $369_{10} = 101110001_2$. En tanto, el campo $GF(2^{16})$ contiene todas las cadenas de 16 dígitos binarios y su polinomio primitivo es:

$$\pi(x) = x^{16} + x^{12} + x^3 + x + 1 \tag{3.2}$$

representado por el número $69643_{10} = 10001000000001011_2$.

En el listado 3.2 se utiliza el número que codifica, en base 10, al polinomio primitivo que corresponde con cada uno de los campos que se soportan en este trabajo, además , se define *alpha_base* para la generación de las tablas de logaritmo y antilogaritmo discreto como variable global.

```
1 void GeneraCampo(unsigned short int k)
2 {
```

```
3     unsigned int primitivo; //Polinomio primitivo de cada campo
4
5     switch(k)
6     {
7       case 8:
8         primitivo=369;
9         alpha_base=primitivo&(unsigned int)campo;
10        break;
11      case 16:
12        primitivo=69643;
13        alpha_base=primitivo&(unsigned int)campo;
14        break;
15    }
16  }
```

Listado 3.2: Definición del polinomio primitivo y base_alfa

Además, se calcula el tamaño del campo como $2^k - 1$ (ver listado 3.1) y se genera una matriz de transformación (ver sección 2.2.5) de tamaño N x M en el listado 3.3.

```
1   void GeneraVandermonde(void)
2   {
3     int i,j,potencia=0;
4     MatrizTransf=Crear_Matriz(N,M);
5     for(i=0;i<N;i++){
6       for(j=0;j<M;j++){
7         MatrizTransf[i][j]=((unsigned long int)(pow((i+1),j)))%campo;
8         }
9     }
10  }
```

Listado 3.3: Matriz de transformación

La función *pow* es una función perteneciente a la biblioteca Math.h, la cual calcula x^y, donde x es el número base, y y es el exponente. Retorna un valor numérico de tipo *double*, es decir, representa valores que contienen parte fraccionada de doble precisión. Sin embargo, para asegurar que los resultados obtenidos en la matriz de transformación sean de tipo entero, se realiza un *casting*, esto es, convertir el dato *double* a tipo entero como podemos observar en el renglón 7 del listado 3.3.

3.1.1. Creación del campo finito

Se generan las tablas de logaritmo discreto *Tabla_Alphas* y antilogaritmo discreto *LAlphas* para el campo finito correspondiente como en el ejemplo 2.2.2, de tal modo que el tiempo de acceso a cualquier entrada de las tablas es constante y equivale al tiempo de acceso a los elementos de un vector de tamaño fijo.

El siguiente código muestra la implementación de las tablas generadas a partir del polinomio primitivo y el parámetro k, que determina el orden del campo.

```
void Generar_Alphas(unsigned short int tam_camp)
{
  unsigned int indice=0,x=0;
  Tabla_Alphas[indice]=indice;
  LAlphas[indice]=indice;
for(indice=0;indice<campo;indice++)
  {
    if(indice<tam_camp){
      Tabla_Alphas[indice]=(pow(2,indice));
      LAlphas[Tabla_Alphas[indice]]=indice;
      }
    else{
      if(indice==tam_camp)
      {
        x=alpha_base;
        Tabla_Alphas[indice]=alpha_base;
        LAlphas[alpha_base]=indice;
      }
      else
      {
        x<<=1;
        if(x<=campo){
            Tabla_Alphas[indice]=x;
            LAlphas[x]=indice;
        }
          else{
            x=x&(unsigned int)(campo);
            x=x^alpha_base;
            Tabla_Alphas[indice]=x;
            LAlphas[x]=indice;
```

```
31              }
32            }
33          }
34      }
35  }
```

Listado 3.4: Generación de las tablas de logaritmo y antilogaritmo discreto

Como vimos en la sección 2.2.1, las entradas de la tabla corresponden con potencias de α. En el listado 3.4 existen 3 casos a considerar para la implementación:

1. Las primeras k entradas, después del 0, son potencias sucesivas de α que pueden entenderse con corrimientos a la izquierda, o bien, k desplazamientos sucesivos a la izquierda del valor inicial (1).

2. Específicamente, la potencia de α correspondiente al orden del campo (α^k) se obtiene del polinomio primitivo $\pi(x)$ llamado *alpha_base*, del cual es la base para generar las demás potencias de α.

3. Las siguientes potencias de α ya no pueden expresarse como una palabra de k dígitos, pero son congruentes con un polinomio de k dígitos que se obtiene nuevamente de reducir en módulo $\pi(x)$ como se ilustra en el ejemplo 2.2.3.

Creado el campo finito, procedemos a la dispersión del archivo, dependiendo del valor de los parámetros N, M, k.

3.1.2. Implementación en $GF(2^8)$

Primeramente, se crea un vector dinámico llamado *BufferArchivo*, donde se alojarán M elementos, cada uno de 8 o 16 dígitos, según sea el orden del campo sobre el cual se trabaja, y leídos en orden secuencial desde el archivo que será procesado; para el caso de $GF(2^8)$, cada elemento de *BufferArchivo* es de 1 byte, que es igual a 8 dígitos binarios. Por esa razón, reservamos memoria para M elementos de tipo *char*, siendo *BufferArchivo* el vector que, en el capítulo previo, denotamos como $\overline{b_i}$ (Ver ecuación 2.28).

Al igual que el vector *BufferArchivo*, se crea otro vector dinámico llamado *VectorDispersos* de N elementos, donde cada elemento es un apuntador a un disperso, cabe señalar, que cada elemento de este vector, será un archivo nombrado como

$D0, D1, D2, ...DN-1$. Al principio de cada disperso, se escribirá un espacio en blanco, el valor de N, el valor de M y el renglón correspondiente de la matriz de transformación. Como puede observarse en la figura 3.1.

```
void Dispersa8(char *file)
{
  FILE *in,**VectorDispersos;
  int i,j,regLeidos=-1,regEscritos,bandera=0;
  unsigned char valor;
  char BufferNombre[10];
  unsigned char* BufferArchivo;
  unsigned int aux=0, Simbolos=0;

  if((BufferArchivo=(char*)malloc(M*sizeof(char)))==NULL)
    printf("No se pudo crear el buffer del archivo");

  VectorDispersos=malloc(N*sizeof(FILE*));

    if(VectorDispersos==NULL){
    printf("No se pudieron crear apuntadores a archivo\n");
    return;
  }
  for(i=0;i<N;i++)//Nombrando cada disperso
  {
    sprintf(BufferNombre,"D%i",i);
    VectorDispersos[i]=fopen(BufferNombre, "wb");
        if(VectorDispersos[i]==NULL){
          printf("No se puede crear el disperso\n");
          return;
        }
  }
  if((in=fopen(file, "rb"))==NULL){//Abriendo archivo a transformar
    printf("no se puede abrir o el archivo no existe\n");
       exit(1);
   }

   for(i=0;i<N;i++) //Escribiendo al principio de cada disperso
   {
    valor=48;
    regEscritos=fwrite(&valor,sizeof(char),1,VectorDispersos[i]);
```

```
37        valor=N;
38        regEscritos=fwrite(&valor,sizeof(char),1,VectorDispersos[i]);
39        valor=M;
40        regEscritos=fwrite(&valor,sizeof(char),1,VectorDispersos[i]);
41        for (j=0; j<M; j++){
42        valor=MatrizTransf[i][j];
43          regEscritos=fwrite(&valor,sizeof(char),1,VectorDispersos[i]);
44          }
45       }
46       //Extrayendo datos del archivo a dispersar
47       do{
48        for(i=0;i<M;i++)
49          BufferArchivo[i]=0;
50        bandera=0;
51        for(i=0;i<M && regLeidos!=0;i++){
52          valor=0;
53          regLeidos=fread(&valor,sizeof(char),1,in);
54          if(regLeidos!=0){
55              BufferArchivo[i]=abs(valor);
56              Simbolos++;
57              bandera++;
58          }
59        }
60        if(bandera!=0){
61          /*GENERAR DISPERSOS*/
62            for(i=0;i<N;i++){
63             for(j=0;j<M;j++){
64                aux=Suma_Resta(aux,
65             Multiplicacion(MatrizTransf[i][j],BufferArchivo[j]));
66             }
67          valor=aux;
68          regEscritos=fwrite(&valor,sizeof(char),1,VectorDispersos[i]);
69          aux=0;
70         }
71        }
72      }while(regLeidos!=0);
73
74        valor=Simbolos%M;
75      for(i=0;i<N;i++){
76        rewind(VectorDispersos[i]);
```

```
77        regEscritos=fwrite(&valor,sizeof(char),1,VectorDispersos[i]);
78      }
79
80       fclose(in);
81       for(i=0;i<N;i++)//Cierre de archivo de cada disperso
82        fclose(VectorDispersos[i]);
83       free(VectorDispersos);
84       free(BufferArchivo);
85   }
```

Listado 3.5: Implementación para realizar operaciones entre *MatrizTransf* y *BufferArchivo*

El listado 3.5 muestra las operaciones entre *MatrizTransf* y *BufferArchivo* como lo muestra la ecuación 2.31, específicamente, de la línea 47 a la 72, se extraen los datos del archivo a transformar, posteriormente se almacenan en el vector *BufferArchivo* para ser multiplicado por la matriz de transformación, dando lugar a la creación del vector ya transformado. Este proceso se repite hasta que se terminen los datos del archivo a dispersar

A partir de la línea 74 del algoritmo, calculamos el relleno con módulo M, esto es, si el tamaño del archivo de entrada no es un múltiplo entero de M, entonces la última instancia del vector *BufferArchivo* se completa con r elementos iguales a 0. Ya que el relleno se calcula después de haber recorrido el archivo a dispersar, se deja un espacio en blanco al principio de cada disperso para posteriormente colocar el relleno al rebobinar cada disperso (línea 76).

Por último se cierran cada uno de los dispersos, se cierra el archivo original y se libera memoria de cada una de las estructuras creadas. Al finalizar la ejecución, obtenemos como salida los N dispersos llamados $D0, D1, D2, D3, ..., DN-1$.

3.1.3. Implementación en $GF(2^{16})$

En cuanto a la implementación de IDA sobre $GF(2^{16})$, de igual manera se utiliza un vector denominado *BufferArchivo*, sin embargo, a diferencia de la implementación estudiada en la sección previa, en este caso cada símbolo extraído es una cadena de 16 dígitos binarios, en otras palabras, 2 bytes por cada uno de los M elementos que componen al vector. Por tal motivo, utilizamos el dato *unsigned short int*, para asegurar que trabajamos un dato entero de 2 bytes sin signo, o bien, 2 tipos de datos *char*.

Dado que todas las operaciones serán calculadas en tamaño de 2 bytes, las tablas *Tabla_Alphas* y *LALphas* son generadas con un total de 2^k entradas, concretamente para

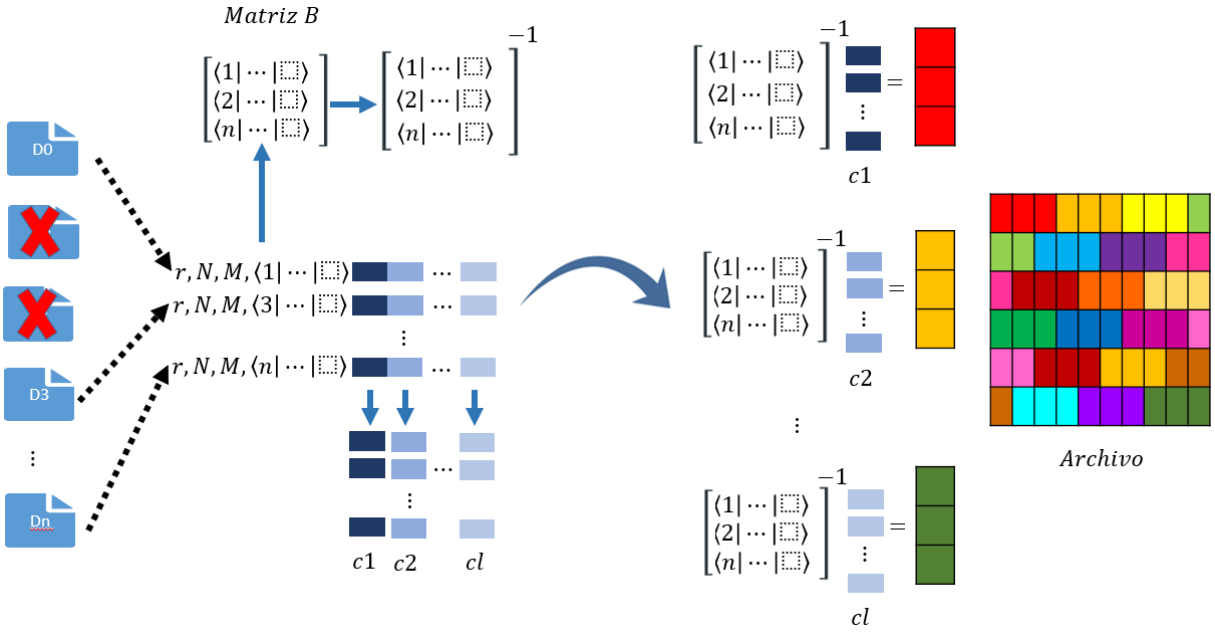

Figura 3.2: Diagrama del proceso de recuperación

el caso de $GF(2^{16})$, serán 65,536 entradas.

3.2. Implementación del proceso de recuperación

Para el proceso de recuperación se requiere como parámetros el nombre del archivo a generar, el parámetro k, que determina el orden del campo, ya sea 8 ó 16, y el nombre de los dispersos con los cuales queremos recuperar el archivo, es decir, los M dispersos sobrevivientes. Cabe mencionar que los dispersos $D1, D2, D3, ..., DM$ pueden ser escritos en cualquier orden.

Antes que nada, se extraen los parámetros N, M y r que colocamos al principio de cada uno de los dispersos (ver figura 3.2). En consecuencia, podemos crear la matriz de transformación, verificar el número mínimo de dispersos requeridos, crear el *BufferArchivo*, *VectorDispersos*, etc. de acuerdo con el campo que se trabaja.

De igual manera, para generar el campo en $GF(2^8)$ ó $GF(2^{16})$, se sigue el mismo procedimiento que en la lista 3.2 y 3.4. Por otra parte, se crea la matriz $\mathbb{B}$ aumentada llamada *MatrizTransf* de tamaño $M \times 2M$. Esto es, dado que la matriz $\mathbb{B}$ se construye de

los M renglones sobrevivientes, se puede adecuar para acoplar la matriz identidad $\mathbb{I}$, dando como resultado una matriz de tamaño $M \times 2M$.

Con esto, logramos facilitar el proceso de inversión por el método de Gauss-Jordan, el cual se detalla en el listado 3.6.

```
void Invertir_Gauss(void)
{
  unsigned int aux[M*2];//Vector auxiliar para realizar operaciones
  unsigned int i=0,j=0,renglon=0,columna=0,pivote_renglon;

  for(j=0;j<M;j++)
  {
    if(MatrizTransf[renglon][columna]!=1)//Si no es pivote
    {
      Dividir_Renglon_G(renglon,MatrizTransf[renglon][columna]);
    }

    pivote_renglon=renglon; //Marcar como pivote
    for(i=0;i<M;i++)
    {
      if(i!=j) //Si no es la diagonal
      {
        if(MatrizTransf[i][j]!=0) //Si no es un cero
        {
        Constante_Renglon_G(pivote_renglon,MatrizTransf[i][j],aux);
        Suma_Resta_Renglon_G(i,aux);
        }
      }
    }
  renglon++;
  columna++;
  }
}
```

Listado 3.6: Implementación del método Gauss-Jordan para inversión de matrices

Como resultado, obtendremos la matriz inversa de $\mathbb{B}$ del lado derecho de la matriz de transformación aumentada.

A continuación, creamos 4 vectores de forma dinámica: *BufferArchivo*, *BufferArchivo2*,

VectorDispersos, *DatRecuperados* que se utilizan como se indican a continuación:

- Vector *VectorDispersos*: Vector que contiene M apuntadores de tipo archivo. Estos son utilizados para acceder a la información de cada disperso.
- Vector *BufferArchivo*: Vector donde se alojan M elementos de 8 o 16 dígitos binarios respectivamente.
- Vector *BufferArchivo2*: Vector auxiliar en la extracción de datos al igual que el vector BufferArchivo.
- Vector *DatRecuperados*: Vector donde se almacena los datos recuperados.

3.2.1. Recuperación sobre $GF(2^8)$

Como hemos dejado claro, para el manejo del campo $GF(2^8)$ se utilizan 8 bits, por consiguiente, se utiliza el tipo de dato primitivo *char* sin signo para asegurar que sólo se trabaje con 1 byte por cada dato.

Ya que definimos los vectores, mostramos el proceso de recuperación de un archivo a partir de sus M dispersos en el listado 3.7.

```
1
2  void Recupera8(char **argv)
3  {
4    FILE *out;
5    int i,j,regLeidos,regLeidos2,regEscritos,
6        primera=1,q=0,k,l,bandera=0;
7    unsigned char valor,valor2;
8    unsigned char* BufferArchivo,*DatRecuperados,* BufferArchivo2;
9    unsigned int aux=0,bas;
10
11   MatrizTransf=Crear_Matriz(M,M*2);
12
13   //Creando vector Dispersos
14   VectorDispersos=malloc(M*sizeof(FILE*));
15     if(VectorDispersos==NULL){
16     printf("No se pudieron reservar apuntadores a dispersos\n");
17     exit(1);
18   }
```

```
19
20    for(i=0;i<M;i++){
21      VectorDispersos[i]=fopen(argv[i+3],"rb");
22      if(VectorDispersos[i]==NULL){
23        printf("No puedo abrir algun disperso %i\n",i);
24        exit(1);
25      }
26    }
27
28    for(i=0;i<M;i++){/*Reconocer r,N,M de cada disperso*/
29      regLeidos=fread(&bas,sizeof(char),1,VectorDispersos[i]);
30      regLeidos=fread(&bas,sizeof(char),1,VectorDispersos[i]);
31      regLeidos=fread(&bas,sizeof(char),1,VectorDispersos[i]);
32    }
33      //Recupera el renglon de la matriz de dispersion.
34    for(i=0;i<M;i++){
35      for(j=0;j<M;j++){
36        regLeidos=fread(&valor,sizeof(char),1,VectorDispersos[i]);
37        MatrizTransf[i][j]=valor;
38      }
39    }
40
41    for(i=0;i<M;i++){//Acoplar matriz identidad
42      for(j=M;j<M*2;j++){
43        if(i+M==j)
44          MatrizTransf[i][j]=1;
45        else
46          MatrizTransf[i][j]=0;
47      }
48    }
49
50    Invertir_Gauss(); //Invertir por metodo de Gauss-Jordan
51
52    out=fopen(argv[1],"wb"); //Abriendo archivo de escritura
53    if(out==NULL)
54    {
55      printf("No se puede crear el archivo de salida\n");
56      exit(1);
57    }
58
```

```
59          /*Creando Buffers*/
60      if((BufferArchivo=(char*)malloc(M*sizeof(char)))==NULL)
61        printf("No se pudo crear el buffer del archivo");
62      if((BufferArchivo2=(char*)malloc(M*sizeof(char)))==NULL)
63        printf("No se pudo crear el buffer del archivo");
64      if((DatRecuperados=(char*)malloc(M*sizeof(char)))==NULL)
65        printf("No se pudo crear el buffer del archivo");
66
67      //Obteniendo datos de cada disperso
68      for(i=0;i<M && regLeidos!=0;i++)
69        {
70          regLeidos=fread(&valor,sizeof(char),1,VectorDispersos[i]);
71          if(regLeidos!=0){
72              BufferArchivo[i]=valor;
73              bandera++;
74          }
75        }
76
77      //Realizar operaciones entre Matriz de dispersion y
78      //vector de los elementos de cada disperso
79      for(i=0;i<M;i++){
80        for(j=M;j<M*2;j++){
81          aux=Suma_Resta(aux,
82          Multiplicacion(MatrizTransf[i][j],BufferArchivo[q]));
83          q++;
84          }
85          DatRecuperados[i]=aux;
86          q=0;
87          aux=0;
88      }
89      //Obteniendo datos de cada disperso
90      for(i=0;i<M && regLeidos!=0;i++)
91      {
92          regLeidos2=fread(&valor2,sizeof(char),1,VectorDispersos[i]);
93          if(regLeidos2!=0){
94              BufferArchivo2[i]=valor2;
95              bandera++;
96        }
97      }
98
```

```
99      do{
100       if(regLeidos2!=0){
101         for(i=0;i<M;i++){
102           valor=DatRecuperados[i];
103           regEscritos=fwrite(&valor,sizeof(char),1,out);
104         }
105         for(i=0;i<M;i++){
106             BufferArchivo[i]=BufferArchivo2[i];
107         }
108         //Realizando operaciones
109         for(i=0;i<M;i++){
110           for(j=M;j<M*2;j++){
111             aux=Suma_Resta(aux,Multiplicacion(
112             MatrizTransf[i][j],BufferArchivo[q]));
113             q++;
114           }
115             DatRecuperados[i]=aux;
116             q=0;
117             aux=0;
118         }
119
120         for(i=0;i<M && regLeidos2!=0;i++)
121         {
122         regLeidos2=fread(&valor2,sizeof(char),1,VectorDispersos[i]);
123         if(regLeidos2!=0){
124             BufferArchivo2[i]=valor2;
125             bandera=0;
126           }
127         }
128       }
129     }while(regLeidos2!=0);
130
131     valor=0;      /*Casos del relleno*/
132     if(Relleno==0)
133     {
134       for(i=0;i<M;i++){
135         valor=0;
136         valor=DatRecuperados[i];
137         regEscritos=fwrite(&valor,sizeof(char),1,out);
138       }
```

```
139     }
140
141     if(Relleno!=0)
142     {
143       for(i=0;i<Relleno;i++){
144         valor=0;
145         valor=DatRecuperados[i];
146         regEscritos=fwrite(&valor,sizeof(char),1,out);
147       }
148     }
149
150     for(i=0;i<M;i++)//Cierre de archivos
151       fclose(VectorDispersos[i]);
152
153     fclose(out);
154     free(BufferArchivo);//Liberacion de memoria
155     free(DatRecuperados);
156     free(VectorDispersos);
157     BufferArchivo==NULL;
158     DatRecuperados==NULL;
159     VectorDispersos==NULL;
160   }
```

Listado 3.7: Implementación del proceso de recuperación sobre $GF(2^8)$

Para empezar, de la línea 66 a la 73 se toma un elemento de cada disperso y se coloca en el *BufferArchivo*, seguidamente se realizan las operaciones correspondientes con la inversa de la *MatrizTransf*, para obtener en el vector *DatRecuperados* los primeros datos recuperados del archivo original.

A continuación, se extraen los siguientes elementos de cada disperso y se colocan en el *BufferArchivo2*. Esto con la finalidad de verificar anticipadamente con el vector *BufferArchivo2* si aún hay elementos que extraer. Si es el caso en el cual todavía existen elementos que procesar en cada uno de los dispersos, el vector *DatRecuperados* se escribe en el archivo de salida, se realiza una copia del *BufferArchivo2* al *BufferArchivo* y se realizan las operaciones correspondientes hasta obtener el vector *DatRecuperados*. Este proceso se repite hasta que *BufferArchivo2* no complete su vector o ya no haya más elementos que extraer de cada uno de los dispersos (línea 129).

Existen 2 casos para el tratamiento del relleno:

Caso 1: Si no hay relleno

Se asume que todos los elementos del vector *DatRecuperados* se escribirán en el archivo reconstruido.

Caso 2: Si hay relleno

Se introduce en el archivo la cantidad de elementos del vector *DatRecuperados* que corresponde a la diferencia entre M y la variable r, que corresponde con el número de ceros de relleno utilizados en el algoritmo inicial.

3.2.2. Recuperación sobre $GF(2^{16})$

Para el caso de recuperación sobre el campo $GF(2^{16})$ se consideran datos primitivos *unsigned short int* (entero corto sin signo) para asegurar que cada elemento, o símbolo manipulado, tiene una longitud de 16 dígitos binarios.

Un aspecto a considerar al recuperar la información sobre el campo $GF(2^{16})$ son los bits sobrantes en el último vector *DatRecuperados*, si se insertara un número menor a 255, este sólo ocuparía 8 bits reales del archivo recuperado y por tanto, los bits más significativos se rellenarían con 0.

Esto trae como consecuencia que al insertar el vector completo de 16 bits, obtengamos un archivo con 1 byte de más, y por consiguiente, no sea el mismo archivo recuperado.

La solución para este tipo de problema es verificar en el último vector, si el número recuperado es menor o igual a 255, en caso de ser afirmativo, se procede sólo a insertar 8 bits, de lo contrario, se insertarán 16 bits en el archivo recuperado.

3.3. Problemas resueltos

La finalidad de esta sección es abordar el problema del final del archivo *EOF (End Of File)* en cada uno de los dispersos.

El EOF es una marca que indica la finalización de un archivo, regularmente éste es marcado como un valor de -1 en lenguaje C bajo linux, sin embargo, en la implementación realizada, se utilizaron tipos de datos primitivos sin signo, ya sea *unsigned char* para $GF(2^8)$ y *unsigned short int* para $GF(2^{16})$.

Esto trae como consecuencia que, al verificar el final del archivo de cada uno de los dispersos, éste no sea interpretado como la marca EOF, es decir, no es interpretado como el valor de -1, si no que es interpretado como un valor de 0 y como resultado, cada valor EOF se interpreta como parte del archivo original, ya que al utilizar la sentencia unsigned, aseguramos que cada valor tomado sea considerado un número sin signo.

Podemos reconocer este error si comparamos el número de bytes del archivo original, con el número de bytes del archivo recuperado, salta a la vista que el archivo recuperado, tiene más bytes que el archivo original, por tanto, ha ocurrido un problema de pérdida de integridad.

Para dar solución a este problema, se implementaron 2 vectores, el primer vector llamado *BufferArchivo2* en el que sólo se dedica a leer un elemento de cada disperso y almacenarlo, para posteriormente proporcionarle los datos al segundo vector *BufferArchivo* que se encarga sólo de realizar las operaciones correspondientes para generar los datos recuperados del archivo.

Dado que el primer vector *BufferArchivo2* puede anticipar el último vector, este no es considerado y se descarta. De manera que sólo obtenemos el penúltimo vector que contiene los últimos datos del archivo a recuperar, para posteriormente hacer la verificación del relleno.

Capítulo 4

Resultados

Resumen

En este capítulo se presentan los resultados que corresponden con 5 familias de experimentos: la primera, se refiere a un conjunto de experimentos de base entre los campos $GF(2^8)$ y $GF(2^{16})$, el resto de las familias establecen un contraste con los resultados iniciales. En cada una de estas familias restantes existe una condición de experimentación diferente, respecto del conjunto inicial como el sistema operativo, el hardware de procesamiento, la cantidad de RAM disponible, o las condiciones de lectura y escritura.

Llamaremos IDA8 a la implementación de IDA sobre $GF(2^8)$ y, de manera semejante, IDA16 a la implementación que funciona en $GF(2^{16})$.

Cada familia de experimentos está definida por todas las posibles combinaciones de los parámetros M y N, donde $2 < M \leq 28$ y $3 < N \leq 29$. Cada una de estas combinaciones tiene un identificador que la distingue, como se muestra en la tabla 4.2. Para cada combinación se ejecuta una instancia de IDA8 y otra de IDA16, por cada uno de los archivos de prueba cuyos tamaños y extensiones se listan en la tabla 4.1. Esto significa que cada familia incluye en total 4,860 experimentos diferentes, dando como resultado un total de 19,440 experimentos.

El resto de las condiciones de una familia, que denominaremos condiciones complementarias, vienen determinadas por el Sistema Operativo, el hardware de procesamiento, cantidad de RAM disponible y los mecanismos de lectura y escritura de vectores. Estas condiciones se reúnen en las tablas 4.3, 4.4, 4.5, 4.6, E.1 y E.2

Para percibir mejor los resultados y facilitar la lectura, sólo se presentan los resultados de cada familia que corresponden con el archivo más pequeño y el más grande considerados

en nuestro estudio, esto es, 1 MB y 1024 MB respectivamente. Sin embargo, en los apéndices A, B, C y D de este trabajo se muestran las gráficas de cada familia considerada.

Cabe mencionar que las implementaciones de IDA8 e IDA16 se realizaron en lenguaje C, con un compilador gcc 5.4.0; además todas las figuras presentadas en esta sección, el eje horizontal representa las combinaciones de N y M, mientras que el eje vertical representa el tiempo medido en segundos. Por último, es importante mencionar que el tiempo de ejecución de la primera familia fue de 108 horas aproximadamente.

Tamaño	Extensión
1 MB	.png
4 MB	.mp3
16 MB	.txt
64 MB	.avi
256 MB	.dat
1024 MB	.izt

Tabla 4.1: Archivos de prueba

Tabla 4.2: Instancias de IDA. Donde N: Número de dispersos, M: umbral o número mínimo con el cual podemos recuperar un archivo, C: Número de combinación

N	M	C	N	M	C	N	M	C	N	M	C	N	M	C
3	2	1	10	5	32	13	9	63	16	4	94	18	6	125
4	2	2	10	6	33	13	10	64	16	5	95	18	7	126
4	3	3	10	7	34	13	11	65	16	6	96	18	8	127
5	2	4	10	8	35	13	12	66	16	7	97	18	9	128
5	3	5	10	9	36	14	2	67	16	8	98	18	10	129
5	4	6	11	2	37	14	3	68	16	9	99	18	11	130
6	2	7	11	3	38	14	4	69	16	10	100	18	12	131
6	3	8	11	4	39	14	5	70	16	11	101	18	13	132
6	4	9	11	5	40	14	6	71	16	12	102	18	14	133
6	5	10	11	6	41	14	7	72	16	13	103	18	15	134
7	2	11	11	7	42	14	8	73	16	14	104	18	16	135
7	3	12	11	8	43	14	9	74	16	15	105	18	17	136
7	4	13	11	9	44	14	10	75	17	2	106	19	2	137
7	5	14	11	10	45	14	11	76	17	3	107	19	3	138
7	6	15	12	2	46	14	12	77	17	4	108	19	4	139
8	2	16	12	3	47	14	13	78	17	5	109	19	5	140
8	3	17	12	4	48	15	2	79	17	6	110	19	6	141
8	4	18	12	5	49	15	3	80	17	7	111	19	7	142
8	5	19	12	6	50	15	4	81	17	8	112	19	8	143
8	6	20	12	7	51	15	5	82	17	9	113	19	9	144
8	7	21	12	8	52	15	6	83	17	10	114	19	10	145
9	2	22	12	9	53	15	7	84	17	11	115	19	11	146
9	3	23	12	10	54	15	8	85	17	12	116	19	12	147
9	4	24	12	11	55	15	9	86	17	13	117	19	13	148
9	5	25	13	2	56	15	10	87	17	14	118	19	14	149
9	6	26	13	3	57	15	11	88	17	15	119	19	15	150
9	7	27	13	4	58	15	12	89	17	16	120	19	16	151
9	8	28	13	5	59	15	13	90	18	2	121	19	17	152
10	2	29	13	6	60	15	14	91	18	3	122	19	18	153
10	3	30	13	7	61	16	2	92	18	4	123	20	2	154

Continúa en la siguiente página

Continuación de la tabla

10	4	31	13	8	62	16	3	93	18	5	124	20	3	155
20	4	156	21	17	187	23	9	218	24	19	249	26	5	280
20	5	157	21	18	188	23	10	219	24	20	250	26	6	281
20	6	158	21	19	189	23	11	220	24	21	251	26	7	282
20	7	159	21	20	190	23	12	221	24	22	252	26	8	283
20	8	160	22	2	191	23	13	222	24	23	253	26	9	284
20	9	161	22	3	192	23	14	223	25	2	254	26	10	285
20	10	162	22	4	193	23	15	224	25	3	255	26	11	286
20	11	163	22	5	194	23	16	225	25	4	256	26	12	287
20	12	164	22	6	195	23	17	226	25	5	257	26	13	288
20	13	165	22	7	196	23	18	227	25	6	258	26	14	289
20	14	166	22	8	197	23	19	228	25	7	259	26	15	290
20	15	167	22	9	198	23	20	229	25	8	260	26	16	291
20	16	168	22	10	199	23	21	230	25	9	261	26	17	292
20	17	169	22	11	200	23	22	231	25	10	262	26	18	293
20	18	170	22	12	201	24	2	232	25	11	263	26	19	294
20	19	171	22	13	202	24	3	233	25	12	264	26	20	295
21	2	172	22	14	203	24	4	234	25	13	265	26	21	296
21	3	173	22	15	204	24	5	235	25	14	266	26	22	297
21	4	174	22	16	205	24	6	236	25	15	267	26	23	298
21	5	175	22	17	206	24	7	237	25	16	268	26	24	299
21	6	176	22	18	207	24	8	238	25	17	269	26	25	300
21	7	177	22	19	208	24	9	239	25	18	270	27	2	301
21	8	178	22	20	209	24	10	240	25	19	271	27	3	302
21	9	179	22	21	210	24	11	241	25	20	272	27	4	303
21	10	180	23	2	211	24	12	242	25	21	273	27	5	304
21	11	181	23	3	212	24	13	243	25	22	274	27	6	305
21	12	182	23	4	213	24	14	244	25	23	275	27	7	306
21	13	183	23	5	214	24	15	245	25	24	276	27	8	307
21	14	184	23	6	215	24	16	246	26	2	277	27	9	308
21	15	185	23	7	216	24	17	247	26	3	278	27	10	309
21	16	186	23	8	217	24	18	248	26	4	279	27	11	310
27	12	311	28	7	331	28	27	351	29	21	371	30	14	391
27	13	312	28	8	332	29	2	352	29	22	372	30	15	392
27	14	313	28	9	333	29	3	353	29	23	373	30	16	393

Continúa en la siguiente página

Continuación de la tabla															
27	15	314	28	10	334	29	4	354	29	24	374	30	17	394
27	16	315	28	11	335	29	5	355	29	25	375	30	18	395
27	17	316	28	12	336	29	6	356	29	26	376	30	19	396
27	18	317	28	13	337	29	7	357	29	27	377	30	20	397
27	19	318	28	14	338	29	8	358	29	28	378	30	21	398
27	20	319	28	15	339	29	9	359	30	2	379	30	22	399
27	21	320	28	16	340	29	10	360	30	3	380	30	23	400
27	22	321	28	17	341	29	11	361	30	4	381	30	24	401
27	23	322	28	18	342	29	12	362	30	5	382	30	25	402
27	24	323	28	19	343	29	13	363	30	6	383	30	26	403
27	25	324	28	20	344	29	14	364	30	7	384	30	27	404
27	26	325	28	21	345	29	15	365	30	8	385	30	28	405
28	2	326	28	22	346	29	16	366	30	9	386			
28	3	327	28	23	347	29	17	367	30	10	387			
28	4	328	28	24	348	29	18	368	30	11	388			
28	5	329	28	25	349	29	19	369	30	12	389			
28	6	330	28	26	350	29	20	370	30	13	390			
No. de experimentos por cada familia: 4860														

4.1. Familia de experimentos 1. IDA8 vs IDA16

Sistema Operativo	Linux/ Ubuntu 16.04 LTS
Procesador	Intel Core i7
Memoria RAM	8 GB
Lectura/Escritura	En Disco
HDD	1TB, 7200 rpm

Tabla 4.3: Condiciones complementarias para la familia de experimentos 1

En la figura 4.1 se muestran los tiempos obtenidos con IDA8 e IDA16 con un archivo de tamaño 1MB. La leyenda corresponde a 4 series: las primeras dos series corresponden al proceso de dispersión y recuperación sobre IDA8; las 2 series restantes corresponden al proceso de dispersión y recuperación sobre IDA16.

Cada una de las series corresponde con un patrón típico en "diente de sierra" en el que cada "diente" está definido por un valor específico de N y todas las posibles combinaciones de M que puede admitir, esto es, para un valor pequeño de N, sólo aplica un conjunto limitado de valores de M, por ejemplo (3,2). Mientras que, un valor grande de N puede admitir, en correspondencia, un número mayor de valores de M, por ejemplo (29,2), (29,3), ..., (29,28). Esto explica por qué, los dientes se hacen más amplios hacia la derecha de la figura, que corresponde con los valores más grandes de N.

Por otro lado, cada diente que corresponde con una operación de dispersión comienza con un valor máximo y termina en un valor mínimo, en tanto, un diente que corresponde a una operación de recuperación comienza en un mínimo y termina en un máximo. El caso de la dispersión se explica porque, para un valor fijo de N, se muestran los resultados para valores ascendentes de M. Un valor pequeño de M significa demasiada información redundante que el algoritmo debe calcular y viceversa, esto es, un valor de M cercano a N significa muy poca información redundante. El mismo razonamiento explica el comportamiento en la recuperación: un número pequeño de dispersos es suficiente para recuperar el archivo original y esto implica poco procesamiento. Un número mayor de dispersos implica un mayor costo de procesamiento.

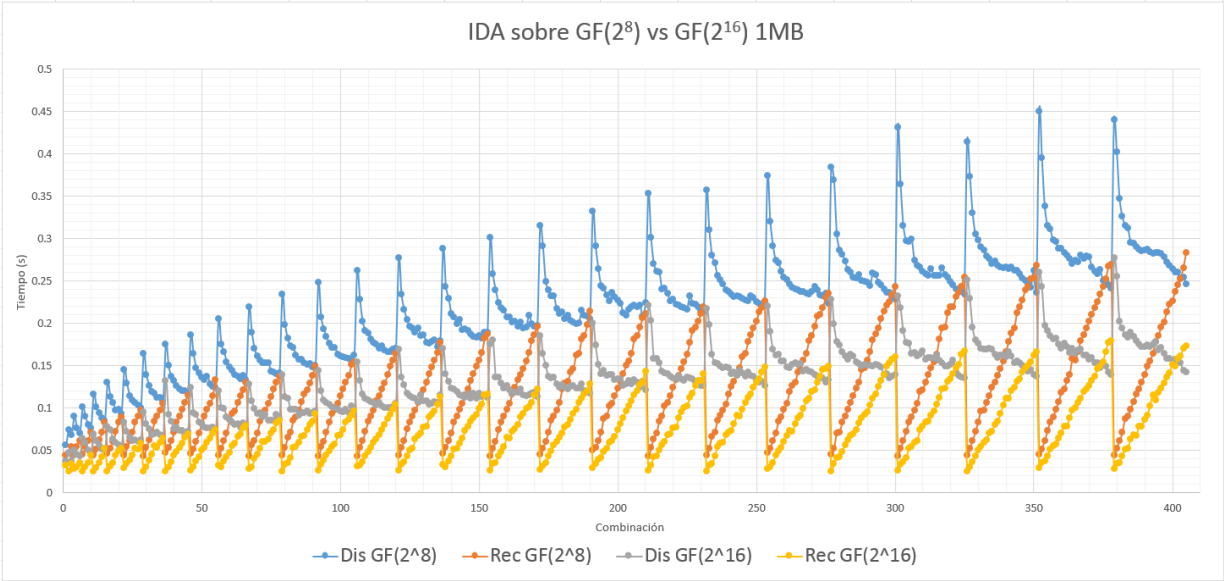

Figura 4.1: IDA8 vs IDA16 con archivo de 1MB

Posteriormente, se realizó otro experimento cambiando el tamaño del archivo a 1024MB y los resultados obtenidos se muestran en la figura 4.2.

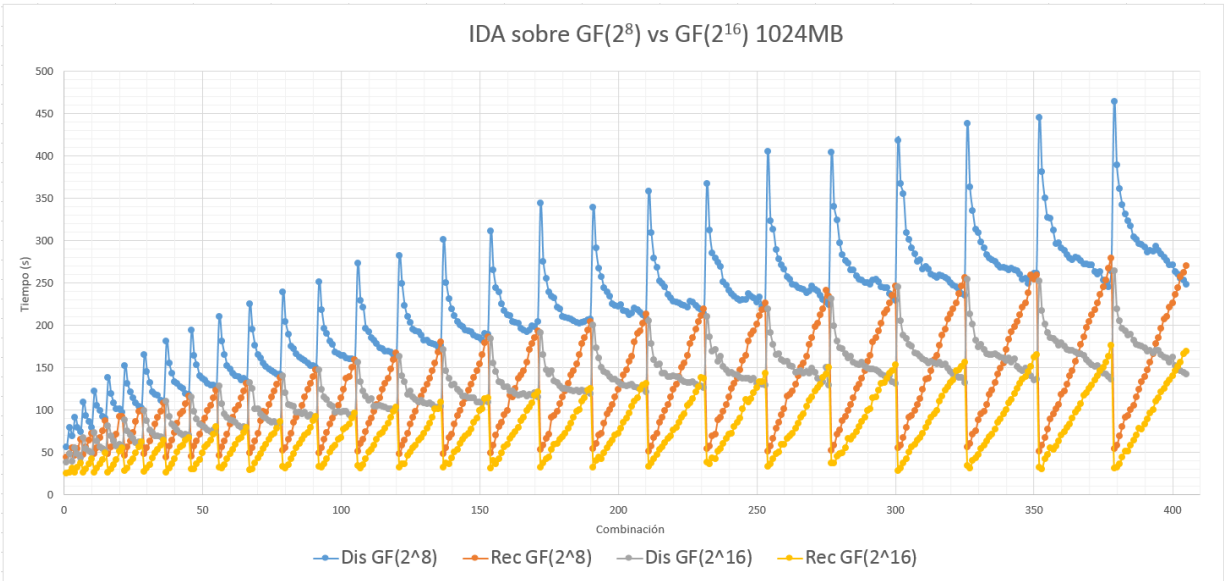

Figura 4.2: IDA8 vs IDA16 con archivo de 1024MB

Las observaciones inmediatas que pueden hacerse sobre los resultados de la primer familia son:

El tiempo de procesamiento depende directamente del tamaño del archivo, es decir, el proceso de dispersión o recuperación aumentan proporcionalmente con el tamaño del archivo original. Asimismo, sobre archivos de mayor tamaño se puede observar mejor el beneficio de usar un campo finito de mayor orden para todos los experimentos realizados.

Observamos que en la figura 4.1 utilizamos un archivo de tamaño 1MB, los tiempos mostrados del proceso de dispersión y el proceso de recuperación son pequeños, sin embargo, se puede apreciar que la diferencia entre utilizar un IDA8 e IDA16 es mínima.

Por otra parte, en la figura 4.2 utilizamos un archivo de tamaño 1024 MB, podemos percibir que el tiempo del proceso de dispersión y recuperación para ambos campos es mayor. Sin embargo, se sigue observando que el menor tiempo en el proceso lo obtiene IDA16, por tanto, para cualquier combinación de N y M y un archivo original fijo, IDA16 será siempre más rápido que IDA8, tanto en dispersión como en recuperación.

Además, el tiempo de dispersión es directamente proporcional a la cantidad de información redundante que se genera y el tiempo de recuperación es inversamente proporcional a la cantidad de información redundante que se genera.

Si revisamos la forma en como IDA realiza el proceso de dispersión sobre $GF(2^8)$ y $GF(2^{16})$, podemos decir que al ejecutar IDA sobre $GF(2^8)$ procesamos el archivo en unidades de 8 bits de longitud tanto en lectura, como en escritura; en cambio, al ejecutar IDA sobre $GF(2^{16})$, procesamos el archivo en unidades de 16 bits. Esto quiere decir que cada vez que ejecutemos IDA8 obtendremos más vectores de 8 bits, por lo tanto, realizará mas operaciones al multiplicar la matriz de transformación por el vector del archivo (ver sección 2.3.2). Así pues, si ejecutamos IDA16, obtendremos al menos la mitad de vectores a procesar que con IDA8 y por tanto, se realizarán menos operaciones y por consecuencia se reduce el tiempo de ejecución.

Con esto, queda claro que el tiempo de ejecución de los algoritmos de dispersión y recuperación depende inversamente del orden del campo con el que se trabaja. Por otro lado, cabe señalar que para archivos de menor tamaño no existe una ganancia significativa en tiempo si se trabaja en uno u otro campo, como se observa en la figura 4.1.

Además se puede contemplar que para cualquier gráfica presentada en este capítulo, el proceso de dispersión toma más tiempo que el proceso de recuperación. Esto se debe a que hay N archivos para crear, en cambio, en el proceso de recuperación, sólo se trabajan M archivos, lo cual disminuye el tiempo de procesamiento. En consecuencia podemos decir que el tiempo de procesamiento de IDA depende directamente de los parámetros M y N.

4.2. Familia de experimentos 2. Impacto del sistema operativo

Sistema Operativo	Windows
Procesador	Intel Core i7
Memoria RAM	8 GB
Lectura/Escritura	En Disco

Tabla 4.4: Condiciones complementarias para la familia de experimentos 2

En esta sección se presentan los resultados obtenidos al ejecutar IDA en un sistema operativo diferente, tal es el caso de Windows, manteniendo las mismas características de hardware que la familia de experimentos 1. En la figura 4.3 se muestran los resultados obtenidos de IDA8 vs IDA16 utilizando un archivo de 1MB.

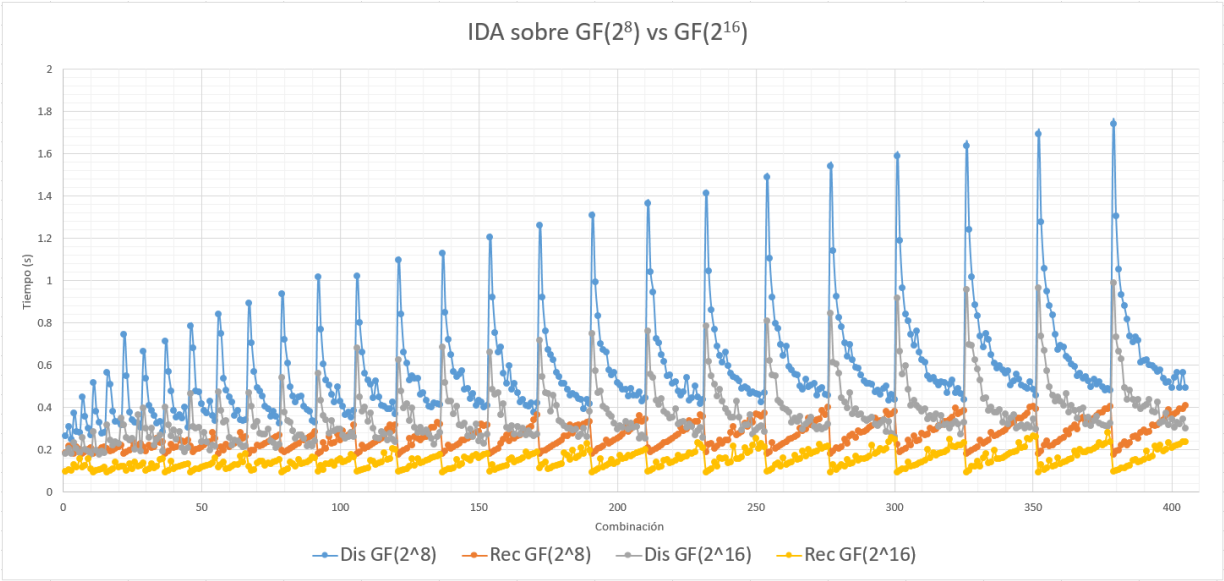

Figura 4.3: IDA8 vs IDA16 con archivo de 1MB en Windows

Por otra parte, se muestran los resultados de IDA con un archivo de 1024 MB con las mismas características mencionadas anteriormente. Los resultados se muestran en la figura 4.4.

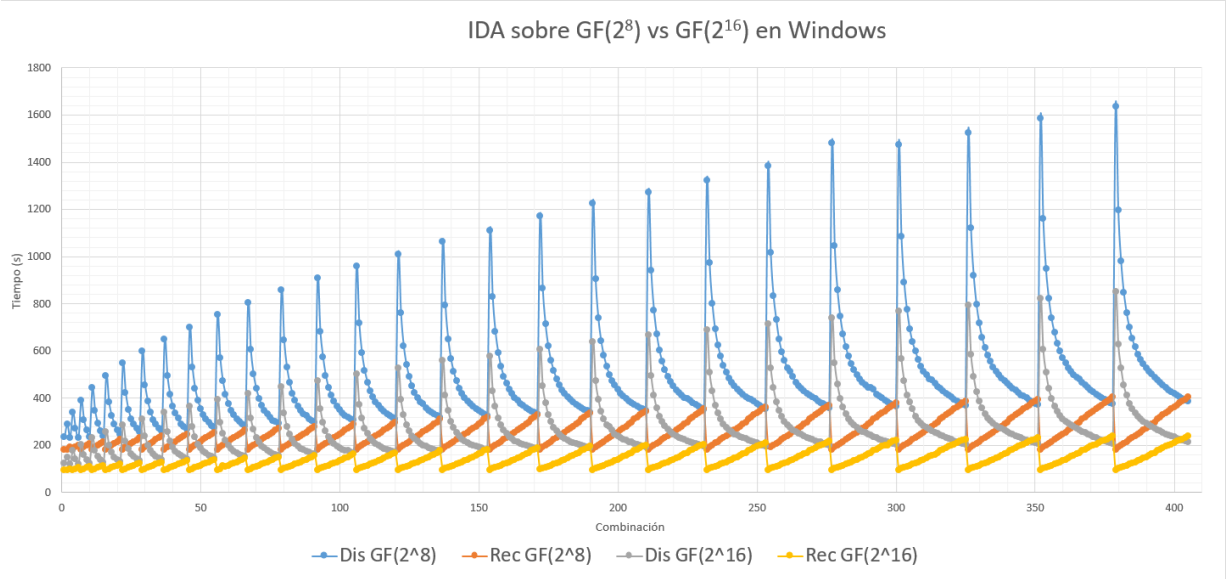

Figura 4.4: IDA8 vs IDA16 con archivo de 1024MB en Windows

Para poder establecer las primeras observaciones de estos resultados, debemos contrastar la figura 4.1 con la figura 4.3, y la 4.2 con la 4.4. De manera consistente, en cada experimento se observa que los tiempos de ejecución se multiplican por un factor aproximado de 4, cuando el Sistema Operativo cambia de Linux a Windows.

IDA bajo el sistema operativo Linux tiene un tiempo de procesamiento menor que en sistemas Windows tanto para la dispersión como la recuperación. La diferencia entre la familia de experimentos 1 y 2 puede explicarse por el sistema de archivos que utiliza cada sistema operativo. Windows utiliza el sistema de archivo NTFS (New Technology FileSystem) mientras que Linux utiliza Ext4 (Extended FileSystem version 4). Ambos sistemas se manejan por una técnica denominada bitácora o "journaling". La idea básica es mantener un registro antes de que se realice la acción en el sistema de archivos, esto con la finalidad de evitar errores y evitar que se corrompa el sistema de archivos, ya que éste debe ser altamente confiable [34].

Sin embargo, NTFS ordena y accede a los datos secuencialmente, es decir, al intentar escribir un archivo en un disco duro, los datos son escritos en bloques consecutivos como lo muestra la figura 4.5a). En cambio Ext4 ordena y accede a los archivos en medio del disco duro como lo muestra la figura 4.5b). Esto trae como consecuencia la reducción de la distancia del movimiento del brazo del disco, ya que no es lo mismo mover el brazo hasta el final del último bloque, que mover el brazo sólo a la mitad del disco y, por consiguiente,

se reduce el tiempo de acceso a archivos. Además Ext4 reduce la fragmentación debido a la forma en que se seleccionan los bloques de disco. En cambio, en NTFS se crean huecos producidos por el borrado de bloques a lo largo del tiempo. Estos huecos se extienden por todo el disco y tienen impacto sobre el rendimiento del dispositivo. [35].

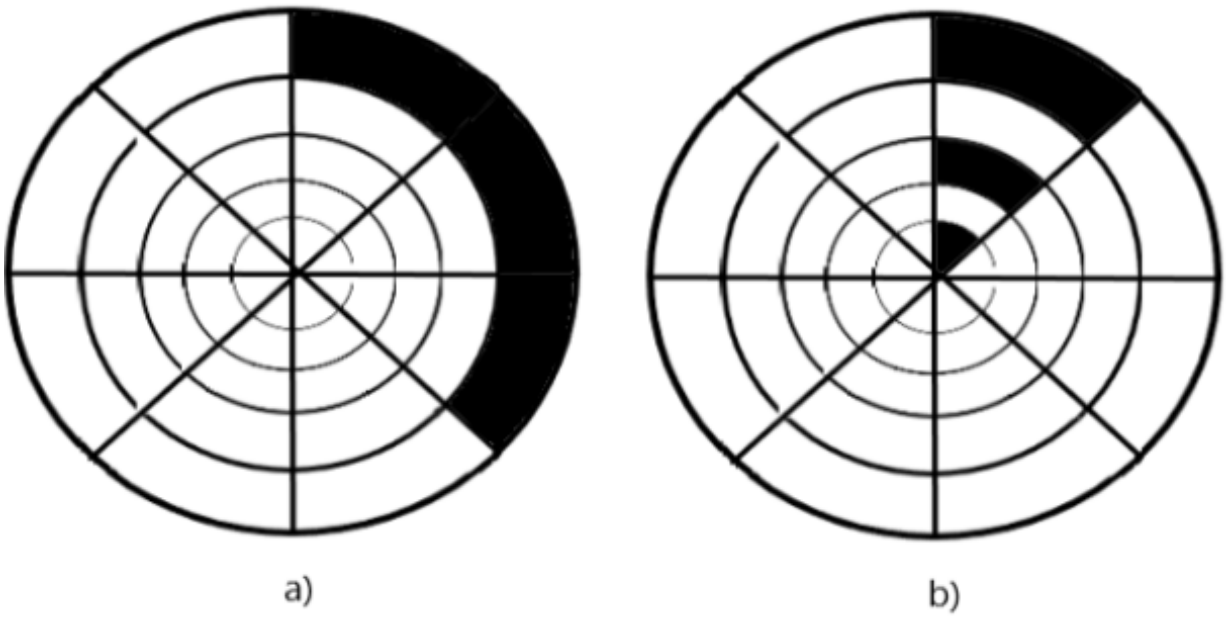

Figura 4.5: a) Datos escritos en bloques consecutivos. b) Datos escritos en medio del disco

4.3. Familia de experimentos 3. Impacto del hardware de procesamiento

Sistema Operativo	Linux/ Ubuntu 16.04 LTS
Procesador	Intel Atom
Memoria RAM	2 GB

Tabla 4.5: Condiciones complementarias en Linux con procesador Atom

Los resultados que se muestran en esta sección corresponden a los experimentos realizados con IDA en una máquina con características limitadas, como el procesador y la memoria RAM. Los resultados que se muestran en la figura 4.6 corresponden a la ejecución de IDA8 vs IDA16 con un archivo de 1MB.

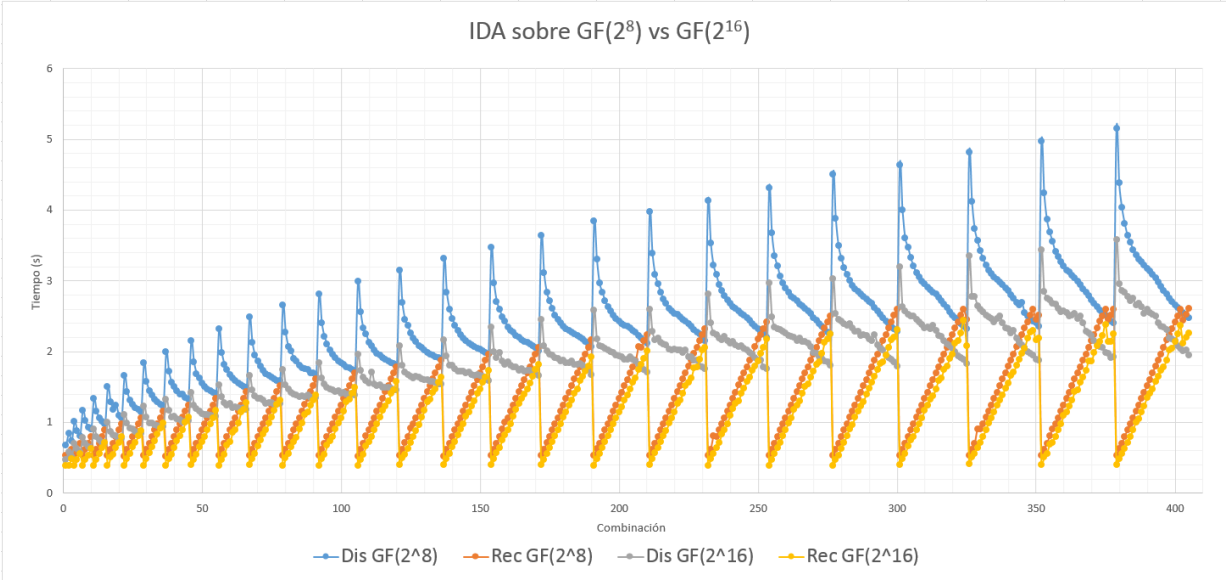

Figura 4.6: IDA8 vs IDA16 con archivo de 1MB con procesador Atom

En la figura 4.7 se muestran los resultados obtenidos con un archivo de 64MB, debido a que la máquina no soportó procesar un archivo de mayor tamaño.

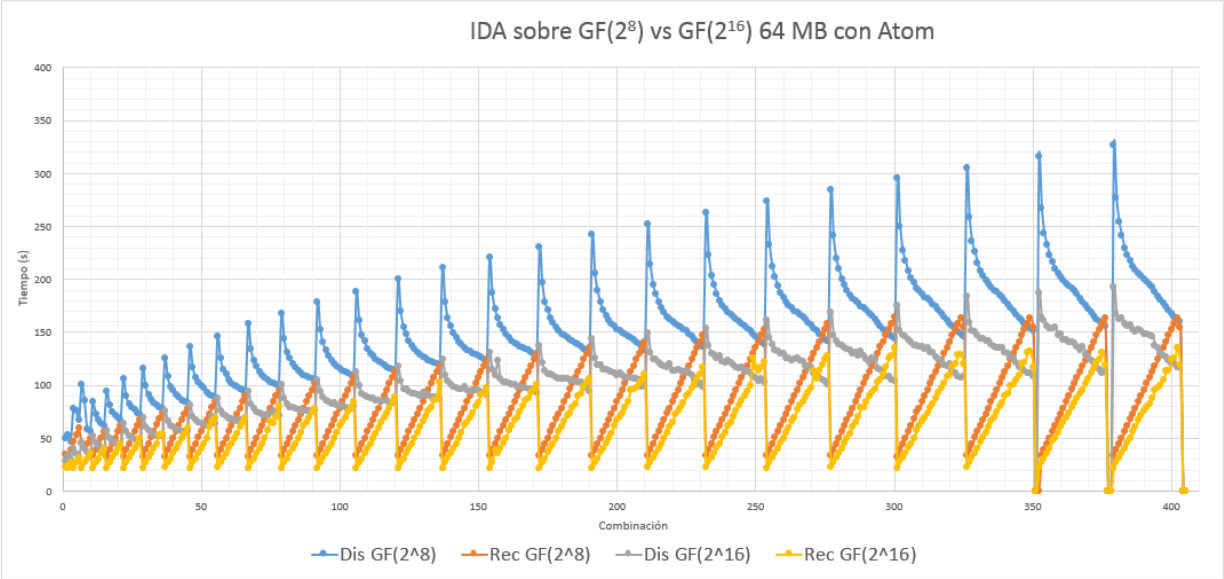

Figura 4.7: IDA8 vs IDA16 con archivo de 64MB con procesador Atom

Las gráficas que representan a esta familia muestran que el tiempo de procesamiento es al menos 10 veces mayor a los tiempos que corresponden a la familia de experimentos 1.

En esta familia de experimentos, sólo se realizó hasta el archivo de tamaño 64 MB, debido a que la máquina no soportó el procesamiento de archivos de mayor tamaño.

Dado que para cualquier instancia de IDA bajo prueba tiene asociado un consumo de memoria que depende, entre otras cosas, de la combinación de parámetros (N, M) y el sistema operativo utiliza una parte de la memoria RAM para su operación y a esto se agrega el hecho de que la memoria RAM es limitada, en definitiva podemos decir que la RAM no es suficiente para ejecutar los archivos mayores a 64MB para esta familia de experimentos.

Sin embargo, aunque es lógico pensar que IDA no obtendrá el mismo tiempo de procesamiento con un hardware de características menores, ni con archivos de gran tamaño, se demostró que es capaz de ejecutarse para cualquier hardware, aunque no para todos los tamaños de archivo.

Esto se puede resolver implementando un IDA estático, es decir, se podría fijar la combinación que se adecue a las necesidades del usuario y del hardware y así evitar reservar

memoria en tiempo de ejecución, ya que esta no cambiaría a través del tiempo. Por ejemplo, la matriz de transformación no se calcularía en tiempo de ejecución, ya que como los parámetros N y M son fijos, la matriz no cambiaría. Así, las variables estáticas estarían contenidas en la zona de datos de la memoria y esta sección tiene reservada el espacio justo de memoria, ya que se conocen en tiempo de compilación. Es importante mencionar que, aun en este escenario, usar IDA16 ofrece ventajas sobre IDA8, lo cual es más notable en el caso de la dispersión.

4.4. Familia de experimentos 4. Archivo en memoria sobre IDA8 e IDA16

Sistema Operativo	Linux/ Ubuntu 16.04 LTS
Procesador	Intel Core i7
Memoria RAM	8 GB
Lectura/Escritura	En Memoria

Tabla 4.6: Condiciones complementarias en Linux con procesamiento en memoria

En esta sección, se presentan los resultados obtenidos de IDA al realizar el proceso de dispersión y recuperación en memoria, es decir, el archivo a manipular y los respectivos dispersos no se mantienen abiertos durante todo el proceso, sino que son montados en la memoria RAM para procesar operaciones entre matrices y vectores, evitando el tiempo de lectura y escritura en el disco.

Cada una de las gráficas de esta sección consta del proceso de dispersión y recuperación con los archivos en disco (Dis D y Rec D) así como el proceso de dispersión y recuperación en memoria (Dis M y Rec M). En la figura 4.8 mostramos los resultados obtenidos con un archivo de 1MB sobre IDA8 y en la figura 4.9 se muestran los resultados con IDA16 con el mismo tamaño de archivo.

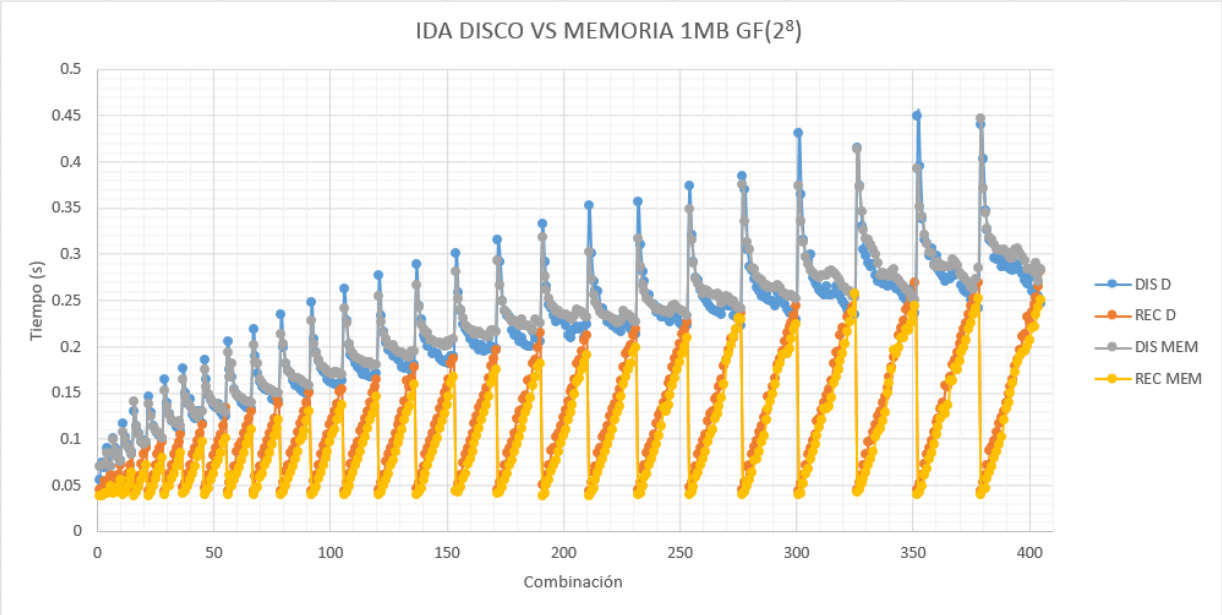

Figura 4.8: IDA8 con archivo de 1MB en disco vs memoria

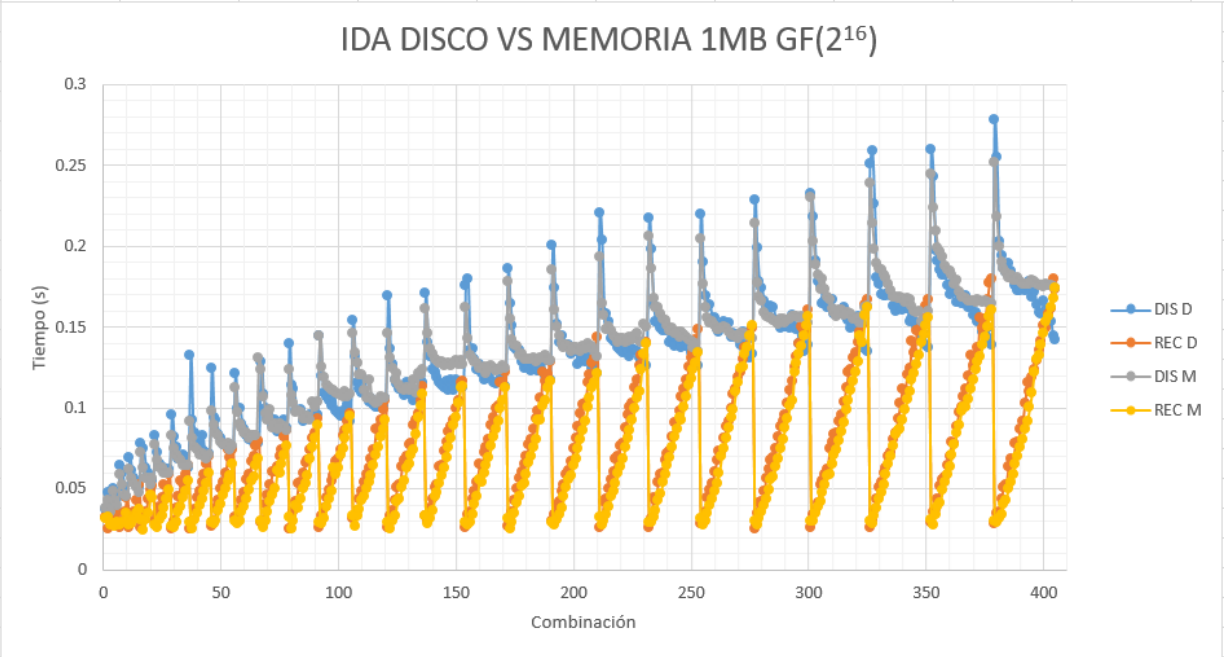

Figura 4.9: IDA16 con archivo de 1MB en disco vs memoria

A continuación se presentan los resultados que se obtuvieron con un archivo de 1024 MB montado a la memoria RAM, tanto para IDA8 (ver figura 4.10) como para IDA16 (ver figura 4.11).

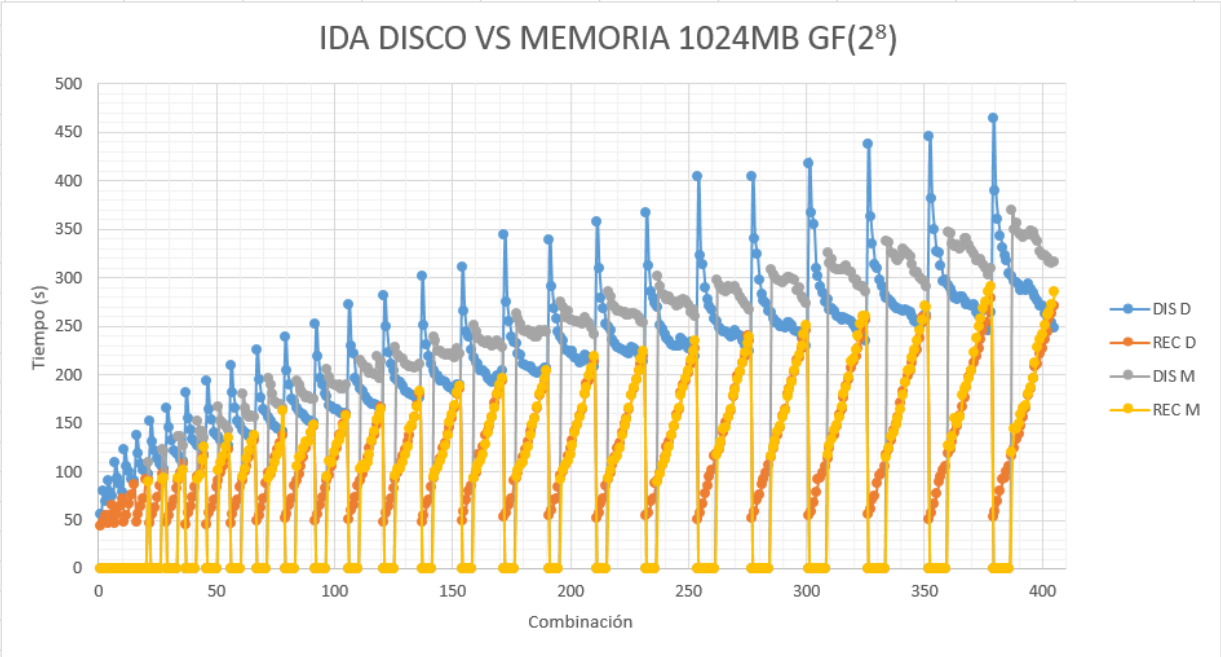

Figura 4.10: IDA8 con archivo de 1024MB en disco vs memoria

Se observa que en los resultados mostrados en la figura 4.8, el proceso de dispersión y recuperación para el archivo en memoria y para el archivo en disco es similar, dicho de otra manera, la diferencia en el tiempo para ambos casos es pequeña, e incluso llegan a estar sobrepuestos.

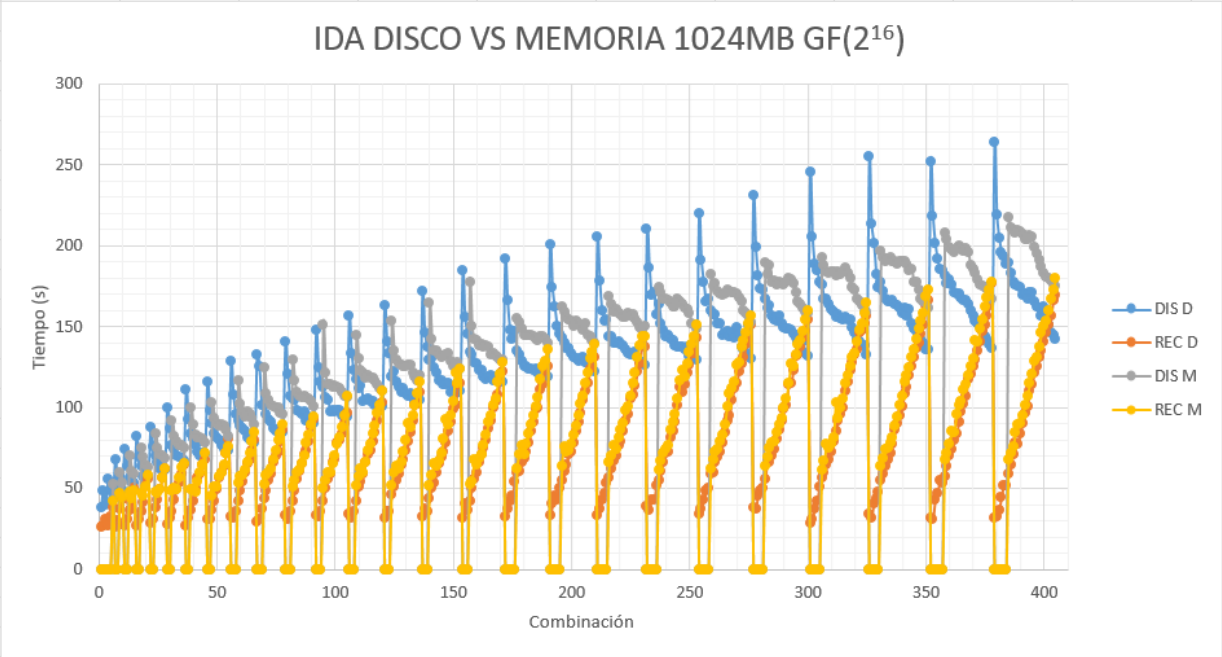

Figura 4.11: IDA16 con archivo de 1024MB en disco vs memoria

Por un lado, si examinamos IDA16 con el mismo tamaño de archivo (ver figura 4.9), observamos que el tiempo de dispersión en disco es mayor que el tiempo de dispersión en memoria para las combinaciones donde se genera una gran cantidad de información redundante. Es decir, en combinaciones donde se obtiene abundante redundancia, por ejemplo, las combinaciones 254, 301, 352 que corresponden con los parámetros $N = 25$ $M = 2$, $N = 27$ $M = 2$, $N = 29$ $M = 2$. Para el proceso de recuperación, observamos que el tiempo es muy similar en ambos casos.

Por otra parte, las gráficas de las figuras (ver figuras 4.10 y 4.11) observamos que no todas las combinaciones se pueden ejecutar en memoria. Percibimos que en las primeras combinaciones no se realiza el proceso de dispersión y como consecuencia no hay archivos con los cuales se pueda recuperar el archivo. Sin embargo, en IDA8 hay más combinaciones que no se pueden realizar, mientras que en IDA16 hay menos combinaciones que no se pueden realizar. Además, este resultado también se aprecia en las combinaciones con demasiada redundancia, es decir, cuando M es pequeña en todos los posibles valores de N.

También se observa que para el caso de IDA8, M está entre los valores 2 y 6 para las combinaciones que no se realiza el proceso, después para las últimas combinaciones, M toma los valores entre 2 y 9. Con esto observamos que entre más crezca el número

de dispersos o sea N, el valor de M aumenta en el sentido de que no se pueden realizar las combinaciones. Por ejemplo, en la tabla 4.7 se puede observar que al aumentar N, M aumenta 3 unidades en las combinaciones que no se ejecutan.

N	M	C	N	M	C
24	2	232	25	2	254
24	3	233	25	3	255
24	4	234	25	4	256
24	5	235	25	5	257
24	6	236	25	6	258
			25	7	259
			25	8	260
			25	9	261

Tabla 4.7: Combinaciones que no pueden ejecutarse

Para el caso de IDA16, sucede el mismo fenómeno pero en menor medida, esto es, el número de combinaciones que no se realizan es menor, es decir, M está entre los valores 2 y 3, luego para las últimas combinaciones que no se realiza el proceso, M está entre 2 y 5.

También, se muestra que el tiempo de proceso de dispersión en memoria es mayor que el tiempo de dispersión en disco para archivos de 1024 MB tanto para IDA8 como para IDA16. Además, el tiempo de recuperación en IDA8 es similar para ambos casos, mientras que en IDA16 el tiempo de recuperación en memoria es mayor que en disco.

Si analizamos la figura 4.8 y la figura 4.9 observamos que el tiempo de procesamiento en disco y en memoria es similar, por lo tanto, podemos decir que no hay una ganancia significativa en tiempo para este tamaño de archivos con IDA8.

Es necesario tener presente que el acceso al disco es más lento en comparación con el acceso a la memoria principal, por ello, existe un mecanismo para optimizar el rendimiento de la lectura y escritura de los datos desde el disco, este mecanismo es llamado cache de disco (Disk cache o Buffer Cache), el cual es una porción de RAM que se utiliza para acelerar las lecturas y escrituras del disco.

Así, cuando accedemos a los datos de forma reciente o recurrente, el buffer cache realiza una copia de los datos leídos del disco y se mantiene en la porción de RAM hasta que los datos no se necesiten por un periodo de tiempo. De igual manera para el proceso de escritura, los cambios se realizan en memoria principal antes de escribirse en el disco. Esto trae como consecuencia que mejore el rendimiento en el tiempo de acceso al disco [36].

Dado que Linux utiliza la memoria RAM que nos sobra para el Buffer Cache, implícitamente hace uso de la memoria RAM sin indicar que el archivo se sube a memoria para el caso de la implementación en disco, por tanto, la diferencia entre montar el archivo a memoria y no hacerlo, no es significativa.

Ahora, analizaremos las figuras 4.10 y 4.11. Como mencionamos anteriormente, las primeras combinaciones para la implementación en memoria no se pueden ejecutar en archivos de 1024 GB. Esto se debe a que éstas combinaciones generan un exceso de información redundante que compite por la memoria con otros requerimientos como el almacenamiento del archivo, la matriz de transformación, las tablas de logaritmo y antilogaritmo discreto, asi como los dispersos que se generan.

Aun cuando el sistema operativo permite que el proceso reserve más memoria del total de la que dispone la máquina entre la memoria RAM y memoria de intercambio (swap), en algún momento acaban realmente requiriendo más de la que está disponible. Por otro lado, el proceso de recuperación no se puede ejecutar, ya que no se generan los dispersos suficientes para recuperar el archivo. Esto sucede para las primeras combinaciones de cada parámetro N.

Por ejemplo, en la figura 4.12 tenemos información del estado de la memoria RAM cuando IDA no está en ejecución. En ésta se muestra en la primera columna el tamaño total de la memoria del sistema, en la segunda columna tenemos la memoria utilizada en ese momento, la tercera columna indica el tamaño de memoria libre o "desperdiciada", la cuarta columna indica la memoria compartida para la comunicación entre procesos, quinta columna indica el buffer cache, y la última columna es la memoria disponible para nuevos procesos.

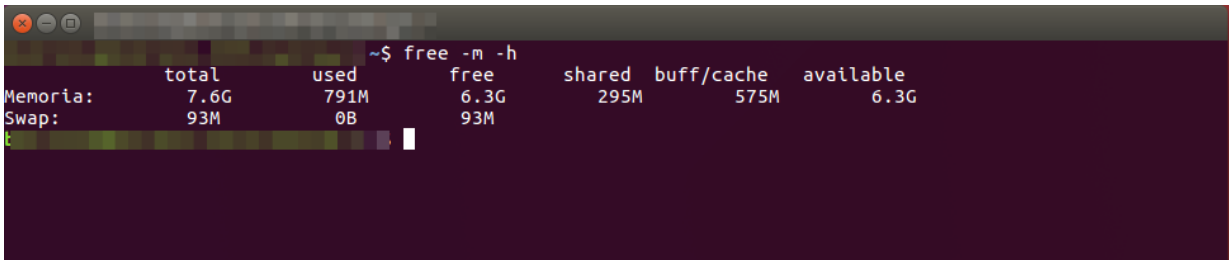

Figura 4.12: Ejemplo del consumo de memoria sin IDA

Se observa en la figura 4.13 que la memoria usada al ejecutar IDA pasa de 791M a 6.8G de un total de 7.6 GB, esto es, ocupa casi toda la memoria RAM. Además, la memoria

swap usa 93M de un total 93M, esto quiere decir que IDA necesita más memoria de la que el sistema operativo le puede proporcionar.

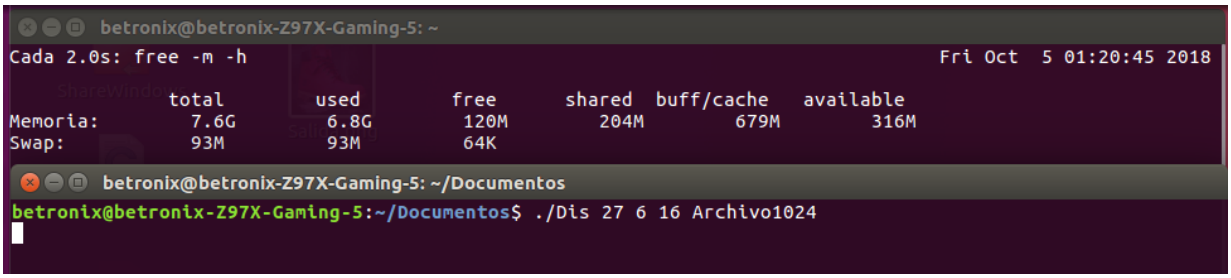

Figura 4.13: Ejemplo del consumo de memoria con IDA

4.5. Familia de experimentos 5. Impacto de las operaciones en IDA

En esta sección, se presentan los resultados de una serie de experimentos en los que estudiamos el impacto de las operaciones de acceso sobre los archivos en IDA al realizar el proceso de dispersión y recuperación para IDA8 e IDA16, es decir, se simuló IDA sin el procesamiento de las operaciones aritméticas. Esto con la finalidad de indagar en el costo de procesamiento de las operaciones de entrada y salida, o dicho de otra manera, que impacto tiene el costo de procesamiento de las operaciones aritméticas en IDA.

Cabe mencionar que "simulamos" la dispersión aun cuando los archivos de salida no funcionan como dispersos, pero contienen la misma cantidad de información que cualquiera de estos y tomaron la misma cantidad de operaciones de entrada y salida. De igual forma, "simulamos" la recuperación, pero eliminando los costos de las operaciones aritméticas.

En cada apartado, se muestra una tabla que indica las características de cada experimento de esta familia, además, cada una de las gráficas de esta sección consta del proceso de dispersión y recuperación haciendo las lecturas y escrituras desde el disco.

4.5.1. Impacto de las operaciones con Linux

Sistema Operativo	Linux/ Ubuntu 16.04 LTS
Procesador	Intel Core i7
Memoria RAM	8 GB
Lectura/Escritura	Disco
Sistema de Archivos	Ext4

Tabla 4.8: Condiciones complementarias en Linux

Cabe mencionar que en esta sección se muestran los resultados del archivo de 1024MB. Las demás gráficas se muestran en el Apéndice E.1. En la figura 4.14 mostramos los resultados obtenidos con un archivo de 1024MB sobre IDA8 y en la figura 4.15 se muestran los resultados con IDA16 con el mismo tamaño de archivo.

Además, en todas las gráficas que se muestran, la línea azul representa el costo de leer un archivo de $|F|$ bytes y de escribir $n \cdot \frac{|F|}{m}$ bytes de salida, como se hace en la dispersión. En tanto, la línea naranja representa el costo de leer un archivo de $|F|$ bytes y de escribir

$|F|$ bytes, como aplica a la recuperación. Por otra parte, la línea gris, representa el costo total de la dispersión, mientras que la amarilla representa el costo total de la recuperación. Ambas, están tomadas de los resultados de la sección 4.1.

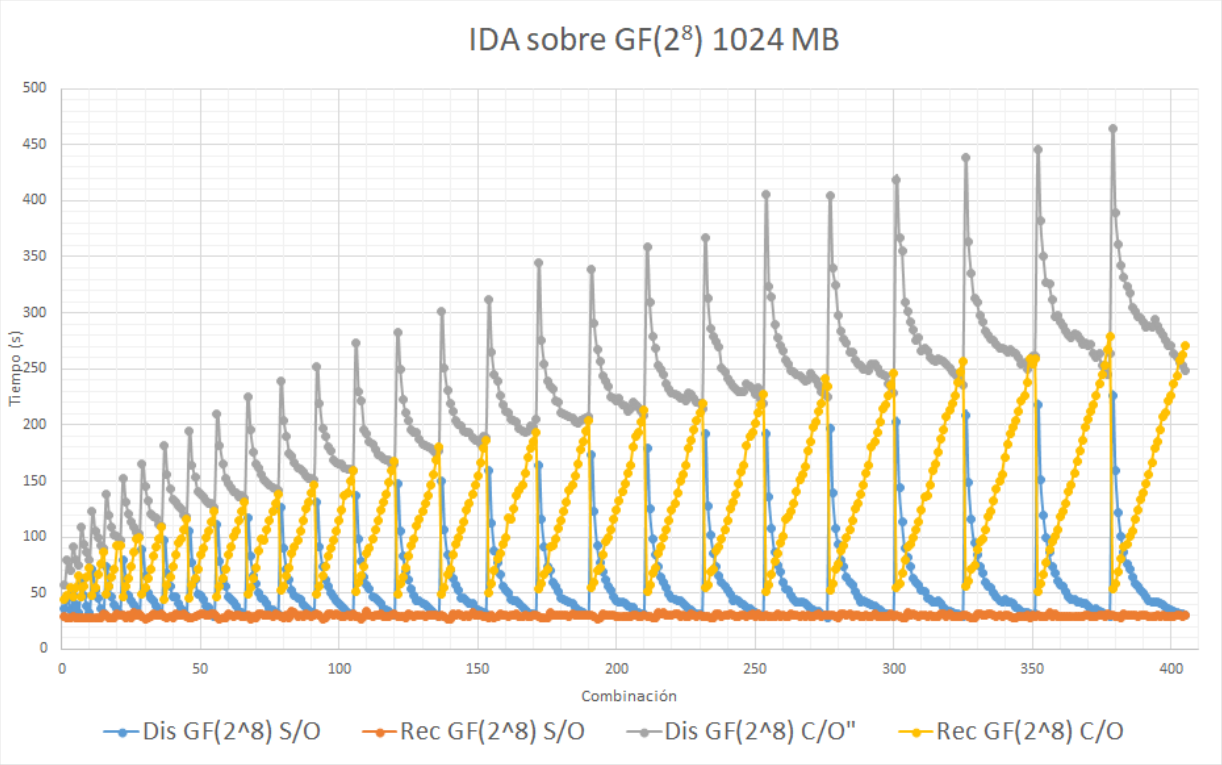

Figura 4.14: IDA8 con archivo de 1024MB sobre Linux sin operaciones aritméticas

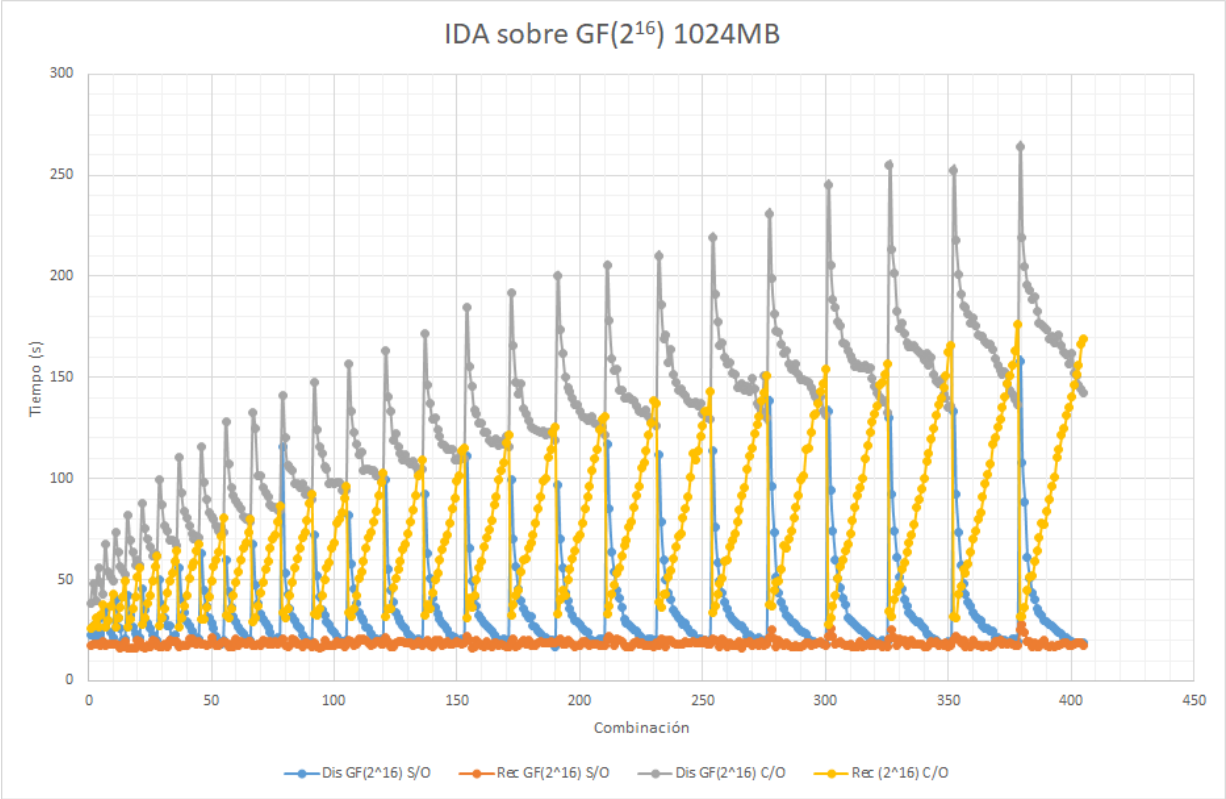

Figura 4.15: IDA16 con archivo de 1024MB sobre linux sin operaciones aritméticas

Se observa que el costo de las operaciones aritméticas en la dispersión puede obtenerse por la diferencia entre las alturas de la línea azul y gris, se puede deducir que esta diferencia es una función que, para n fija, no depende de m o es constante. Mientras que, para la recuperación, se consigue por la diferencia entre la línea amarilla y la naranja y es evidente que, para n fija, es una función lineal de m.

4.5.2. Impacto de las operaciones con Windows

Para esta sección se realizaron las pruebas en un sistema de archivos diferente y que en la tabla E.2 se muestran las características de este experimento. Al igual que el experimento anterior, se muestra el resultado con un archivo de tamaño 1024MB; los demás resultados se encuentran en el Apéndice E.2.

Sistema Operativo	Windows 10
Procesador	Intel Core i7
Memoria RAM	8 GB
Lectura/Escritura	Disco
Sistema de Archivos	NTFS

Tabla 4.9: Condiciones complementarias en Windows

En la figura 4.16 mostramos los resultados obtenidos con un archivo de 1024MB sobre IDA8 y en la figura 4.17 se muestran los resultados con IDA16 con el mismo tamaño de archivo.

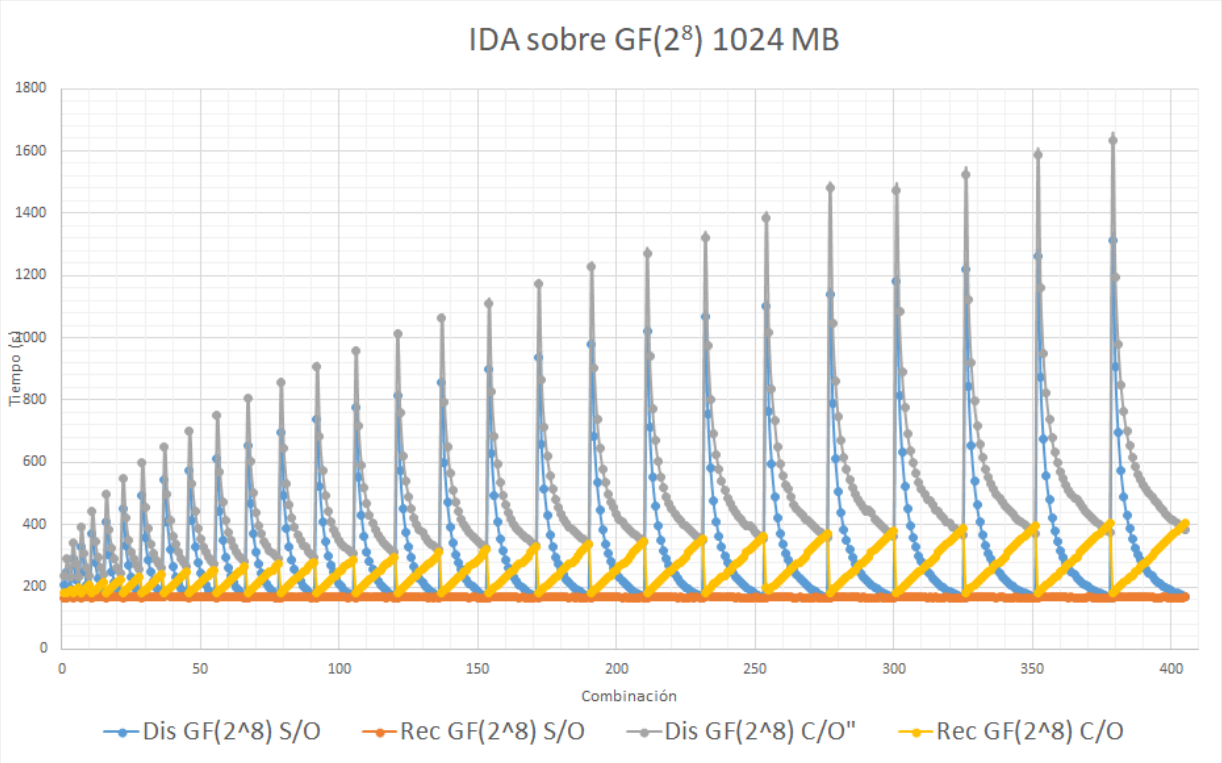

Figura 4.16: IDA8 con archivo de 1024MB sobre Windows sin operaciones aritméticas

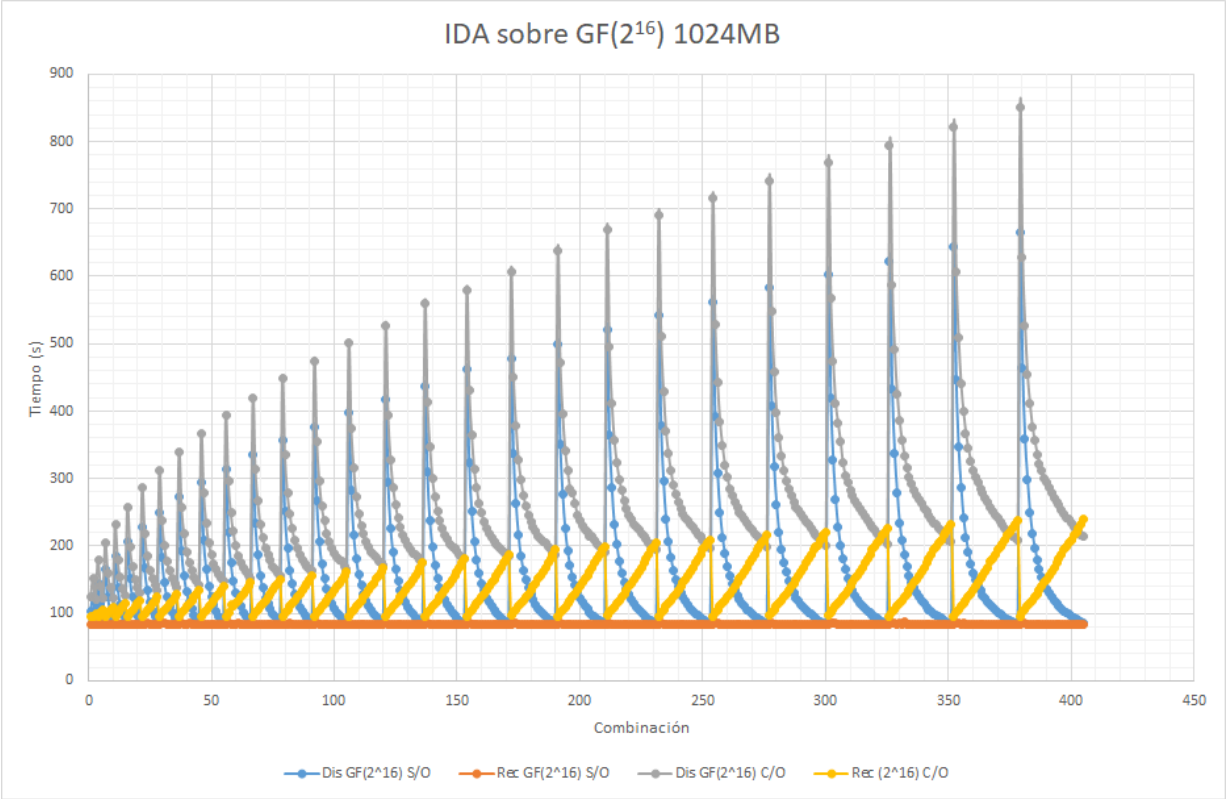

Figura 4.17: IDA16 con archivo de 1024MB sobre Windows sin operaciones aritméticas

Al igual que el experimento anterior, se observa que para el proceso de dispersión y una n fija, no depende de m, mientras que para el proceso de recuperación, es una función lineal de m para una n fija o particular.

Por lo que se refiere al costo de las operaciones aritméticas, en todos los casos son más rápidas sobre Linux que sobre Windows, pero la diferencia en tiempos puede ser relativamente pequeña o incluso marginal. En tanto, sobre un mismo Sistema Operativo, el costo de las operaciones aritméticas se reduce, cuando mucho a la mitad, cuando pasamos de IDA8 a IDA16.

Luego, para una n fija, el costo de la dispersión es superior al de la recuperación para valores pequeños de m y sólo cuando $m \to n$ se puede invertir la relación, pero en un valor marginal. Sin embargo, esta relación de costos es más pronunciada sobre Windows, comparada con Linux. Esta diferencia entre los costos de dispersión y recuperación se reduce al pasar de IDA8 a IDA16 aproximadamente a la mitad.

Capítulo 5

Conclusiones

Este trabajo ha presentado un estudio sobre el Algoritmo de Dispersión de Información, en el que se han investigado las diferentes condiciones que inciden en sus costos de operación, como son: el orden del campo finito sobre el que se opera, la cantidad de información redundante que se genera, el sistema operativo con el que se trabaja, el hardware con el que se desempeña, la gestión de las operaciones de lectura y escritura y la influencia de los tiempos de las operaciones aritméticas.

5.1. Conclusiones generales

Los resultados presentados en los capítulos anteriores pueden contrastarse con el análisis recién expuesto, para concluir que la complejidad del algoritmo directo e inverso dependen de condiciones que se hacen evidentes solamente bajo un ambiente de experimentación como el que construimos.

En este contexto pudimos reconocer particularidades tales como:

La implementación del algoritmo de dispersión de información (IDA) sobre los campos finitos de orden $GF(2^8)$ y $GF(2^{16})$ con lo cual se comprobó que utilizar un campo de mayor orden reduce el tiempo de procesamiento.

Es preferible utilizar un campo $GF(2^{16})$ con archivos de mayor tamaño, de lo contrario, no tendría una ganancia significativa de tiempo.

El tiempo del proceso de dispersión será mayor al tiempo de recuperación de un archivo

Nuestro estudio ofrece al diseñador, información suficiente para elegir la tecnología de almacenamiento y el campo finito con los cuales debe trabajar, así como los criterios para elegir una combinación entre n y m, que se acomode mejor a las necesidades de una aplicación en particular como diferentes puntos de equilibrio entre la redundancia, la tolerancia a fallas, tiempo de procesamiento, número de dispersos, entre otros.

El tamaño del archivo afecta directamente al tiempo de dispersión y recuperación.

El hardware tiene un impacto considerable en el rendimiento de IDA, aunque es posible ejecutar IDA en cualquier dispositivo, aún si este es de menor rendimiento.

Es conveniente ejecutar IDA en memoria con archivos de menor tamaño, por otro lado, es adecuado procesar un archivo de mayor tamaño sobre disco.

El costo de invertir la matriz de recuperación tiene un impacto despreciable en el desempeño del algoritmo inverso. Ello se explica porque, aun cuando se tiene una complejidad cúbica sobre el parámetro m, no puede compararse con el volumen de las demás operaciones aritméticas que, como hemos visto, dependen del tamaño del archivo original.

También es importante considerar, que un valor grande de n implica un mayor tiempo de dispersión. Por lo anterior, no parece muy conveniente la alternativa $m \to n$, cuando n toma un valor muy grande, por ejemplo $(n > 10)$, por el costo de procesamiento pero, sobretodo, por el riesgo en que se pone al sistema de almacenamiento, especialmente cuando sus componentes envejecen y aumentan sus probabilidades de falla.

Es importante mencionar que todas las operaciones aritméticas efectuadas en nuestro trabajo están basadas en el método del logaritmo discreto. Con este enfoque se construye una tabla con las potencias del elemento generador del campo de trabajo. Esta tabla permite conseguir tiempos constantes en todas las operaciones, pero su limitación más seria está en el tamaño de dicha tabla. Para IDA8 la tabla tiene 2^8 entradas, y para IDA16 tiene 2^{16}. Aun cuando es posible leer y escribir secuencias con longitudes superiores a 16 bits, es una limitación seria pensar en tablas de 2^{24} entradas, por ejemplo.

Este trabajo inició conjeturando la importancia del orden del campo finito sobre el cual se trabaja, pero nuestros experimentos demostraron que el orden del campo es tan sólo una forma de controlar las operaciones de lectura y escritura de archivos, las cuales tienen el mayor impacto en el desempeño de IDA.

5.2. Trabajo futuro

Para la continuación de esta investigación se sugiere realizar una implementación sobre campos de mayor orden, ya que se ha comprobado que al aumentar el campo se reduce la velocidad de procesamiento tanto para la dispersión como para la recuperación. Vale la pena agregar que existen otras formas de acelerar el algoritmo, usando algún tipo de procesamiento paralelo, o utilizando un enfoque de transformada rápida, pero estos últimos sólo aplican sobre campos finitos muy particulares.

Se propone variar la memoria memoria virtual del sistema operativo para observar el comportamiento del proceso de IDA y en que afectaría, ya que se observó que IDA en Linux hace uso de ésta. Además de ejecutar IDA con más procesos para ver el consumo de recursos en el equipo de computo, ya que será utilizado en aplicaciones de alto desempeño.

Se sugiere realizar diferentes combinaciones de hardware como discos duros de estado sólido, aumento de la memoria RAM, entre otros, ya que los nuevos equipos de cómputo tienen nuevos componentes tecnológicos.

Apéndice A

Resultados de familia de experimentos 1. IDA8 vs IDA16

Sistema Operativo	Linux/ Ubuntu 16.04 LTS
Procesador	Intel Core i7
Memoria RAM	8 GB
Lectura/Escritura	En Disco

Tabla A.1: Condiciones complementarias para la familia de experimento 1

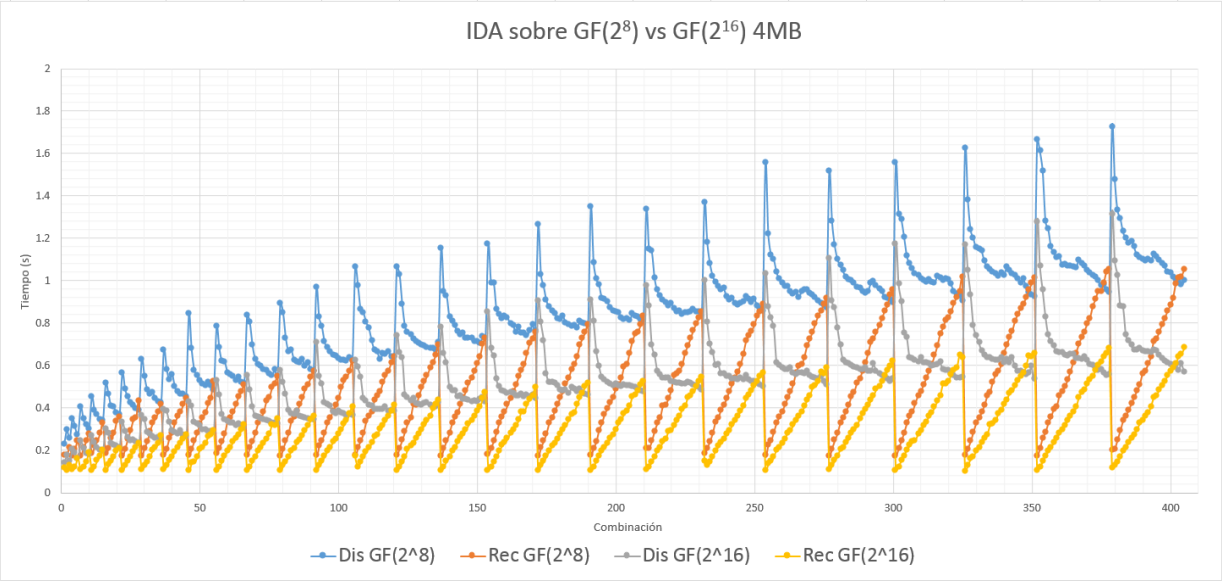

Figura A.1: IDA8 vs IDA16 con archivo de 4MB

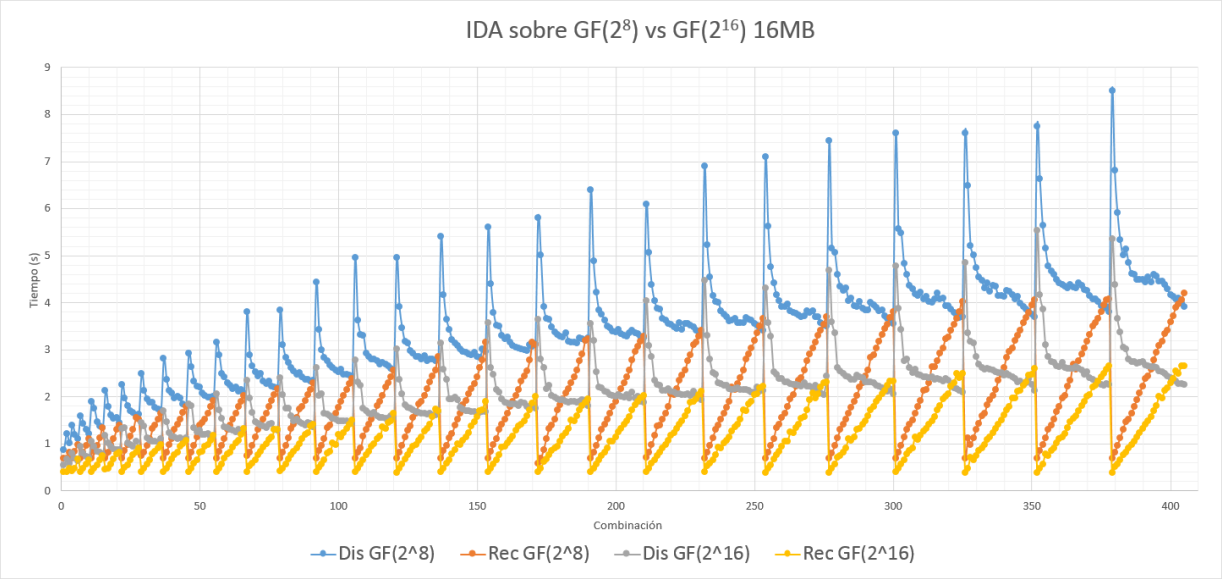

Figura A.2: IDA8 vs IDA16 con archivo de 16MB

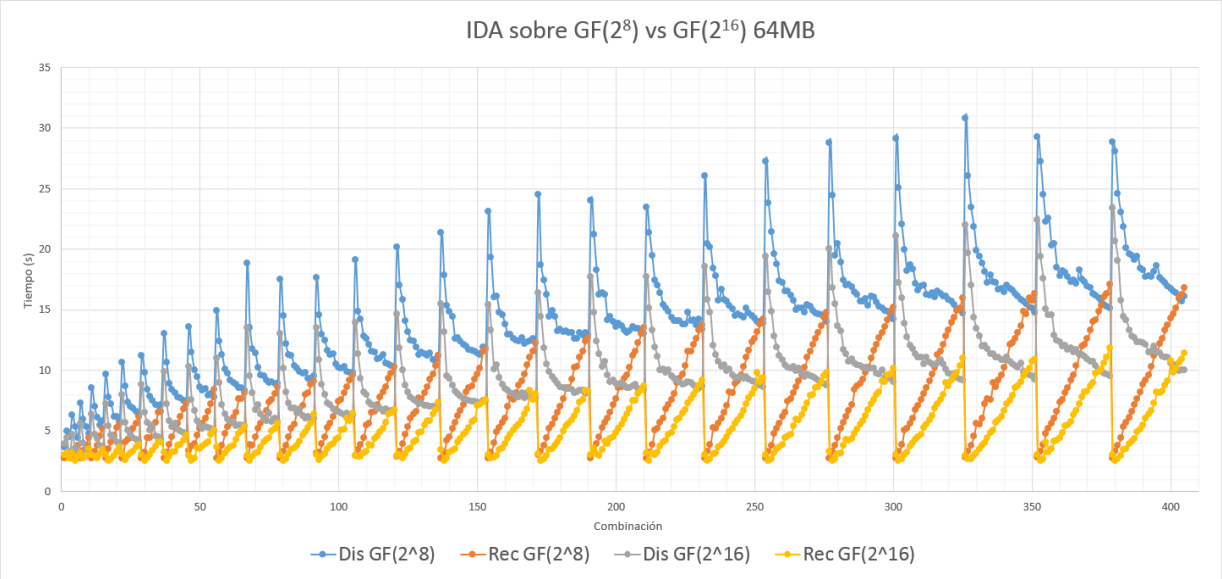

Figura A.3: IDA8 vs IDA16 con archivo de 64MB

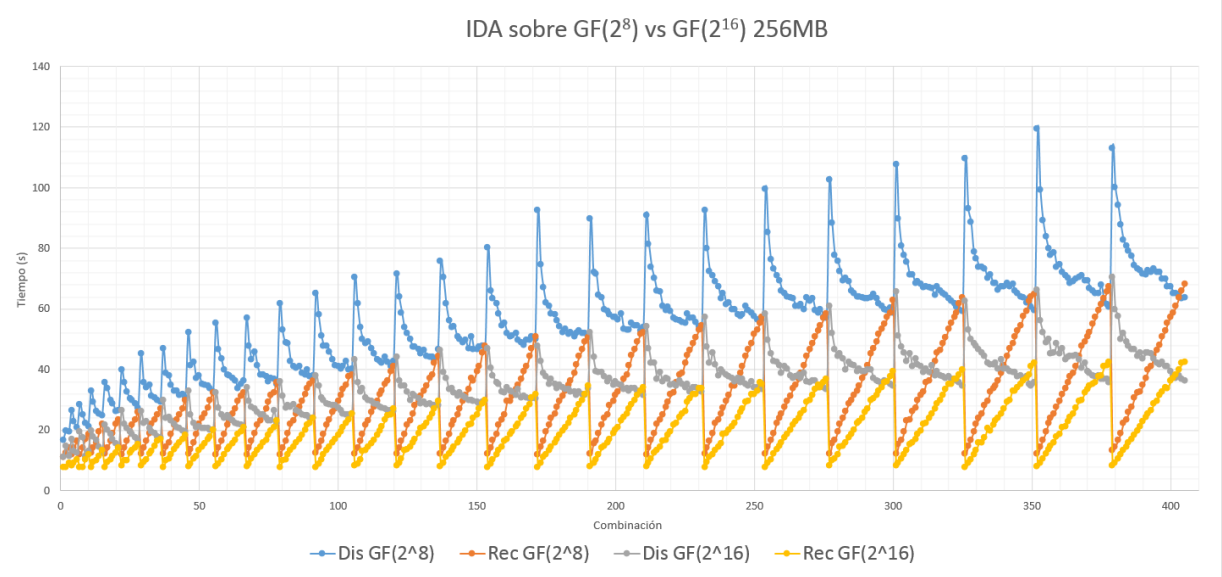

Figura A.4: IDA8 vs IDA16 con archivo de 256MB

Apéndice B

Resultados de familia de experimentos 2. Impacto del sistema operativo

Sistema Operativo	Windows
Procesador	Intel Core i7
Memoria RAM	8 GB
Lectura/Escritura	En Disco

Tabla B.1: Condiciones complementarias para la familia de experimentos 2

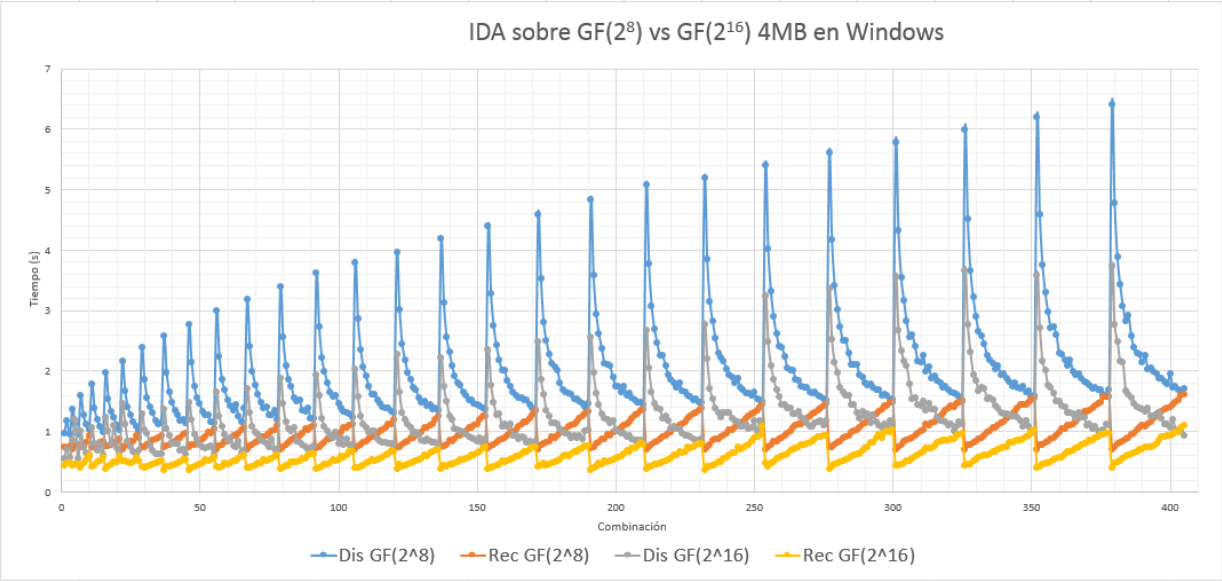

Figura B.1: IDA8 vs IDA16 con archivo de 4MB en Windows

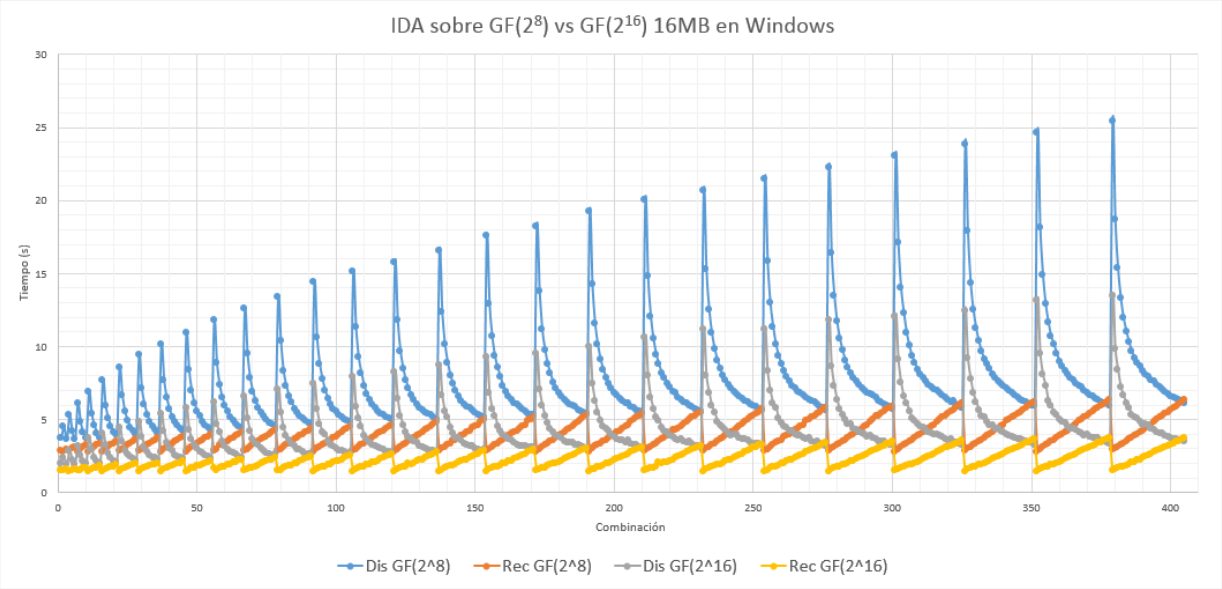

Figura B.2: IDA8 vs IDA16 con archivo de 16MB en Windows

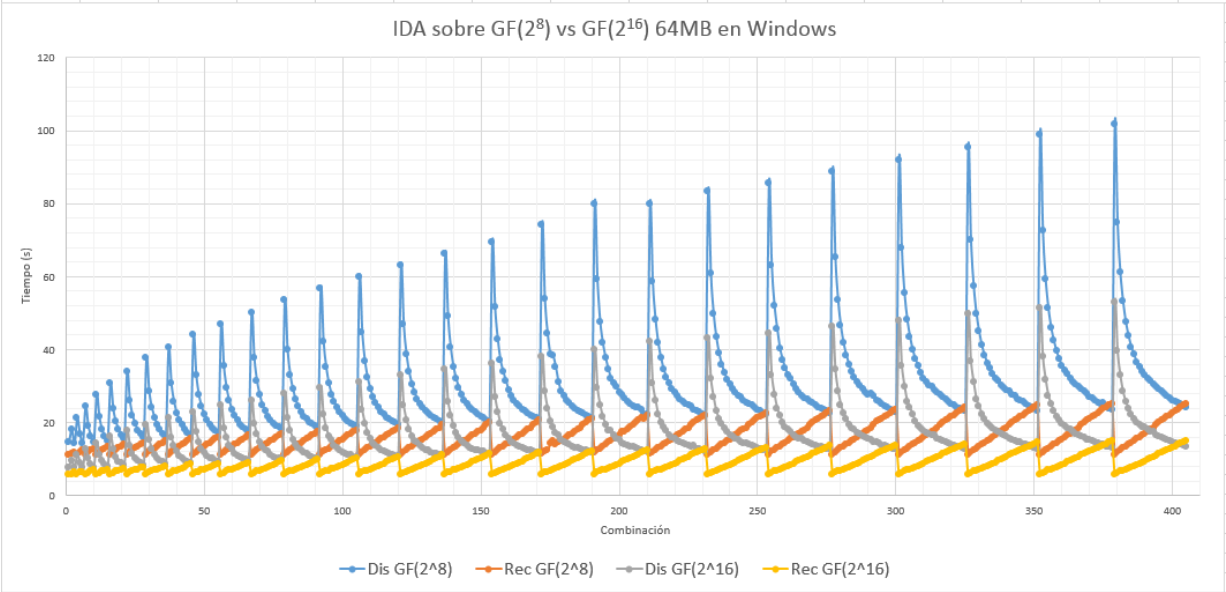

Figura B.3: IDA8 vs IDA16 con archivo de 64MB en Windows

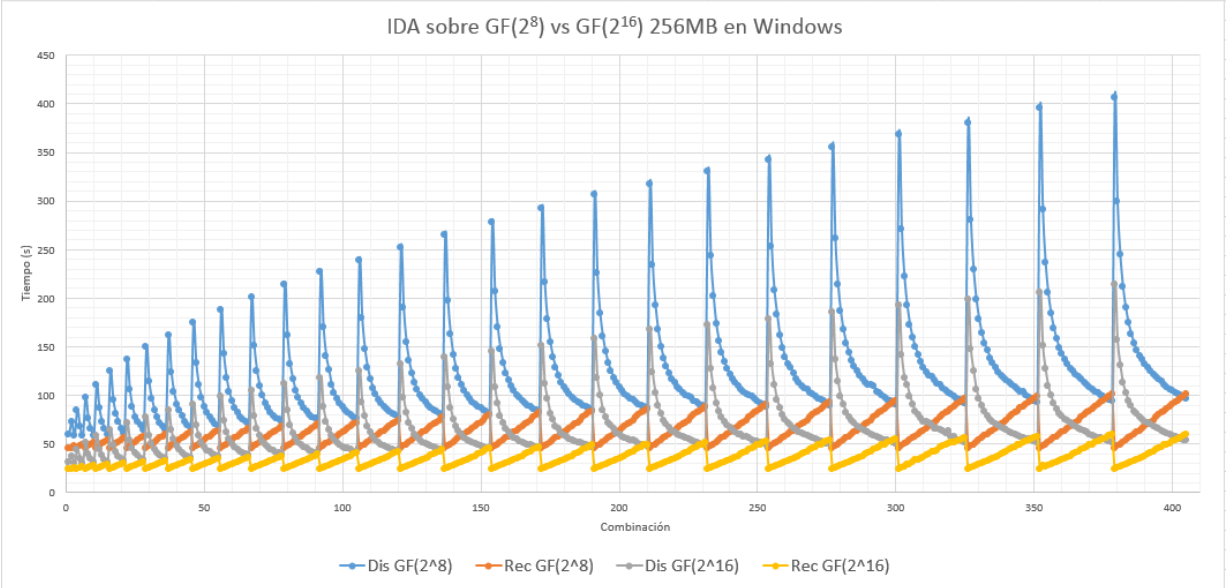

Figura B.4: IDA8 vs IDA16 con archivo de 256MB en Windows

Apéndice C

Resultados de familia de experimentos 3. Impacto del hardware de procesamiento

Sistema Operativo	Linux/ Ubuntu 16.04 LTS
Procesador	Intel Atom
Memoria RAM	2 GB

Tabla C.1: Condiciones complementarias en Linux con procesador Atom.

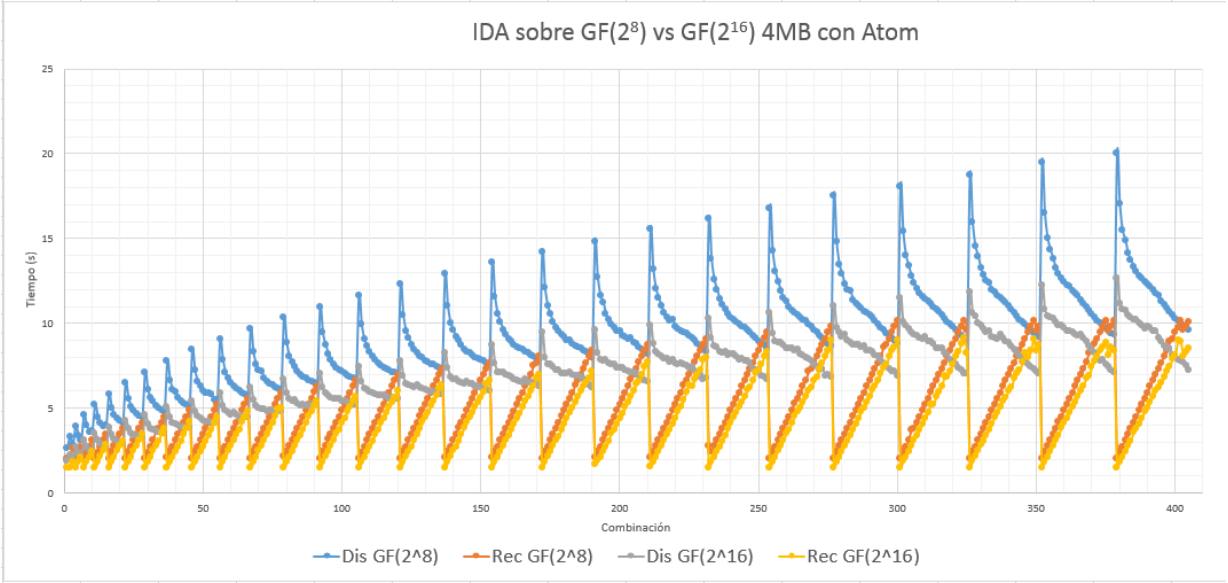

Figura C.1: IDA8 vs IDA16 con archivo de 4MB con procesador Atom

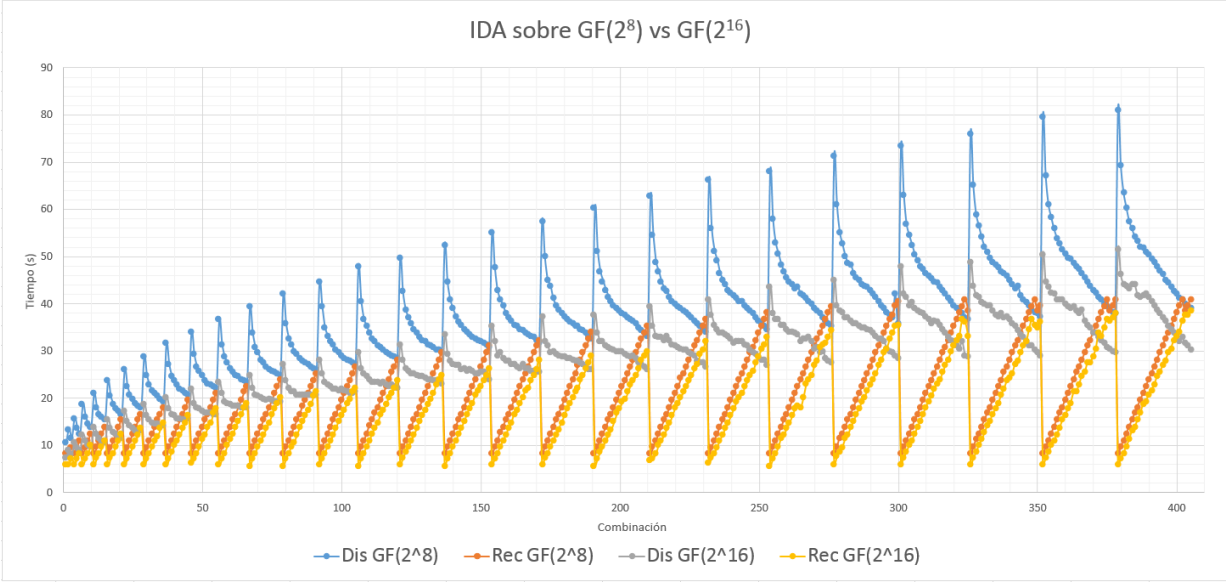

Figura C.2: IDA8 vs IDA16 con archivo de 16MB con procesador Atom

Apéndice D

Resultados de familia de experimentos 4. Archivo en memoria sobre IDA8 e IDA16

Sistema Operativo	Linux/ Ubuntu 16.04 LTS
Procesador	Intel Core i7
Memoria RAM	8 GB
Lectura/Escritura	En Memoria

Tabla D.1: Condiciones complementarias en Linux con procesamiento en memoria.

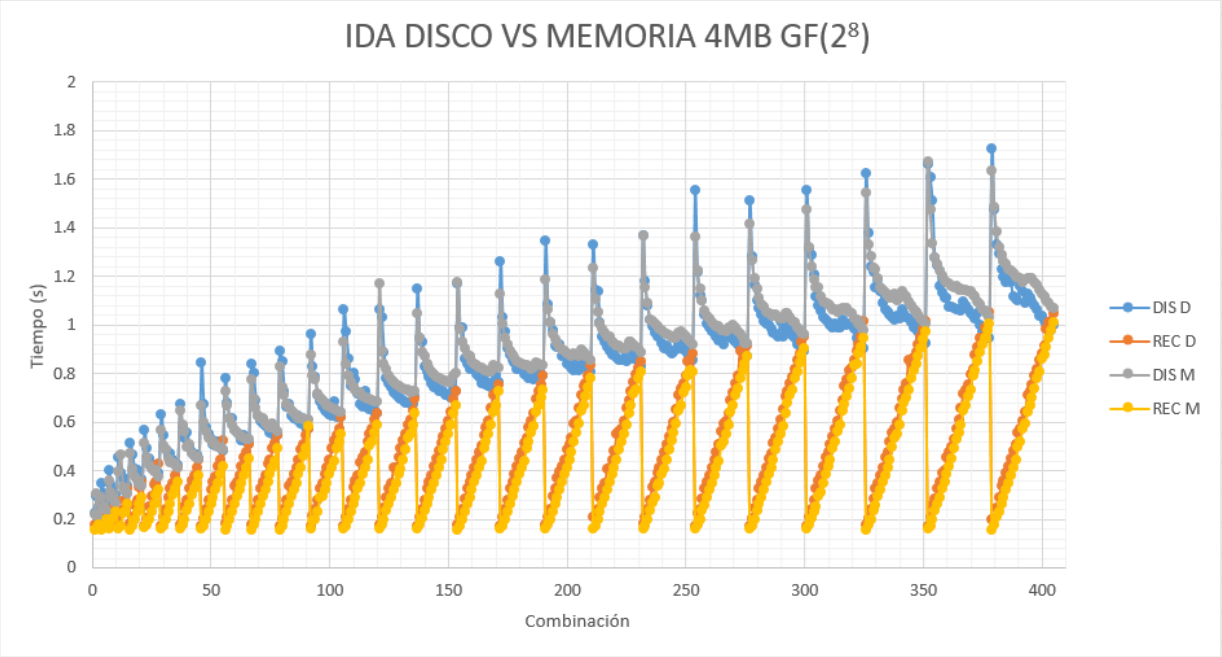

Figura D.1: IDA8 con archivo de 4MB en disco vs memoria

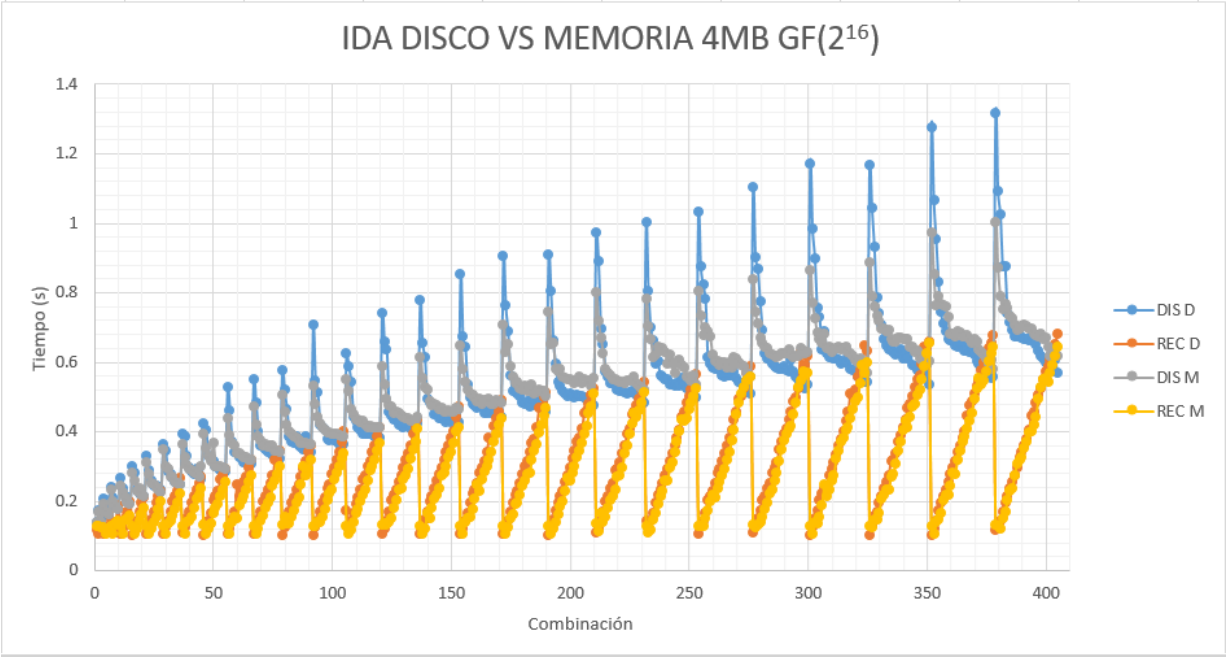

Figura D.2: IDA16 con archivo de 4MB en disco vs memoria

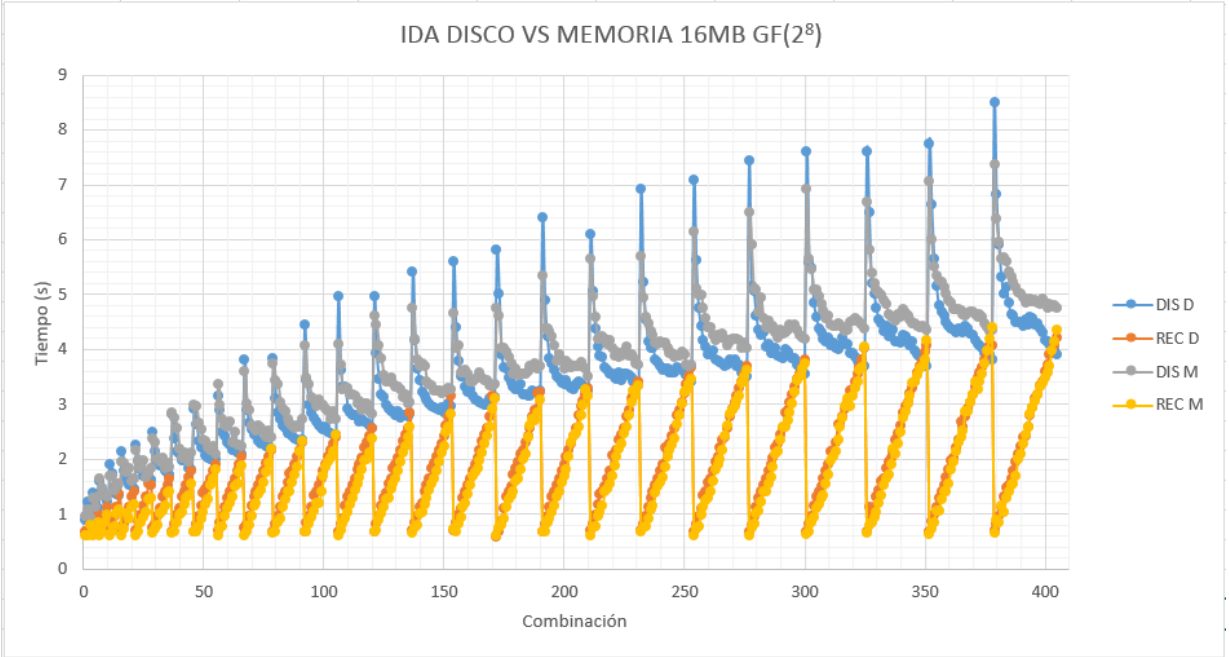

Figura D.3: IDA8 con archivo de 16MB en disco vs memoria

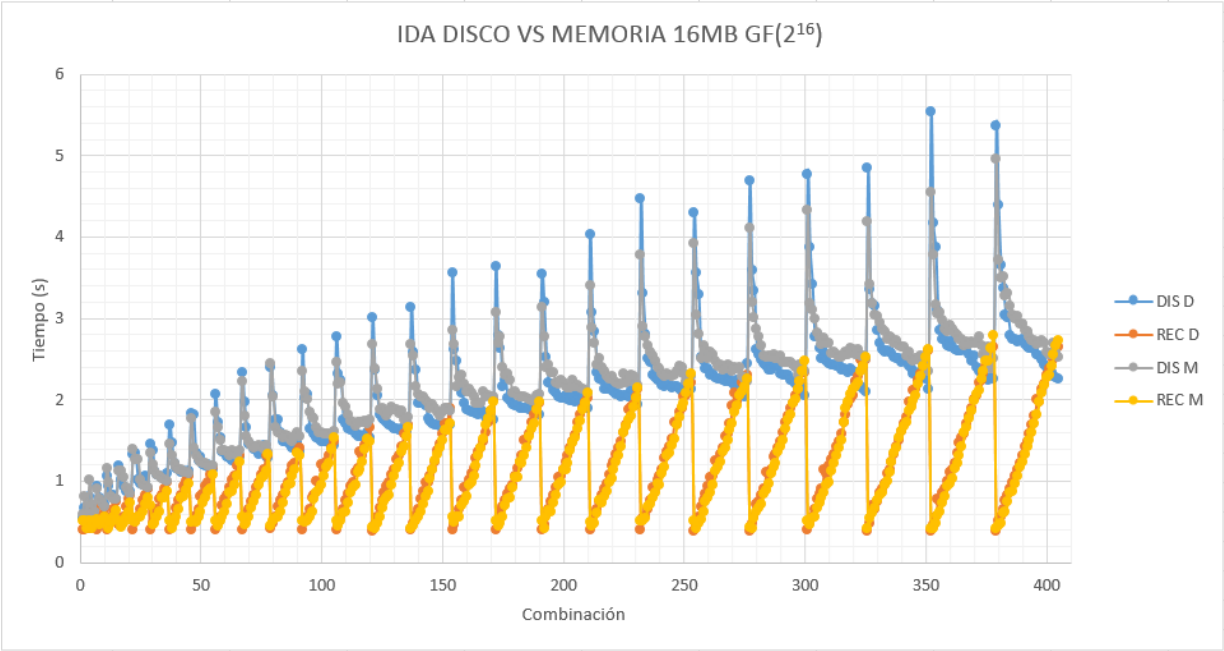

Figura D.4: IDA16 con archivo de 16MB en disco vs memoria

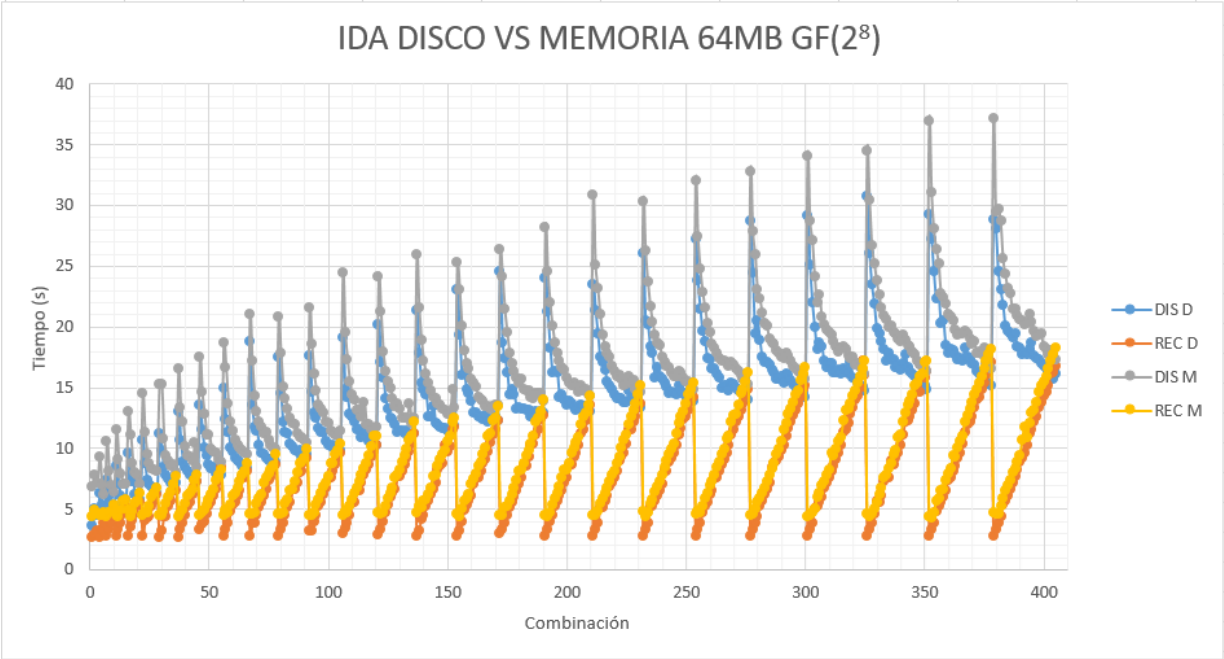

Figura D.5: IDA8 con archivo de 64MB en disco vs memoria

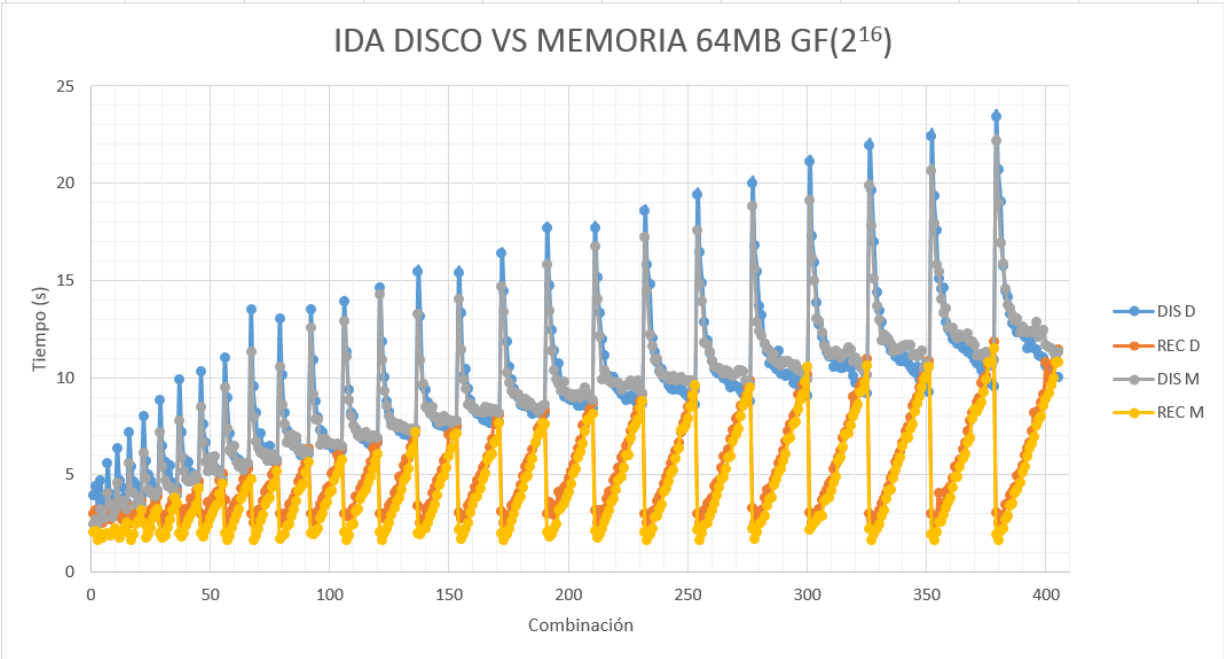

Figura D.6: IDA16 con archivo de 64MB en disco vs memoria

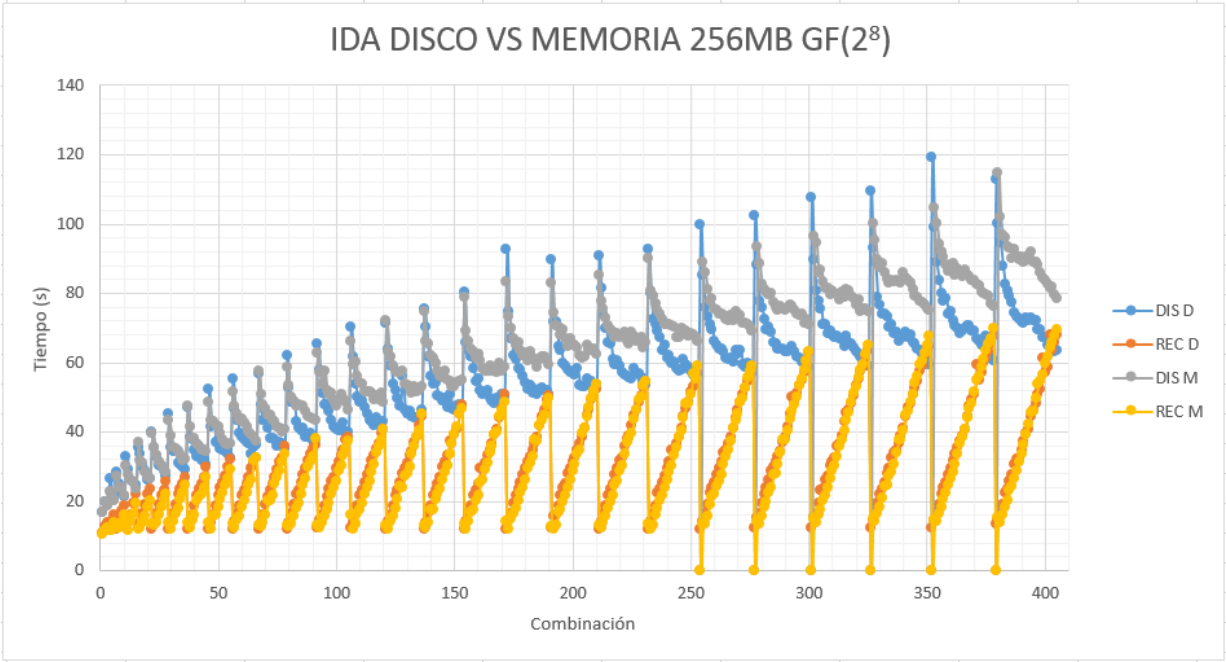

Figura D.7: IDA8 con archivo de 256MB en disco vs memoria

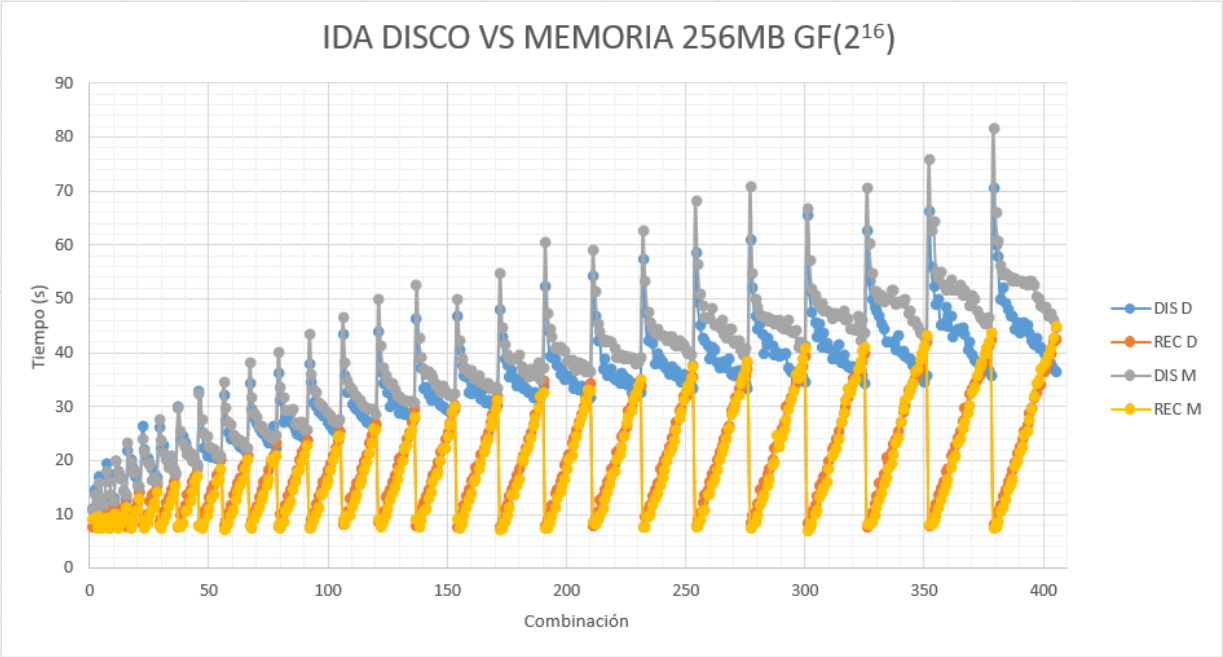

Figura D.8: IDA16 con archivo de 256MB en disco vs memoria

Apéndice E

Resultados de familia de experimentos 5. Impacto de las operaciones en IDA

E.1. Impacto de las operaciones con Linux

Sistema Operativo	Linux/ Ubuntu 16.04 LTS
Procesador	Intel Core i7
Memoria RAM	8 GB
Lectura/Escritura	Disco
Sistema de Archivos	Ext4

Tabla E.1: Condiciones complementarias en Linux

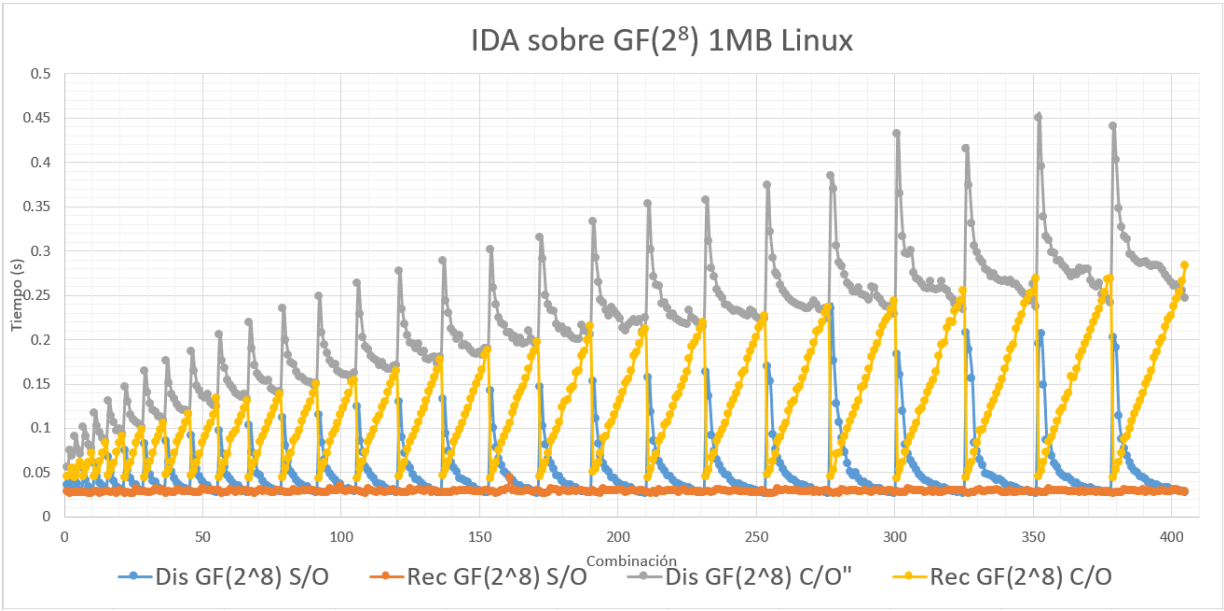

Figura E.1: IDA8 con archivo de 1MB sobre Linux sin operaciones aritméticas

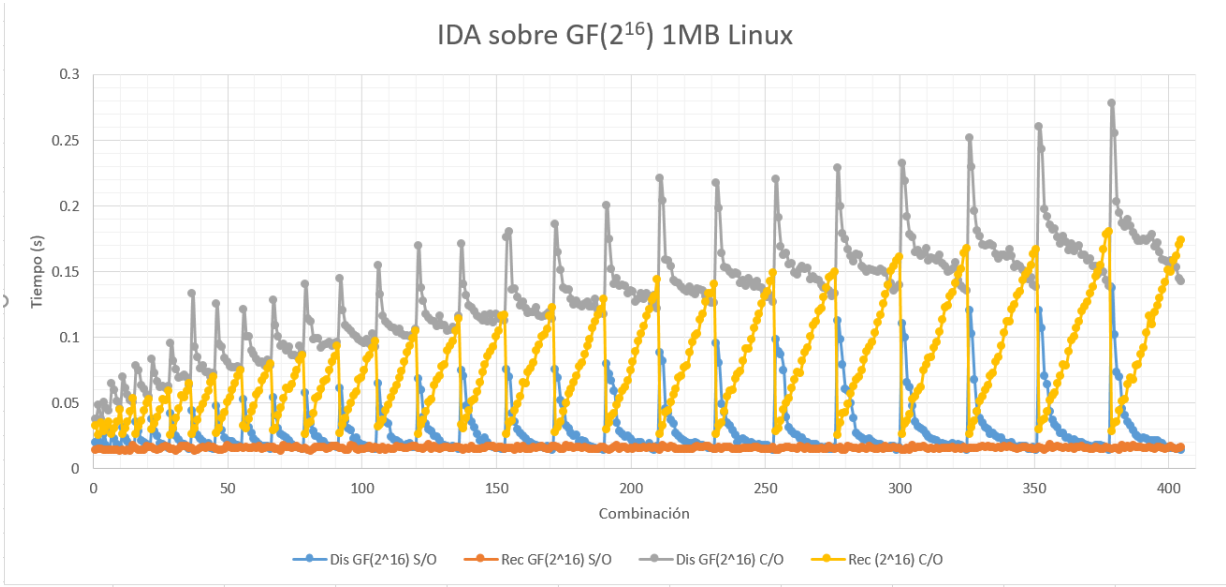

Figura E.2: IDA16 con archivo de 1MB sobre linux sin operaciones aritméticas

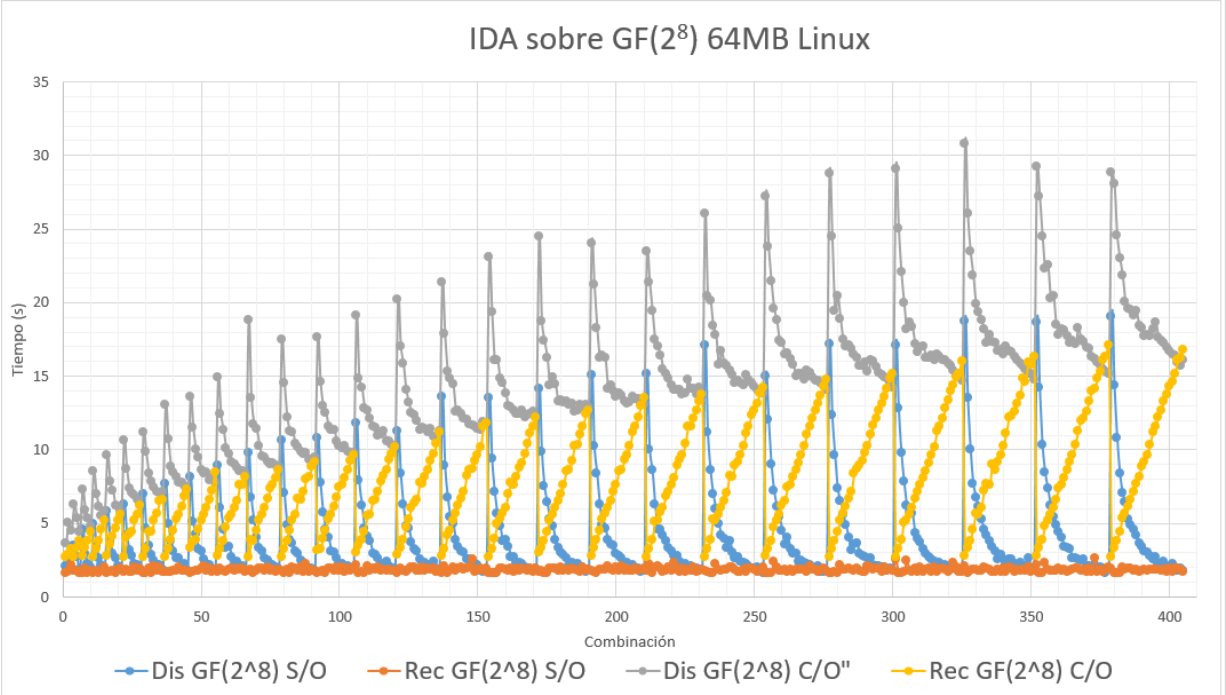

Figura E.3: IDA8 con archivo de 64MB sobre Linux sin operaciones aritméticas

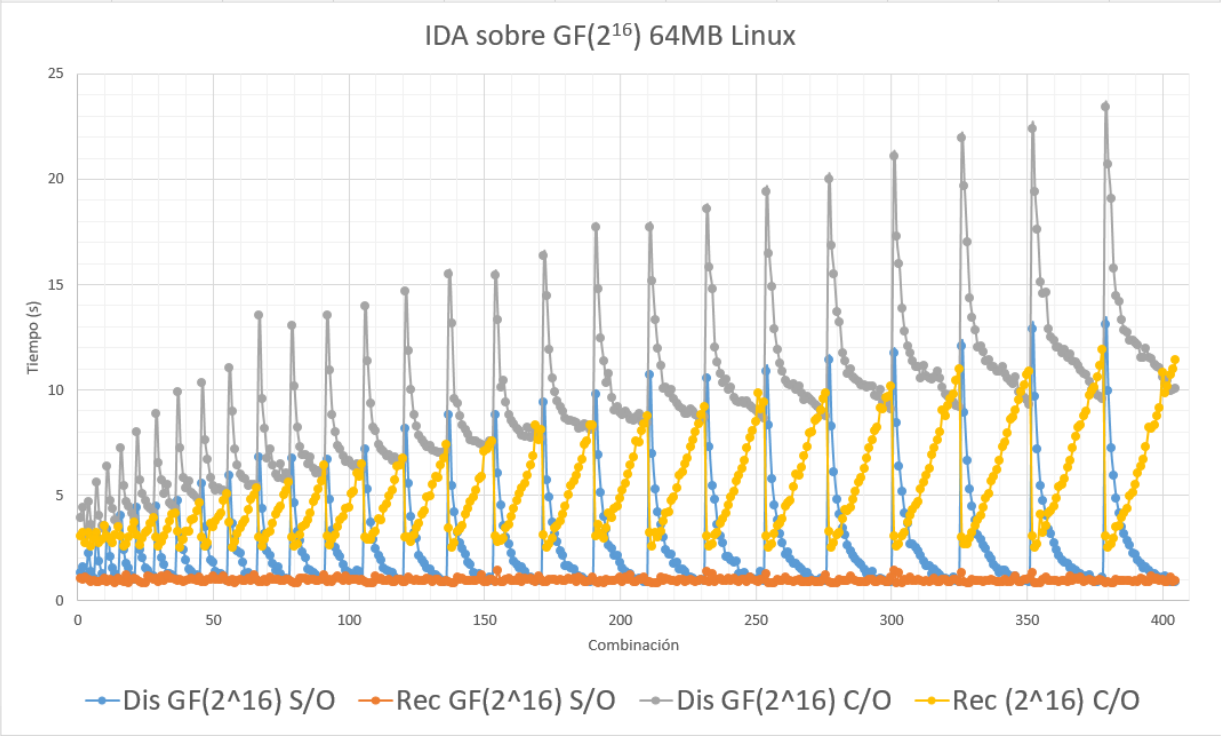

Figura E.4: IDA16 con archivo de 64MB sobre linux sin operaciones aritméticas

E.2. Impacto de las operaciones con Windows

Sistema Operativo	Windows 10
Procesador	Intel Core i7
Memoria RAM	8 GB
Lectura/Escritura	Disco
Sistema de Archivos	NTFS

Tabla E.2: Condiciones complementarias en Windows

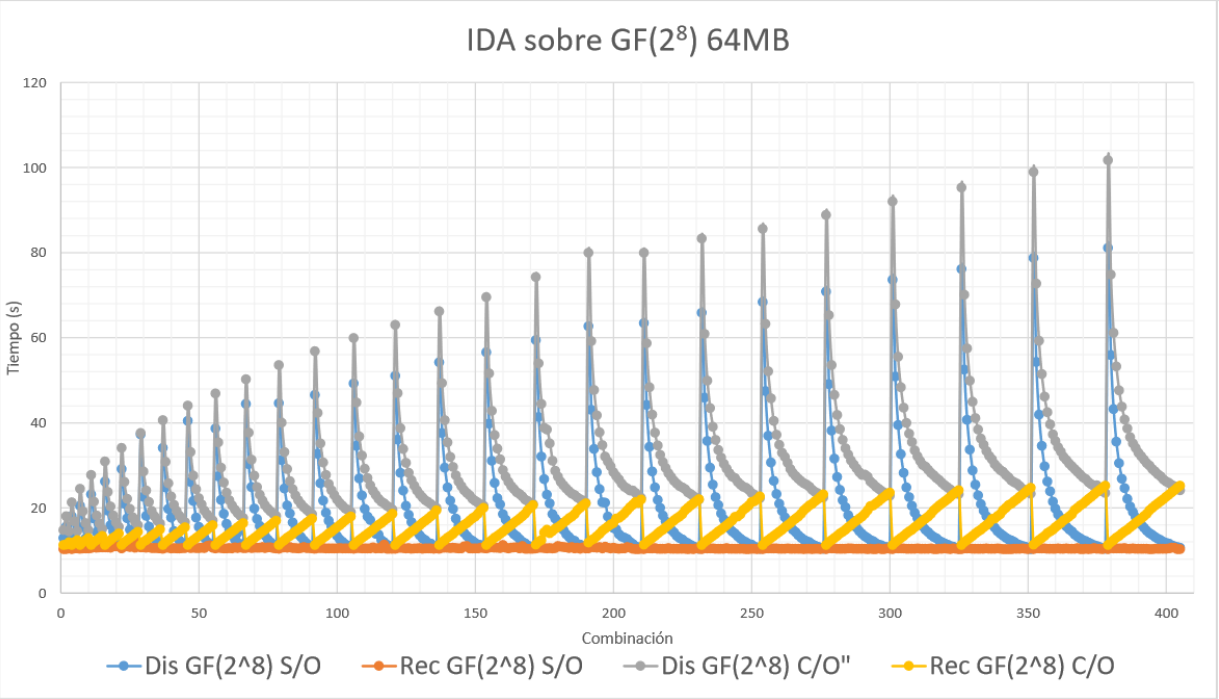

Figura E.5: IDA8 con archivo de 64MB sobre Windows sin operaciones aritméticas

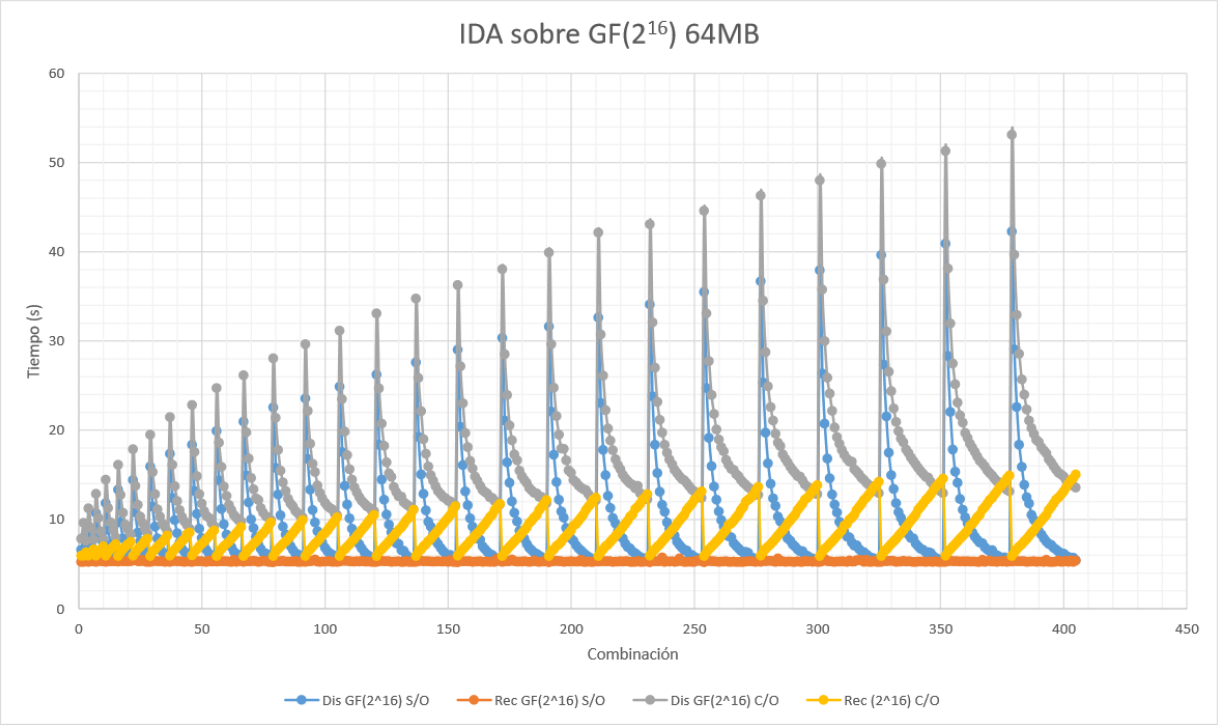

Figura E.6: IDA16 con archivo de 64MB sobre Windows sin operaciones aritméticas

Referencias

[1] R. Prytherch, "Harrod's librarians' glossary and reference book (10th edition)," pp. 351–352, 2005, [Online; accessed 19. Feb. 2019].

[2] CISCO, "Cisco global cloud index: Forecast and methodology, 2016–2021 white paper," Noviembre 2018. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html#_Toc503317516

[3] M. O. Rabin, "Efficient dispersal of information for security, load balancing, and fault tolerance," *J. ACM*, vol. 36, no. 2, pp. 335–348, Apr. 1989. [Online]. Available: http://doi.acm.org/10.1145/62044.62050

[4] S. Sinclair, "EMC Elastic Cloud Storage Offers Resilient Scalability for the New Generation of Workloads," Oct. 2015.

[5] M. Q. Naquid, R. M. Jiménez, and J. L. G. Compeán, "The babel file system," in *2014 IEEE International Congress on Big Data*, June 2014, pp. 234–241.

[6] K. K. Mar, Z. Hu, C. Y. Law, and M. Wang, "Securing cloud data using information dispersal," in *2016 14th Annual Conference on Privacy, Security and Trust (PST)*, Dec 2016, pp. 445–448.

[7] M. Quezada-Naquid, R. Marcelín-Jiménez, and J. L. González-Compeán, "Babel:: The Construction of a Massive Storage System," *International Journal of Web Services Research*, vol. 13, no. 4, pp. 36–53, Oct. 2016. [Online]. Available: http://services.igi-global.com/resolvedoi/resolve.aspx?doi=10.4018/IJWSR.2016100103

[8] A. S. Tanenbaum and M. v. Steen, *Sistemas distribuidos: principios y paradigmas*. México: Pearson Educación, 2008.

[9] M. O. Rabin, *The Information Dispersal Algorithm and its Applications*. New York, NY: Springer New York, 1990, pp. 406–419. [Online]. Available: http://dx.doi.org/10.1007/978-1-4612-3352-7_32

[10] D. Zhao, K. Burlingame, C. Debains, P. Alvarez-Tabio, and I. Raicu, "Towards high-performance and cost-effective distributed storage systems with information dispersal algorithms," in *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2013, pp. 1–5.

[11] J. C. Moreira and P. G. Farrell, *Essentials of error-control coding*. Chichester: Wiley, 2006, oCLC: 936803535. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1002/9780470035726.app2/pdf

[12] Y. Dictionary, "Disperse," [Web; accedido el 26-06-17]. [Online]. Available: URL{http://www.yourdictionary.com/disperse}

[13] G. T. Informática, "Recovery," [Web; accedido el 26-06-17]. [Online]. Available: URL{http://www.tugurium.com/gti/termino.php?Tr=recovery%20file&Tp=T&Or=0}

[14] J. S. Plank, "A tutorial on reed–solomon coding for fault-tolerance in raid-like systems," *Software: Practice and Experience*, vol. 27, no. 9, pp. 995–1012, 1997. [Online]. Available: http://dx.doi.org/10.1002/(SICI)1097-024X(199709)27:9⟨995::AID-SPE111⟩3.0.CO;2-6

[15] A. B. Alnafoosi and T. Steinbach, "An integrated framework for evaluating big-data storage solutions - ida case study," in *2013 Science and Information Conference*, Oct 2013, pp. 947–956.

[16] Y.-D. Lyuu, *Information Dispersal*, ser. Cambridge International Series on Parallel Computation. Cambridge University Press, 1993, p. 8–25.

[17] H.-M. Sun and S.-P. Shieh, "Optimal information-dispersal for increasing the reliability of a distributed service," *IEEE Transactions on Reliability*, vol. 46, no. 4, pp. 462–472, Dec 1997.

[18] M. K. Nakayama and B. Yener, "Optimal information dispersal for probabilistic latency targets," *Computer Networks*, vol. 36, no. 5, pp. 695–707, 2001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128601001840

[19] A. B. Cheikh, H. Abbes, and G. Fedak, "Towards privacy for MapReduce on hybrid clouds using information dispersal algorithm," in *International Conference on Data Management in Cloud, Grid and P2P Systems*. Springer, 2014, pp. 37–48. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-319-10067-8_4

[20] A. Afianian, S. S. Nobakht, and M. B. Ghaznavi-Ghoushchi, "Energy-efficient secure distributed storage in mobile cloud computing," in *2015 23rd Iranian Conference on Electrical Engineering*, May 2015, pp. 740–745.

[21] Z.-t. Yu, Q. Qian, R. Zhang, and C.-L. Hung, "SIDA: An Information Dispersal Based Encryption Algorithm," in *Frontier Computing*, J. C. Hung, N. Y. Yen, and K.-C. Li, Eds. Singapore: Springer Singapore, 2016, vol. 375, pp. 239–249, dOI: 10.1007/978-981-10-0539-8_25. [Online]. Available: http://link.springer.com/10.1007/978-981-10-0539-8_25

[22] H. Weatherspoon and J. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," in *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, ser. IPTPS '01. London, UK, UK: Springer-Verlag, 2002, pp. 328–338. [Online]. Available: http://dl.acm.org/citation.cfm?id=646334.687814

[23] E. M. Hernandez-Ramirez, V. J. Sosa-Sosa, and I. Lopez-Arevalo, "A Comparison of Redundancy Techniques for Private and Hybrid Cloud Storage," *Journal of applied research and technology*, vol. 10, no. 6, pp. 893–901, 2012. [Online]. Available: http://www.scielo.org.mx/scielo.php?pid=S1665-64232012000600009&script=sci_arttext

[24] O. T. Lee, S. D. M. Kumar, and P. Chandran, "Erasure coded storage systems for cloud storage x2014; challenges and opportunities," in *2016 International Conference on Data Science and Engineering (ICDSE)*, Aug 2016, pp. 1–7.

[25] T. Ermakova and B. Fabian, "Secret sharing for health data in multi-provider clouds," in *2013 IEEE 15th Conference on Business Informatics*, July 2013, pp. 93–100.

[26] C. Hewlett Packard, "Cleversafe dsNet® Object Storage Software for HPE," Hewlett Packard Enterprise Development LP., Tech. Rep., 2015.

[27] IBM, "The definitive Guide to IBM Cloud Objetc Storage dispersed storage," © Copyright IBM Corporation 2016, IBM Cloud, Technical Whitepaper, Jul. 2016. [Online]. Available: https://www-01.ibm.com/common/ssi/cgi-bin/ssialias?htmlfid=TSW03335USEN

[28] S. Castañeda Hernández, A. Barrios Sarmiento, and I. Gutiérrez García, *Manual de Álgebra lineal*. Universidad del Norte Editorial, 2017, pp. 4–5.

[29] T. W. Judson and S. F. Austin, *Abstract Algebra Theory and Applications*. State University, Aug. 2018, ch. 22, p. 273. [Online]. Available: http://abstract.pugetsound.edu/download/aata-20180801.pdf

[30] K. Lundengård, J. Österberg, and S. Silvestrov, "Optimization of the determinant of the vandermonde matrix and related matrices," *Methodology and Computing in Applied Probability*, vol. 20, no. 4, p. 1418, Dec 2018. [Online]. Available: https://doi.org/10.1007/s11009-017-9595-y

[31] T. K. Moon, "BCH and RS codes: Designer cyclic codes," Spring 2006. [Online]. Available: http://ocw.usu.edu/Electrical_and_Computer_Engineering/Error_Control_Coding/lecture6.pdf

[32] E. Steiner, *The Chemistry Maths Book*. OUP Oxford, 2008, ch. 20, p. 584.

[33] L. R. Ojeda, *Análisis Numérico Básico*. Escuela Superior Politécnica del Litoral, ESPOL, 2011, ch. 4, pp. 64–67.

[34] B. Djordjevi and V. Timcenko, "Ext4 file system in linux environment: Features and performance analysis," in *International Journal of Computers*, 2012, pp. 37–45.

[35] A. S. Tanenbaum, *Sistemas Operativos Modernos*. México: Pearson Educación, 2009.

[36] L. Wirzenius, J. Oja, S. Stafford, A. Weeks, and R. I. Zurita, "Guía Para Administradores de Sistemas GNU/Linux," Apr 2003, [Online; accessed 4. Oct. 2018]. [Online]. Available: http://www.tldp.org/pub/Linux/docs/ldp-archived/system-admin-guide/translations/es/gasl.pdf

Casa abierta al tiempo
UNIVERSIDAD AUTÓNOMA METROPOLITANA

ACTA DE EXAMEN DE GRADO

No. 00079
Matrícula: 2163802651

Evaluación de rendimiento del Algoritmo de Dispersión de información sobre los campos finitos GF(2^8) y GF(2^{16}).

En la Ciudad de México, se presentaron a las 17:00 horas del día 4 del mes de noviembre del año 2019 en la Unidad Iztapalapa de la Universidad Autónoma Metropolitana, los suscritos miembros del jurado:

DR. FRANCISCO DE ASIS LOPEZ FUENTES
M. EN C. JOSE IGNACIO CASTILLO VELAZQUEZ
DR. LEONARDO PALACIOS LUENGAS

Bajo la Presidencia del primero y con carácter de Secretario el último, se reunieron para proceder al Examen de Grado cuya denominación aparece al margen, para la obtención del grado de:

MAESTRA EN CIENCIAS (CIENCIAS Y TECNOLOGIAS DE LA INFORMACION)

DE: BETZAYDA DEYANIRA VELAZQUEZ MENDEZ

y de acuerdo con el artículo 78 fracción III del Reglamento de Estudios Superiores de la Universidad Autónoma Metropolitana, los miembros del jurado resolvieron:

Aprobar

Acto continuo, el presidente del jurado comunicó a la interesada el resultado de la evaluación y, en caso aprobatorio, le fue tomada la protesta.

BETZAYDA DEYANIRA VELAZQUEZ MENDEZ
ALUMNA

REVISÓ

MTRA. ROSALIA SERRANO DE LA PAZ
DIRECTORA DE SISTEMAS ESCOLARES

DIRECTOR DE LA DIVISIÓN DE CBI

DR. JESUS ALBERTO OCHOA TAPIA

PRESIDENTE

DR. FRANCISCO DE ASIS LOPEZ FUENTES

VOCAL

M. EN C. JOSE IGNACIO CASTILLO VELAZQUEZ

SECRETARIO

DR. LEONARDO PALACIOS LUENGAS