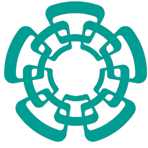


CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS DEL
INSTITUTO POLITÉCNICO NACIONAL

UNIDAD ZACATENCO
DEPARTAMENTO DE COMPUTACIÓN

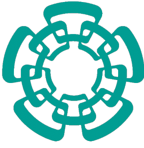
Paralelización de los algoritmos de cifrado simétrico AES-CTR y AES-OTR sobre un kit de desarrollo NVIDIA Jetson TK1

Tesis que presenta
Daniel Alberto Torres González

Para obtener el grado de
Maestro en Ciencias de la Computación

Director de la Tesis: **Dr. Amilcar Meneses Viveros**
Co-director de la Tesis: **Dr. Cuauhtémoc Mancillas López**

Ciudad de México Noviembre 2016



CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS DEL
INSTITUTO POLITÉCNICO NACIONAL

ZACATENCO CAMPUS
COMPUTER SCIENCE DEPARTMENT

Parallelization of the symmetric cipher algorithms AES-CTR and AES-OTR on a development kit NVIDIA Jetson TK1

Submitted by
Daniel Alberto Torres González

as the fulfillment of the requirement for the degree of
Master in Computer Science

Advisor: **Dr. Amilcar Meneses Viveros**
Co-advisor: **Dr. Cuauhtémoc Mancillas López**

Mexico City November 2016

Resumen

Actualmente muchas corporaciones y agencias gubernamentales se encuentran investigando nuevas formas de asegurar grandes volúmenes de información considerada sensible en intervalos de tiempo cortos. Para lograr esta tarea se requiere cifrar la información con un algoritmo criptográfico, el cual puede requerir de operaciones computacionales bastante costosas y por ende degradar el desempeño del equipo de cómputo en el que se ejecuta. La constante demanda de soluciones criptográficas eficientes ha crecido continuamente en diversas áreas durante la última década, como consecuencia del uso del Internet. En esta tesis se discuten implementaciones paralelas eficientes de los algoritmos criptográficos AES-CTR y AES-OTR. También se discute una optimización del modo de operación OTR. Las implementaciones se realizaron sobre un equipo móvil, el kit de desarrollo NVIDIA Jetson TK1, que cuenta con una arquitectura para una versión multinúcleo y una versión en muchos núcleos. Los resultados obtenidos en la arquitectura móvil muestran una aceleración de 3,92 y un rendimiento de 2,67 Gb/s en el modo de operación CTR, una aceleración de 2,32 y un rendimiento de 1,41 Gb/s en el modo de operación OTR, y una aceleración de 2,91 y un rendimiento de 1,68 Gb/s en la optimización propuesta al modo de operación OTR. Además, con la finalidad de demostrar que las implementaciones de los algoritmos paralelos son eficientes, se muestran pruebas realizadas sobre un servidor que incluye un microprocesador Intel I7 una tarjeta gráfica Tesla C2070. Los resultados obtenidos en el servidor muestran una aceleración de 21,105 y un rendimiento de 12,89 Gb/s en el modo de operación CTR, una aceleración de 2,303 y un rendimiento de 2,86 Gb/s en el modo de operación OTR, y una aceleración de 8,86 y un rendimiento de 11,01 Gb/s en la optimización propuesta al modo de operación OTR. Al contar con estas nuevas implementaciones los dispositivos que incluyan en sus arquitecturas componentes multinúcleo y GPU serán capaces de cifrar de una manera más eficiente una entrada de bytes proporcionados. Por consiguiente, AES-CTR y/o AES-OTR podrán realizar el cifrado de datos considerados sensibles en intervalos de tiempo cortos, permitiendo a otras aplicaciones aprovechar mayormente el uso de CPU como también de GPU, incrementando así la cantidad de información que puede ser cifrada/descifrada y ser transferida a través de un canal de comunicación si así se desea.

Abstract

Nowadays many corporations and government agencies are investigating new ways to protect big amounts of information considered really sensitive in short time intervals. To achieve this task it is required to encrypt the information with a cryptographic algorithm, which may need really expensive computational operations and thus degrade the computer equipment performance in which it is executed. The constant demand for efficient cryptographic solutions has increased continuouslly in many diverse areas during the last decade, as consequence of using the Internet. The main motivation of this thesis is to discuss efficient parallel implementations of the cryptographic algorithms AES-CTR and AES-OTR. It is also discussed an optimization for the operation mode OTR. The implementations were done on a mobile device, the development kit NVIDIA Jetson TK1, which includes an architecture for a multicore version and a manycore version. The results obtained on the mobile architecture show a speedup of 3,92 and a throughput of 2,67 Gb/s in the operation mode CTR, a speedup of 2,32 and a throughput of 1,41 Gb/s in the operation mode OTR, and a speedup of 2,91 and a throughput of 1,68 Gb/s in the proposed optimization to the operation mode OTR. Besides, in order to show that the parallel implementations are efficient, we show tests made on a server which includes an Intel I7 microprocessor and a graphic card Tesla C2070. The results obtained on the server show a speedup of 21,105 and a throughput of 12,89 Gb/s in the operation mode CTR, a speedup of 2,303 and a throughput of 2,86 Gb/s in the operation mode OTR, and a speedup of 8,86 and a throughput of 11,01 Gb/s in the proposed optimization to the operation mode OTR. By having these new implementations the devices that include components multicore and GPU in their architectures will be able to encrypt in a most efficient way the provided byte input. Consequently, AES-CTR and/or AES-OTR will be able to encrypt the information considered sentitive in short time intervals, allowing other applications to take advantage of the CPU platform as well of GPU, thus increasing the amount of information that can be encrypted/decrypted and transfer through a communication channel.

Agradecimientos

- A mi familia (padre, madre, hermano, hermanas, *neveu et beau-frère*)
- A los amigos de generación, del Laboratorio de Idiomas y a Reyna
- Al Dr. Amilcar Meneses Viveros por sus recomendaciones en el desarrollo y escritura de la tesis
- Al Dr. Cuauhtémoc Mancillas López por recibirme en Francia y brindarme las asesorías necesarias para el pleno desarrollo y escritura de la tesis
- A mis sinodales, Dr. Francisco Rodríguez Henríquez y Dr. Luis Gerardo de la Fraga
- Al Departamento de Computación del CINVESTAV-IPN
- Al Laboratorio de Idiomas del CINVESTAV-IPN
- Al CONACyT por el apoyo otorgado

Contenido

Resumen I

Abstract III

Agradecimientos V

Lista de acrónimos IX

Lista de figuras XI

Lista de tablas XIII

Lista de algoritmos XV

1. Introducción **1**
1.1. Antecedentes 1
1.2. Planteamiento del problema 3
1.3. Objetivos 4
1.3.1. General 4
1.3.2. Particulares 4
1.4. Organización de la tesis 6

2. SoC Tegra de NVIDIA y el kit Jetson TK1 **7**
2.1. GPU con arquitectura Kepler GK110/210 9
2.1.1. Arquitectura de una unidad SMX 10
2.1.1.1. Planificador de warps y acceso a registros 10
2.1.1.2. Instrucción aleatoria 10
2.1.1.3. Operaciones atómicas 11
2.1.2. Subsistema de memoria Kepler 12
2.2. SoC Tegra K1 12
2.3. Tarjeta Jetson TK1 14
2.3.1. Sistema operativo 14
2.3.2. Arquitectura 15
2.3.3. JetPack TK1 16

3. Modos de operación de cifradores por bloque **19**
3.1. Definiciones . . . 21
3.2. Modo de Encadenamiento de Bloques de Cifrado . . . 21
3.3. Modo de Encadenamiento de Bloques de Cifrado en Propagación . . . 23
3.4. Modo de Retroalimentación Cifrada . . . 24
3.5. Modo de Retroalimentación de Salida . . . 25
3.6. Modo de Libro de Códigos Electrónico . . . 27
3.7. Modo Contador . . . 28
3.8. Modo Contador de Galois . . . 29
3.9. Modo de Dos-Rondas de Desplazamiento . . . 32
3.10. Resumen de características . . . 38

4. Estándar de Cifrado Avanzado **39**
4.1. Definiciones básicas . . . 40
4.2. Transformaciones . . . 42
4.2.1. Suma de la llave de ronda . . . 42
4.2.2. Sustitución de bytes . . . 42
4.2.3. Corrimiento de filas . . . 43
4.2.4. Mezclado de columnas . . . 44
4.3. Transformaciones inversas . . . 44
4.4. Expansión de llave . . . 45
4.5. Optimización con cajas-T . . . 46

5. Implementaciones clásicas **49**
5.1. Variables constantes y estructuras . . . 49
5.2. Cifrador AES optimizado con cajas-T . . . 50
5.3. Expansión de llave . . . 52
5.4. AES-CTR secuencial . . . 52
5.5. AES-OTR secuencial . . . 54

6. Algoritmos paralelos y sus implementaciones **59**
6.1. AES-CTR multinúcleo . . . 59
6.2. AES-CTR CUDA . . . 61
6.3. AES-OTR multinúcleo . . . 63
6.4. AES-OTR CUDA . . . 67
6.5. Optimización de AES-OTR . . . 70

7. Ejecución de algoritmos paralelos y sus resultados **75**
7.1. Almacenamiento y tiempos medidos . . . 75
7.2. Lectura y escritura . . . 76
7.3. Subida y bajada desde GPU . . . 76
7.4. Ejecuciones secuenciales y paralelas . . . 77
7.5. Rendimiento . . . 82

8. Conclusiones **87**

Apéndices 93

Apéndice A. Preliminares 95
A.1. Grupos . . . 95
A.2. Anillos . . . 95
A.3. Campos . . . 96
A.4. Campos finitos . . . 96
A.5. Polinomios sobre un campo . . . 97
A.6. Operaciones en polinomios . . . 97
A.6.1. Adición . . . 97
A.6.2. Multiplicación . . . 98
A.6.3. Doblado . . . 99

Apéndice B. Tiempos de ejecución 101
B.1. Lectura vs escritura . . . 102
B.2. Subida vs bajada en GPU . . . 103
B.3. CTR . . . 104
B.4. OTR . . . 106
B.5. OTR optimizado . . . 109

Apéndice C. Rendimiento 111
C.1. CTR . . . 112
C.2. OTR . . . 113
C.3. OTR optimizado . . . 114

Apéndice D. Aceleraciones 115
D.1. CTR . . . 116
D.2. OTR . . . 117
D.3. OTR optimizado . . . 118

Apéndice E. Resultados en servidor 119
E.1. CTR . . . 119
E.2. OTR . . . 120
E.3. OTR optimizado . . . 120

Bibliografía 120

Lista de acrónimos

AES	Advanced Encryption Standard
API	Application Programming Interface
CBC	Cipher Block Chaining
CFB	Cipher Feedback
CMAC	Cipher-based Message Authentication Code
CPU	Central Process Unit
CTR	Counter
CUDA	Compute Unified Device Architecture
DES	Data Encryption Standard
ECB	Electronic Codebook
GCM	Galois Counter Mode
GF	Galois Field
GMAC	Galois Message Authentication Code
GMU	Grid Management Unit
GPU	Graphics Processing Unit
HMAC	Hash-based Message Authentication Code
ISP	Image Signal Processor
L4T	Linux for Tegra
MAC	Message Authentication Code
MPI	Message Passing Interface
NIST	National Institute of Standards and Technology
OFB	Output Feedback

OMP	OpenMP
OTR	Offset Two-Rounds
PCBC	Propagating Cipher Block Chaining
SMX	Streaming Multiprocessor
SoC	System on a Chip
VI	Vector de Inicialización
XOR	O Exclusiva

Lista de figuras

2.1. Arquitectura SoC . . . 8
2.2. Arquitectura SMX . . . 11
2.3. Subsistema de memoria Kepler . . . 13
2.4. Arquitectura Tegra K1 . . . 14
2.5. Arquitectura Jetson TK1 . . . 15

3.1. Cifrado CBC . . . 22
3.2. Descifrado CBC . . . 22
3.3. Cifrado PCBC . . . 23
3.4. Descifrado PCBC . . . 24
3.5. Cifrado CFB . . . 25
3.6. Descifrado CFB . . . 25
3.7. Cifrado OFB . . . 26
3.8. Descifrado OFB . . . 26
3.9. Cifrado ECB . . . 27
3.10. Descifrado ECB . . . 27
3.11. Cifrado CTR . . . 29
3.12. Descifrado CTR . . . 29
3.13. Cifrado GCM . . . 31
3.14. Descifrado GCM . . . 31
3.15. Generación de L en OTR . . . 32
3.16. Cifrado OTR . . . 33
3.17. Cifrado OTR última partición par . . . 35
3.18. Cifrado OTR última partición impar . . . 35
3.19. Generación de Q en OTR . . . 37
3.20. Generación de la etiqueta TA OTR . . . 37
3.21. Generación de las etiquetas TE y T OTR . . . 38

6.1. Generación de variables α_i y β_i . . . 71
6.2. Cifrado OTR optimizado . . . 72
6.3. Cifrado OTR optimizado última partición par . . . 73
6.4. Cifrado OTR optimizado última partición impar . . . 73

7.1. Tiempos CTR . . . 78
7.2. Tiempos OTR . . . 80
7.3. Tiempos OTR optimizado . . . 82

7.4. Rendimiento CTR 83
7.5. Rendimientos OTR 85
7.6. Rendimientos OTR optimizado 86

Lista de tablas

2.1. Especificaciones Tegra K1 13
2.2. Distribuciones en Jetson TK1 15
2.3. Características Jetson TK1 16
2.4. Expansiones extra en la Jetson TK1 16
2.5. Software en el JetPack TK1 17
2.6. APIs en la Jetson TK1 17

3.1. Características de modos de operación 38

4.1. Caja-S 43

8.1. Cantidades recomendadas para obtener ganancia en aceleración 89
8.2. Resumen de resultados Jetson TK1 90
8.3. Resumen de resultados servidor 90

B.1. Tiempos promedio de lectura y escritura 102
B.2. Tiempos promedio de subida y bajada en GPU 103
B.3. Tiempos promedio de cifrado CTR OMP 104
B.4. Tiempos promedio de cifrado CTR CUDA 105
B.5. Tiempos promedio de cifrado OTR OMP 106
B.6. Tiempos promedio de cifrado OTR CUDA (total) 107
B.7. Tiempos promedio de cifrado OTR CUDA (4 GB) 108
B.8. Tiempos promedio de cifrado OTR optimizado OMP 109
B.9. Tiempos promedio de cifrado OTR optimizado CUDA 110

C.1. Rendimiento CTR 112
C.2. Rendimiento OTR 113
C.3. Rendimiento OTR optimizado 114

D.1. Aceleraciones CTR 116
D.2. Aceleraciones OTR 117
D.3. Aceleraciones OTR optimizado 118

E.1. Tiempos de cifrado CTR OMP en servidor 119
E.2. Tiempos de cifrado CTR CUDA en servidor 119
E.3. Tiempos de cifrado OTR OMP en servidor 120
E.4. Tiempos de cifrado OTR CUDA en servidor 120

E.5. Tiempos de cifrado OTR optimizado OMP en servidor 120
E.6. Tiempos de cifrado OTR optimizado CUDA en servidor 120

Lista de algoritmos

1. Cifrador AES . . . 40
2. Expansión de llave . . . 45
3. Cifrador AES cajas-T . . . 51
4. Expandir k_j a partir de k_i . . . 52
5. Expansión de llave optimizado . . . 52
6. Cifrador AES-CTR OpenSSL . . . 53
7. Descifrador AES-CTR OpenSSL . . . 53
8. Cifrador AES-OTR secuencial . . . 55
9. Descifrador AES-OTR secuencial . . . 56
10. Generador de la etiqueta de autenticación TA en OTR . . . 57
11. Cifrador AES-CTR multinúcleo OMP . . . 59
12. Descifrador AES-CTR multinúcleo OMP . . . 60
13. Cifrador AES-CTR CUDA . . . 61
14. Descifrador AES-CTR CUDA . . . 62
15. Cifrador AES-OTR multinúcleo OMP . . . 65
16. Descifrador AES-OTR multinúcleo OMP . . . 66
17. Cifrador AES-OTR CUDA . . . 68
18. Descifrador AES-OTR CUDA . . . 69
19. Cifrador AES con 4 rondas . . . 71

CAPÍTULO 1

Introducción

Los continuos avances tecnológicos han producido una evolución en el desarrollo de los nuevos microprocesadores utilizados en los dispositivos de cómputo. Los nuevos desarrollos no se enfocan en incrementar la frecuencia de un solo microprocesador, si no en incrementar el número de núcleos dentro del mismo chip; a este modelo de microprocesadores se le conoce como *microprocesadores multinúcleo* (*multicore*). En el diseño de estos microprocesadores multinúcleo se busca brindar un bajo consumo energético y a la vez obtener un alto rendimiento computacional al ejecutar algoritmos especializados, a pesar de las limitaciones que puedan llegar a existir en el equipo de cómputo al momento de realizar algún trabajo sobre el microprocesador.

Debido a la competencia en el mercado de tarjetas gráficas se nos ha permitido tener acceso a una gran cantidad de tarjetas con extraordinaria capacidad de cómputo. Además, las industrias que se dedican al desarrollo de este tipo de tarjetas han comenzado a proveer herramientas de desarrollo para explotar debidamente su poder de cómputo. Gracias a estas herramientas, los desarrolladores ahora pueden acceder a recursos poderosos, pero también están forzados a replantear los algoritmos que fueron diseñados para trabajar en microprocesadores con un solo núcleo.

Las redes de cómputadoras forman parte cotidianamente en nuestras vidas en prácticamente todos los lugares del mundo, y es debido a esto que la seguridad y privacidad se han convertido en grandes preocupaciones. La protección de datos sensibles juega un rol muy importante en la sociedad moderna. Muchas áreas necesitan garantizar confidencialidad, integridad y origen de los datos. Los servidores que son utilizados para administrar comunicaciones seguras son responsables de mantener una gran cantidad de conexiones simultáneas, asegurando calidad óptima en el servicio, minimizando el tiempo de cómputo y reduciendo la duración de las transacciones. Por esta razón, los servidores de protección deben ser equipados con múltiples microprocesadores o tarjetas aceleradoras multinúcleo con co-procesadores dedicados. Desafortunadamente, el costo de esta infraestructura es muy elevado.

1.1. Antecedentes

Una Unidad de Procesamiento Gráfico (GPU, del inglés Graphics Processing Unit) es un circuito electrónico especialmente diseñado para acceder y manipular rápidamente las diferentes memorias que se encuentran dentro de su propia arquitectura. Cuando un proceso

necesita utilizar la GPU para realizar operaciones, dependiendo de la implementación de dicho proceso, su rendimiento computacional se incrementa al realizar varias operaciones al mismo tiempo y al ahorrar tiempo de lectura y/o escritura en memoria. Los usos más comunes que se asignan a una GPU es la creación de gráficos en general, debido a que este tipo de cálculos requieren de operaciones repetitivas independientes una de otra, y al realizar varios cálculos al mismo tiempo (en paralelo) sobre un conjunto de información de gráficos se provoca un decremento bastante considerable en los tiempos de ejecución de las aplicaciones. Pero además de ser capaces de trabajar eficientemente en la creación de gráficos, las GPUs también poseen un diseño optimizado para trabajar con operaciones de punto flotante y es debido a esto que, tanto industrias como investigadores, buscan nuevas formas de aprovechar su capacidad de procesamiento masivo de información en cálculos computacionalmente intensos [**?**].

Tomando en cuenta que la arquitectura de la GPU es altamente paralela, al requerirse un procesamiento sobre una gran cantidad de información, para algunas tareas en particular, las GPUs son más eficientes que las arquitecturas basadas en Unidad de Proceso Central (CPU, del inglés Central Process Unit), ya que la GPU puede ejecutar varias instrucciones en paralelo, obteniendo el mismo resultado que la CPU y por ende, reduciendo el tiempo de ejecución de la aplicación. La GPU puede ser vista como un dispositivo capaz de ejecutar un número de procesos (hilos) en paralelo y por medio de un código especial llamado *código de núcleo* (o *código de kernel*) puede inicializarlos en diferentes bloques de procesos. Por ejemplo, una ventaja de utilizar una GPU es que se puede aligerar la carga de trabajo de la CPU en aplicaciones que involucren procesar grandes cantidades de imágenes 3D interactivas (e.g. un videojuego en tiempo real). De esta forma, mientras todo lo relacionado con los gráficos se procesa en la GPU, la CPU puede dedicarse a otro tipo de cálculos.

Hoy en día los sistemas computacionales están pasando de realizar el procesamiento central en la CPU a realizar un co-procesamiento paralelo (e incluso distribuido) entre la CPU y la GPU con la intención de optimizar el rendimiento computacional de las aplicaciones [**?**][**?**]. Los sistemas de cómputo que en la actualidad suelen incluir dentro de su arquitectura una GPU son los sistemas empotrados, dispositivos móviles, computadoras personales, estaciones de trabajo, consolas de videojuegos, etc. Los nuevos desarrollos de GPU conducen al incremento de paralelismo masivo y con esto se permiten nuevos desarrollos de software de propósito general.

Para facilitar la creación de nuevas aplicaciones altamente paralelas sobre GPUs la corporación NVIDIA ha inventado la Arquitectura Unificada de Dispositivos de Cómputo (CUDA, del inglés Compute Unified Device Architecture) [**?**][**?**], la cual es una plataforma de cómputo paralelo y un modelo de programación que permite disminuir el tiempo de ejecución de algoritmos paralelos y aplicaciones paralelas, incrementando el rendimiento computacional de un proceso y aprovechando eficientemente la GPU. En este modelo de programación el desarrollador tiene libertad de seleccionar los espacios de memoria a utilizar, la información que será compartida entre hilos, las dimensiones del bloque de hilos, etc. La selección de la memoria a utilizar depende de diferentes factores, como puede ser la velocidad de acceso, la cantidad de memoria requerida y las operaciones a realizar sobre la información almacenada. En muchos casos el rendimiento de las aplicaciones depende de la estrategia de uso de memoria y debido a esto la complejidad de diseñar una aplicación eficiente se acrecienta.

Diseñar e implementar un algoritmo paralelo para ser ejecutado en una CPU multinúcleo

es complicado debido a la problemática que conlleva realizar este tipo de procesos, como puede ser la compartición de información requerida por los procesos, las sincronizaciones que deben existir para evitar malos manejos de memoria y/o corrupción de datos, entre otras cosas. Peor aún, diseñar e implementar un algoritmo paralelo para ser ejecutado en una GPU es mucho más complicado que con la CPU, debido a que existen aún más restricciones relacionadas con la arquitectura de la GPU, tomando en cuenta que existe una gran variedad de arquitecturas, la latencia en la transferencia de datos desde la GPU hacia la CPU o viceversa, mayores limitaciones en la cantidad de memoria disponible, velocidad de tranferencia del bus PCIe debido a que se puede convertir en un cuello de botella al momento de transferir grandes cantidades de información, número de bloques de hilos a ser ejecutados, número de hilos por bloque, etc.

Hoy en día existen arquitecturas industriales especializadas en diferentes temas, como pueden ser comunicación en redes, procesamiento de señales o criptografía [?]; escribir aplicaciones en estas arquitecturas suele ser muy complicado, pero a cambio proporcionan un rendimiento mucho más elevado en el área para la que fueron desarrollados [?]. En particular, nos enfocaremos en la criptografía, debido a que en los últimos años se ha convertido en un tema de gran interés para los usuarios y se han desarrollado varias extensiones de hardware [?][?][?], provocando que los algoritmos criptográficos puedan ser acelerados en arquitecturas multinúcleo e incluso en GPU [?][?][?][?][?][?][?].

1.2. Planteamiento del problema

Día a día cantidades inmensas de información son generadas en diversos dispositivos de cómputo alrededor del mundo. Mucha de esta nueva información puede ser distribuida a un conjunto de usuarios con intereses muy específicos. Para lograr que la información sea visualizada solamente por usuarios que han sido autorizados a descubrir el contenido, ésta debe protegerse de lecturas no autorizadas ocultando el contenido original, de tal forma que sea completamente ilegible al momento de ser transferida por un canal de comunicación. Además, cuando un usuario autorizado recibe la información ilegible, ésta se debe poder transformar nuevamente en la información original. Estas transformaciones se deben realizar a altas velocidades de procesamiento para aprovechar eficientemente los componentes del dispositivo de cómputo y evitar un mayor consumo energético. Estas características se pueden conseguir con ayuda de un algoritmo criptográfico implementado sobre una arquitectura altamente paralela [?][?].

Uno de los algoritmos criptográficos (de llave simétrica) que ha generado gran interés entre la comunidad científica y que además es el algoritmo de mayor uso a nivel mundial es el Estándar de Cifrado Avanzado (AES, del inglés Advanced Encryption Standard) [?][?], debido a que es relativamente fácil de implementar en arquitecturas limitadas y a que proporciona la seguridad suficiente en cualquier sistema que se implemente. Pero además de necesitar de AES para proteger datos, también se necesita de un algoritmo especial llamado *modo de operación*[?][?] para describir cómo realizar operaciones repetitivas sobre grandes cantidades de datos y asegurarlos. Esta tesis hace un enfoque sobre los modos de operación Contador (CTR, del inglés Counter)[?][?] y Dos-Rondas de Desplazamiento (OTR, del inglés Offset Two-Rounds)[?].

Una plataforma en la que es posible desarrollar este tipo de algoritmos criptográficos

y hacer buen uso de ellos es la plataforma NVIDIA Jetson TK1 [?], creada especialmente para procesamiento en paralelo con la GPU en sistemas empotrados móviles. Tomando en cuenta que la información es generada en dispositivos móviles de uso cotidiano (e.g. laptops, smartphones, tablets) o más especializados a investigación y/o seguridad nacional (e.g. cluster, servidor, drone espía), los cuales están siendo desarrollados en dispositivos que poseen arquitecturas con componentes capaces de soportar paralelismo, esta tesis propone paralelizar los algoritmos criptográficos AES-CTR y AES-OTR en la plataforma móvil Jetson TK1 para minimizar los tiempos de cifrado/descifrado y maximizar la cantidad de datos cifrados/descifrados en intervalos de tiempo cortos.

1.3. Objetivos

1.3.1. General

El principal objetivo de la tesis es generar nuevos diseños e implementaciones paralelas optimizadas del algoritmo de cifrado simétrico estandarizado AES-CTR y del algoritmo de cifrado simétrico en competencia AES-OTR sobre un kit de desarrollo NVIDIA Jetson TK1, de tal forma que, para una entrada de datos, los algoritmos sean capaces de ejecutar varias instancias de AES en paralelo sobre la CPU o sobre la GPU, para así disminuir los tiempos de ejecución del proceso de cifrado/descifrado de datos, registrando todos los resultados y comparándolos contra una implementación secuencial de AES-CTR hecha con las bibliotecas estandarizadas de OpenSSL y contra una implementación secuencial de AES-OTR escrita por el autor, para finalmente corroborar que el rendimiento de los algoritmos criptográficos ha sido mejorado.

1.3.2. Particulares

Para alcanzar por completo el objetivo general de la tesis, primero se deben alcanzar una serie de objetivos previos:

- **Investigación del algoritmo AES:** con el fin de comprender enteramente la forma en que el algoritmo AES cifra/descifra la información de entrada es necesario realizar una búsqueda extensa de literatura científica que explique el funcionamiento en cada paso del algoritmo. En particular, se debe investigar sobre: las transformaciones y sus respectivas operaciones sobre un bloque de datos, cómo realizar la expansión de llaves de ronda utilizando la llave inicial, mejores formas de optimizar las operaciones bit a bit en campos finitos, entre otras cosas.
- **Investigación de los modos de operación CTR y OTR:** lo que se busca es conseguir optimizaciones bastante eficientes. Para ello se deben tomar en cuenta los modos de operación; al conocer diferentes modos de operación se puede tener una mejor visión de cómo se lleva a cabo el cifrado de datos. Al explorar diferentes modos de operación se podrá concluir que el modo CTR es el más factible a paralelizar sin afectar la seguridad de los datos. Por lo tanto, se logrará obtener optimizaciones eficientes. Si además de cifrar datos se necesita autenticarlos, al realizar una investigación sobre el modo de operación OTR se vislumbrará que este nuevo modo de operación resulta ser el adecuado para realizar dicha tarea.

- **Comprobación de dependencias de datos:** cuando se paraleliza un algoritmo, en la mayoría de los casos, se llegan a localizar dependencias de datos; es decir, no se puede generar un dato i sin antes generar el dato $i-1$ y a la vez, no se puede generar el dato $i-1$ sin antes generar el dato $i-2$ y, de esta forma, hasta llegar al caso base, que normalmente es el dato que se proporciona como entrada transformado de alguna forma en particular. Al momento de pretender paralelizar modos de operación normalemente se llegan a encontrar este tipo de dependencias; esto ocurre en el modo OTR. Por lo tanto, se tiene que llevar a cabo un análisis a fondo del modo de operación para verificar qué pasos contienen dependencias de datos y lidiar con este tipo de percances que afectan el paralelismo. Por su inherente paralelismo, en el modo CTR este tipo de dependencias de datos son inexistentes entre diferentes bloques de datos.

- **Análisis de algoritmos secuenciales y diseño de algoritmos paralelos AES-CTR y AES-OTR:** ya con las dependencias de datos localizadas, se busca diseñar versiones paralelas de ambos algoritmos, tanto para ser implementados y ejecutados en una CPU como en una GPU. En este diseño son tomados en cuenta varios factores que afectan el rendimiento de la implementación como son: la arquitectura de la plataforma en la que se ejecutan los algoritmos, la cantidad de memoria disponible tanto en CPU como en GPU, velocidades de acceso a las distintas memorias localizadas en la GPU, cantidad de cores en CPU y GPU, etc.

- **Desarrollo de los algoritmos paralelos propuestos:** con el fin de establecer tiempos de referencia base para hacer las comparaciones correspondientes entre las versiones paralelas propuestas de AES-CTR y AES-OTR contra la versión más común de ambos algoritmos, que es la versión secuencial, es necesario implementar inicialmente una versión secuencial de AES-CTR con bibliotecas estandarizadas de OpenSSL y una versión secuencial de AES-OTR con códigos optimizados escritos por el autor. Finalmente, al comprobar que dichas implementaciones secuenciales producen resultados correctos, se procede a escribir las optimizaciones paralelas de ambos algoritmos.

- **Pruebas sobre los algoritmos paralelos propuestos y comparación de resultados:** las pruebas son ejecutadas sobre el kit de desarrollo NVIDIA Jetson TK1 y sobre un servidor con arquitectura paralela, con entradas de diferentes tamaños, desde 16 B hasta 4 GB de datos, capturando el tiempo que toma a los algoritmos efectuar una actividad en particular sobre un conjunto de bloques de datos, como puede ser: lectura, escritura, cifrado, descifrado, subida y bajada del equipo anfitrión (CPU) a la tarjeta gráfica (GPU), ésto último exclusivamente para la versión implementada en CUDA. Cada prueba realizada a los algoritmos es ejecutada un total de diez veces sobre una misma entrada de datos, desde las versiones secuenciales de AES-CTR y AES-OTR hasta sus versiones paralelas en CPU y GPU, esto para poder obtener un resultado promedio entre las diez ejecuciones de cada algoritmo y proceder con las comparaciones de resultados.

1.4. Organización de la tesis

La tesis se encuentra organizada de la siguiente manera: en el capítulo 2 se lleva a cabo una descripción de los componentes incluídos en las nuevas GPU de NVIDIA con arquitectura Kepler y los beneficios que se pueden obtener al utilizar este tipo de arquitecturas en la optimización de algoritmos y aplicaciones. En el mismo capítulo se presenta una descripción general del kit de desarrollo NVIDIA Jetson TK1, desde los componentes de hardware hasta los componentes de software.

En el capítulo 3 se mencionan algunos modos de operación que son utilizados en la actualidad en conjunto con cifradores por bloques, explicando cada uno de ellos. Los modos de operación seleccionados están estandarizados por el Instituto Nacional de Estándares y Tecnología (NIST, del inglés National Institute of Standards and Technology), exceptuando uno de ellos, que se encuentra en competencia.

En el capítulo 4 se describe el algoritmo AES. En esta descripción se incluyen las definiciones básicas que son utilizadas en el algoritmo, la descripción a fondo de las transformaciones que son requeridas para cifrar un bloque de datos, el proceso de expansión de la llave inicial y la optimización cajas-T (T-box) utilizada para pre-calcular dos transformaciones.

En el capítulo 5 se presenta el algoritmo AES optimizado con caja-T que fue implementado para realizar el proceso de cifrado de datos, además del algoritmo para expansión de llaves también optimizado. De igual manera, se incluyen los algoritmos secuenciales clásicos AES-CTR y AES-OTR.

En el capítulo 6 se muestran los algoritmos paralelos AES-CTR y AES-OTR generados durante el trabajo, tanto la versión multinúcleo como la versión muchos núcleos con CUDA, describiendo la forma en que se decidió implementarlos y explicando el por qué. También se incluye una optimización adicional al modo de operación OTR.

Casi para finalizar, en el capítulo 7 se incluyen las comparaciones de las ejecuciones, tanto de los algoritmos secuenciales como de los algoritmos paralelos.

Finalmente, las conclusiones de este trabajo se presentan en el capítulo 8.

CAPÍTULO 2

SoC Tegra de NVIDIA y el kit Jetson TK1

Una plataforma móvil normalmente incluye un microprocesador ARM de tipo RISC (del inglés Reduced Instruction Set Computing); esto es, el conjunto de instrucciones que se pueden ejecutar en esta arquitectura es bastante reducido en comparación con los de tipo complejo, como es la computadora ordinaria. Al ejecutar menos instrucciones la arquitectura se vuelve menos versátil, pero se gana eficiencia con respecto al consumo energético. Este tipo de arquitecturas son mucho más baratas de producir, además de que ocupan poco espacio.

Un Sistema en un Chip (SoC, del inglés System on a Chip) es un circuito integrado que incluye los componentes de una computadora u otro sistema electrónico en un mismo chip. Los diseños modernos integran múltiples núcleos dentro del SoC; por lo tanto, las aplicaciones empotradas comenzaron a modificar sus sistemas basados en un solo procesador a sistemas multiprocesador, de tal manera que fueran capaces de manejar comunicaciones intensivas de datos entre componentes del SoC o incluso, entre diferentes SoCs. Este tipo de aplicaciones demandan un alto rendimiento computacional combinado con bajo consumo energético y por lo tanto, necesitan utilizar arquitecturas multiprocesador dentro del mismo chip, dotadas con infraestructuras de comunicación complejas. Con el paso del tiempo, la tendencia de contruir SoCs multiprocesador ha sido acelerada de acuerdo a los requerimientos de las aplicaciones embebidas actuales [**?**].

En el año 2008, la corporación NVIDIA comenzó a abordar nuevos temas relacionados con la creación de SoCs, generando así los primeros diseños y desarrollos que poco tiempo después dieron a conocer como el SoC Tegra [**?**], cuyo objetivo principal era el procesamiento multimedia mejorado. Debido al éxito obtenido por este nuevo desarrollo, diversas corporaciones decidieron adoptar esta arquitectura para construir nuevos sistemas empotrados móviles (e.g. reproductores multimedia), y además enfocar los nuevos desarrollos de software (e.g. navegadores de Internet para dispositivos móviles) para operar especialmente en esta arquitectura altamente paralela.

Con el paso del tiempo la demanda de nuevos desarrollos en SoCs fue creciendo, provocando que los diseños de las arquitecturas anteriores se mejoraran. La evolución en los SoCs se dio de tal forma que se comenzó a proporcionar soporte para una variedad mayor de sistemas operativos (e.g. Ubuntu Linux, Android), para una variedad mayor de arquitecturas (e.g. Dual-core, Quad-core) e incluso, para sistemas de cómputo móvil más sofisticados, como puede ser el sistema de procesamiento de imágenes montado sobre un vehículo [**?**].

Los desarrollos de este tipo de dispositivos se enfocan a mejorar el desempeño en juegos de

tiempo real manteniendo un bajo consumo energético, integrándolos directamente en consolas de videojuegos, mejorando considerablemente la calidad de los gráficos. Además, grandes cantidades de dispositivos móviles, tales como smartphones, tablets, laptops, dispositivos móviles de Internet, etc., también han comenzado a integrar SoCs en sus arquitecturas. Lo más usual en un dispositivo NVIDIA de esta naturaleza es incluir en su arquitectura una CPU ARM multinúcleo, una GPU con arquitectura Kepler, un chip controlador de memoria[1] (MMC, del inglés Memory Chip Controller), puente norte[2] y puente sur[3], todo dentro del mismo circuito. La figura 2.1 muestra un diagrama básico del sistema tradicional de un SoC.

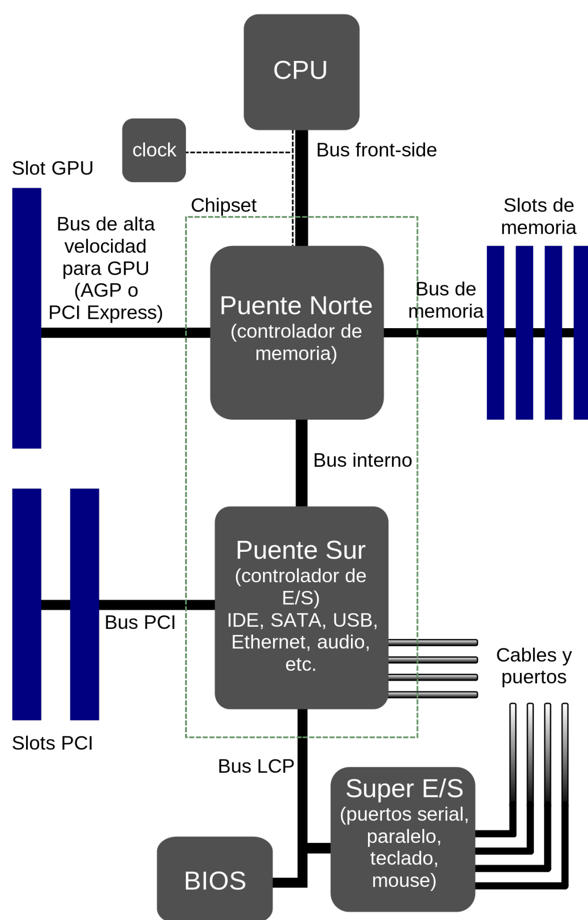


Figura 2.1: Arquitectura tradicional montada en un SoC

En la sección siguiente se hará un enfoque más detallado en el tipo de GPU integrada en un SoC Tegra moderno de NVIDIA; en particular en la arquitectura Kepler GK110/210, que se definirá en la sección siguiente.

[1]circuito digital que administra el flujo de datos hacia y desde la memoria principal de una computadora

[2]chip dentro de un circuito, conectado directamente con la CPU, responsable de las tareas que requieren mayor rendimiento

[3]chip dentro de un circuito, conectado a puertos de E/S, lleva a cabo las tareas de menor rendimiento

2.1. GPU con arquitectura Kepler GK110/210

Las GPU con arquitectura Kepler GK110/210 [?] simplifican el desarrollo de nuevos programas altamente paralelos ofreciendo mucho más capacidad de procesamiento que generaciones de GPU anteriores, proporcionan nuevos métodos para optimizar e incrementar el nivel de paralelismo en una aplicación y permiten aprovechar mayormente la GPU, consumiendo mucho menos energía y generando menos calentamiento en el SoC. Está diseñada para brindar operaciones rápidas de doble precisión para acelerar grandes cargas de trabajo. Las implementaciones en versión completa incluyen 15 unidades Multiprocesador de Flujo (SMX, del inglés Streaming Multiprocessor) y controladores de memoria de 64 bits. Por supuesto, diferentes productos utilizarán diferentes configuraciones. Algunas características importantes son: la arquitectura SMX y un subsistema de memoria mejorado.

Esta arquitectura está enfocada al rendimiento computacional, incorporando características particulares diseñadas especialmente para ser poderosas herramientas de procesamiento paralelo. Entre las características más importantes se encuentran:

- Paralelismo dinámico: se añade la capacidad para que la GPU genere su propio trabajo de forma automática, sincronice resultados y controle la planificación de ese trabajo por medio de rutas de hardware aceleradas, todo esto sin involucrar la CPU. Al proveer esta característica para adaptar la cantidad y forma de paralelismo en tiempo de ejecución, los programadores pueden crear diversos tipos de procesamiento en paralelo y así, desarrollar algoritmos capaces de aprovechar más eficientemente la GPU. Además, con el paralelismo dinámico se permite realizar tareas más complejas para ser ejecutadas fácilmente, permitiendo planificar gran parte de las instrucciones directamente en la GPU. Por lo tanto, los programas son más fáciles de generar y la CPU permanecerá disponible para realizar otras tareas.

- Hyper-Q: da la oportunidad a múltiples núcleos en la CPU de asignar trabajo en la misma GPU simultaneamente, incrementando el uso de GPU y reduciendo el uso de CPU. Incrementa el número de colas de trabajo entre el anfitrión y la GPU, permitiendo 32 conexiones en paralelo administradas por hardware. Es una solución que permite generar conexiones diferentes en flujos múltiples, ya sea desde procesos Interfaz de Paso de Mensajes (MPI, del inglés Message Passing Interface) o incluso procesos con múltiples hilos. Las aplicaciones que previamente se veían afectadas por falsas serializaciones entre tareas, limitando el uso de GPU, pueden incrementar su desempeño sin necesidad de modificar el código ya existente.

- Unidad de Administración de Malla (GMU, del inglés Grid Management Unit): para poder habilitar el paralelismo dinámico se necesita una buena administración de malla[4] y un buen sistema de planificación. La GMU GK110 gestiona y da prioridad a mallas para ser ejecutadas en la GPU. Puede interrumpir la planificación de nuevas mallas, encolarlas y/o suspenderlas, hasta que la GPU esté disponible para ejecutarlas nuevamente. Además asegura que las cargas de trabajo tanto de la CPU como de la GPU son planificadas adecuadamente.

[4]Conjunto de bloques de hilos organizados en una, dos o tres dimensiones. El número de bloques que conforman una malla es usualmente determinado por la cantidad de datos a ser procesados o por el número de microprocesadores en el sistema

- NVIDIA GPUDirect: es una característica que permite a las GPU localizadas dentro una misma computadora o localizadas en servidores diferentes, conectados a través de una red, intercambiar datos directamente sin la necesidad de acceder al sistema de memoria de alguna GPU en particular. Con esto se permite decrementar significativamente la latencia de mensajes MPI con destino a la memoria de la GPU.

2.1.1. Arquitectura de una unidad SMX

La unidad SMX (incluída en las GPUs con arquitectura Kepler GK110/210) ha sido creada especialemente para procesar grandes cargas de trabajo que requieren de operaciones computacionales de doble precisión. Cada unidad SMX cuenta con 192 núcleos CUDA, y cada uno de ellos cuenta con unidades aritméticas para punto flotante y para valores enteros. El objetivo de una unidad SMX es incrementar el rendimiento de doble precisión en la GPU y, para ello también cuenta con 64 unidades de doble precisión. Además incluye 32 unidades de función especial para aproximaciones eficientes de operaciones computacionales. En la figura 2.2 se muestra un diagrama con los componentes integrados en una unidad SMX.

2.1.1.1. Planificador de warps y acceso a registros

Al momento de comenzar una ejecución, la SMX planifica hilos paralelos en grupos de 32, llamados *warps*. Cada SMX incluye cuatro planificadores de warps y ocho unidades de planificación, lo que permite a cuatro warps ser ejecutados concurrentemente.

Cuando se ejecuta un gran número de hilos en paralelo, muchos de estos pueden requerir acceder a una gran variedad de registros, provocando un cuello de botella en el bus de datos y ralentizando el acceso. Es por esto que el número de registros que pueden ser accedidos por un hilo está definido en 255 registros. De esta forma se permite a las aplicaciones hacer uso de los registros sin sacrificar el número de hilos que pueden acceder concurrentemente en una SMX y por lo tanto, los códigos que necesiten de accesos masivos de hilos a registros pueden obtener aceleraciones considerables.

2.1.1.2. Instrucción aleatoria

Para mejorar el rendimiento en la arquitectura Kepler se implementó la instrucción aleatoria, que permite la compartición de datos entre hilos dentro de un warp. Antes, el compartir datos entre hilos requería de un almacenamiento previo por separado y posteriormente se debían ejecutar operaciones de carga para transferir los datos a través de la memoria compartida. Con la instrucción aleatoria, los hilos dentro del warp pueden leer valores de otros hilos en el warp, sin importar qué hilo sea, a lo que se le conoce como *referencias indexadas arbitrarias*. Además, esta característica ofrece una ventaja sobresaliente en el rendimiento de la memoria compartida, ya que provoca que las operaciones de almacenamiento y carga sean ejecutadas en un solo paso. También puede reducir la cantidad de memoria compartida requerida por un bloque de hilos, dado que la información compartida en el bloque no necesita ser almacenada nuevamente en otra sección de la memoria compartida.

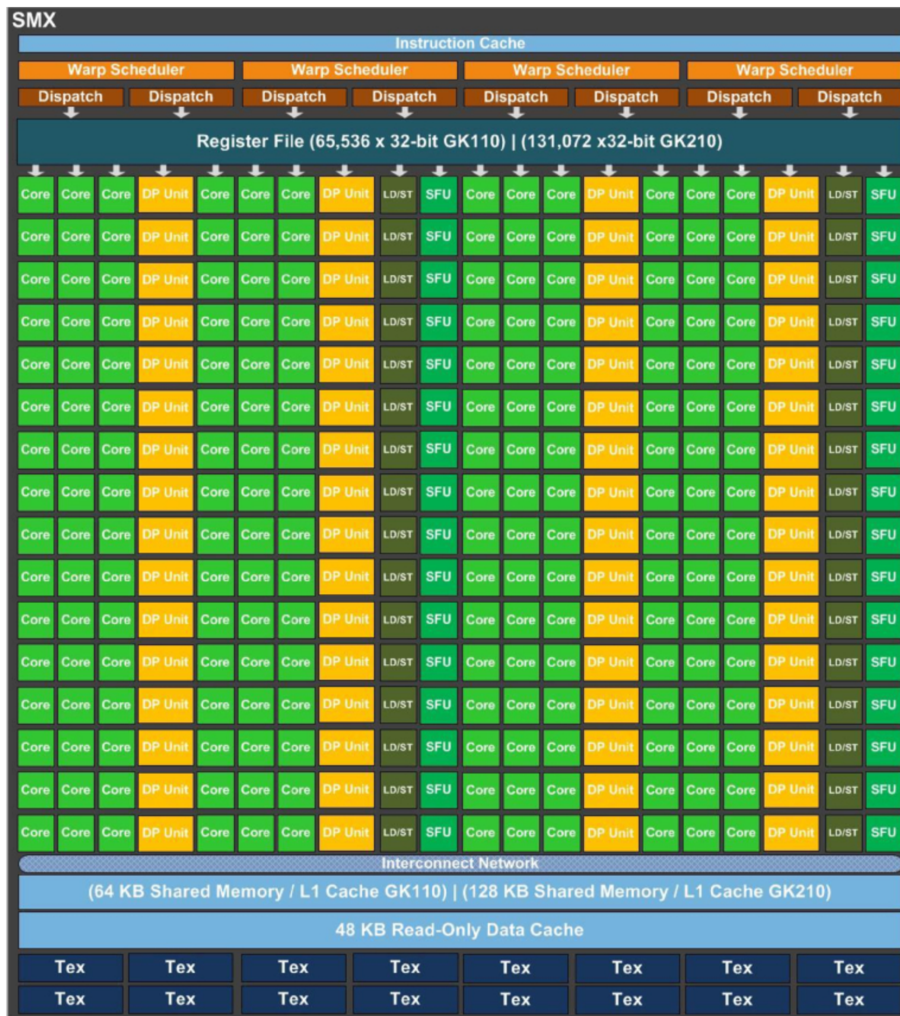


Figura 2.2: Arquitectura general de una unidad SMX, con 192 núcleos CUDA, 64 unidades de doble precisión, 32 unidades de función especial y 32 unidades de carga y almacenamiento

2.1.1.3. Operaciones atómicas

Las operaciones atómicas[5] en memoria son muy importantes en la programación concurrente y paralela, debido a que permiten a los hilos realizar correctamente operaciones de lectura, modificación y escritura en la memoria compartida por varios hilos. Por ejemplo, las operaciones de suma, resta, valor mínimo, valor máximo o de comparación de valores son consideradas atómicas porque las operaciones de lectura, modificación y escritura son ejecutadas sin la interrupción de otros hilos.

En la arquitectura Kepler GK110/210, el rendimiento de las operaciones atómicas ha sido optimizado. Una operación atómica sobre una dirección de memoria global puede ser realizada en un ciclo de reloj. Una operación atómica sobre direcciones de memoria independientes

[5]operación indivisible o no interrumpible que debe ser ejecutada en su totalidad en un sistema dado y que además parece ocurrir instantáneamente

también fue acelerada significativamente, además de haber hecho más eficiente el manejo de conflictos de direcciones. Estos incrementos en las velocidades provoca que este tipo de operaciones puedan ser utilizadas con frecuencia dentro de ciclos internos en una invocación de algún núcleo en particular, eliminando el proceso de reducción que normalmente sería requerido por algunos algoritmos para consolidar los resultados finales de su ejecución.

2.1.2. Subsistema de memoria Kepler

El subsistema de memoria para arquitecturas Kepler ha sido adaptado para operar con cuatro tipos de memoria: memoria compartida, caché L1 por unidad SMX, caché L2 y caché de sólo-lectura. Cada una de estas memorias tiene características particulares:

- Memoria compartida y caché L1 configurables:

  En la arquitectura Kepler GK110, cada SMX tiene 64 KB de memoria que puede ser configurada con 48 KB de memoria compartida y 16 KB de caché L1 o viceversa. Kepler ahora permite una flexibilidad adicional en la configuración de alojamiento de memoria compartida y caché L1, permitiendo un particionamiento de 32/32 KB entre memoria compartida y caché L1.

  Para la arquitectura GK210, la cantidad de memoria configurable fue doblada a 128 KB, permitiendo un máximo de 112 KB para memoria compartida y 16 KB para caché L1. Otra posible configuración de memoria es 32 KB caché L1 y 96 KB de memoria compartida, o 48 KB de caché L1 y 80 KB de memoria compartida.

- 48 KB de caché sólo-lectura:

  En la arquitectura Kepler se introduce un caché de datos con 48 KB que es conocido por ser de sólo-lectura durante la ejecución de alguna función. Este caché tiene acceso directamente a la unidad SMX para operaciones generales de carga. Además, el ancho de banda soporta accesos de alta velocidad a memoria no alineada. El uso de esta memoria puede ser gestionado automáticamente por el compilador o explícitamente por el programador.

- Caché L2 mejorada:

  Las GPU Kepler GK110/210 tienen integrada una memoria dedicada caché L2 con 1536 KB de espacio. El caché L2 es el primer punto de unificación de datos entre las unidades SMX, gestionando todas las peticiones de carga y almacenamiento, y proporcionando compartición de datos a alta velocidad entre la GPU.

La figura 2.3 muestra la jerarquía utilizada en el subsistema de memoria Kepler.

2.2. SoC Tegra K1

Tegra K1 [?] es un SoC móvil creado por la corporación NVIDIA que se caracteriza por haber sido diseñado con una arquitectura Kepler y con tecnología móvil que incluye soporte para varias tecnologías orientadas a juegos, como DirectX 11 u OpenGL 4.4, y que además es el primer SoC móvil con soporte para CUDA. Básicamente dentro de su arquitectura

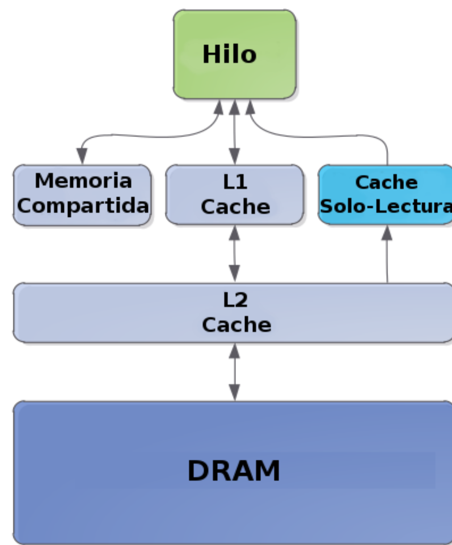


Figura 2.3: Jerarquía del subsistema de memoria en una arquitectura Kepler

se encuentran los mismos componentes con los que cuenta una GPU moderna, más aparte todos los componentes generalmente integrados en un SoC Tegra, y además proporciona una ventaja extra, que es el bajo consumo energético. Esto permite que los nuevos sistemas empotrados creados con un SoC Tegra K1 incluído en su arquitectura utilizen el mismo código CUDA que puede ejecutarse en una GPU tradicional de escritorio, brindando niveles similares en las aceleraciones y en el rendimiento de la GPU optimizando el desempeño del sistema en general. En la tabla 2.1 se describen las especificaciones técnicas más importantes del SoC Tegra K1 y en la figura 2.4 se muestra un diseño de la arquitectura general del SoC.

	Especificación
CPU	NVIDIA 4-Plus-1 ARM Cortex-A15 Quad-Core “r3” a 2.32 GHz
GPU	NVIDIA con arquitectura Kepler “GK20a” con 192 núcleos CUDA
Memoria	8 GB DDR3L y LPDDR3 con 40 bits de extensión de dirección
Video	LCD 3840x2160 y HDMI UltraHD 4096x2160

Tabla 2.1: Especificaciones técnicas del SoC Tegra K1

Algunos ejemplos de dispositivos que ya incluyen dentro de su arquitectura un SoC Tegra K1 son: NVIDIA SHIELD Tablet, NVIDIA SHIELD Portable, NVIDIA Jetson TK1, Google Nexus 9, Google Project Tango tablet, Acer Chromebook 13 CB5-311-T1UU, Acer Chromebook 13 CB5-311-T7NN, Nabi Big Tab(24”), Nabi Big Tab(20”), HP Chromebook 14 G3, ASUS Transformer Pad TF701, Lenovo ThinkVision 28, LG G2 MINI D625, EVGA Tegra Note 7, entre otros varios. En particular, se hará un enfoque más detallado sobre el kit de desarrollo NVIDIA Jetson TK1.

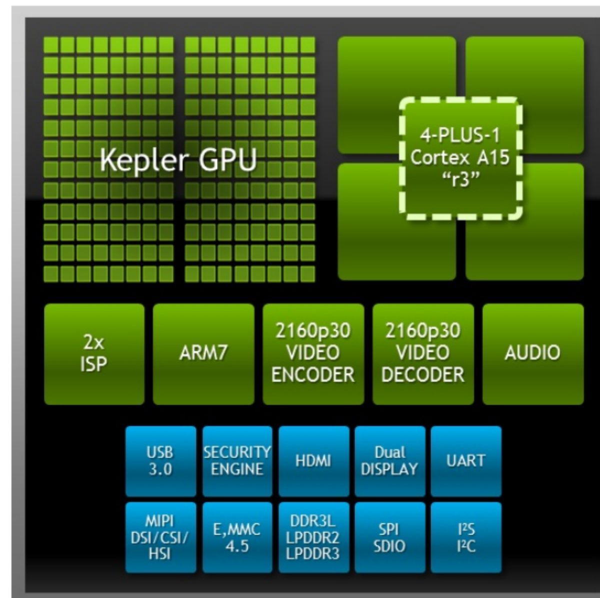


Figura 2.4: Arquitectura general de un SoC Tegra K1

2.3. Tarjeta Jetson TK1

El kit de desarrollo o tarjeta NVIDIA Jetson TK1 es una plataforma de desarrollo empotrada para sistemas Linux creada en abril del 2014 por la corporación NVIDIA, en la cual se integra un SoC Tegra K1 con una característica extra integrada: un Procesador de Señal de Imagen (ISP, del inglés Image Signal Processor) dentro del mismo SoC. La GPU integrada está especialmente diseñada para procesamiento masivo acelerado en paralelo y alto rendimiento, además de garantizar un bajo consumo energético, con lo que el kit también se convierte en una poderosa herramienta para visualización por computadora, procesamiento de imágenes y procesamiento de datos en tiempo real para proyectos empotrados, aprendizaje de máquina u otras tareas que requieran de operaciones computacionales intensivas. Puede ser montada fácilmente en sistemas móviles empotrados como drones, sistemas robóticos autónomos, equipo para procesamiento de imágenes médicas, etc.

2.3.1. Sistema operativo

La tarjeta Jetson TK1 viene con un sistema operativo NVIDIA Linux for Tegra (L4T) pre-instalado, que básicamente es un sistema Ubuntu versión 14.04, pero con controladores especiales pre-configurados para mejorar la carga del sistema, las operaciones a nivel de núcleo, las operaciones de OpenGL, el manejo de X.Org, el manejo multimedia, etc. Existe también soporte oficial para instalar otras distribuciones de Linux en la tarjeta que mantienen escencialmente el núcleo requerido por la tarjeta. Algunas de las distribuciones oficiales son listadas en la tabla 2.2.

Distribución	Descripción
Fedora	instalación mínima del sistema con un entorno XFCE
openSUSE	instalación mínima del sistema con un entorno XFCE
openSUSE (SD/SATA)	ejecuta openSUSE directamente desde una tarjeta SD o un disco duro SATA sin modificar el contenido en memoria eMMC
Arch	instalación mínima del sistema
Gentoo	ejecuta Gentoo directamente desde la memoria eMMC
Gentoo (SD)	ejecuta Gentoo directamente desde una tarjeta SD sin modificar el contenido en memoria eMMC
Busybox	crea un sistema de archivos raíz mínimo

Tabla 2.2: Distribuciones oficiales disponibles para ser cargadas en la Jetson TK1

2.3.2. Arquitectura

Además de la CPU y la GPU, la Jetson TK1 incluye componentes similares a una Raspberry Pi[6], pero también incluye puertos más orientados a una computadora personal, como un puerto SATA, mini-PCIe y un ventilador especial que mejora el enfriamiento del SoC permitiéndole permanecer en funcionamiento durante más tiempo con cargas de trabajo bastante grandes. La figura 2.5 muestra el diagrama con la arquitectura de la tarjeta y en la tabla 2.3 se describen algunas características incluídas en la arquitectura de la Jetson TK1.

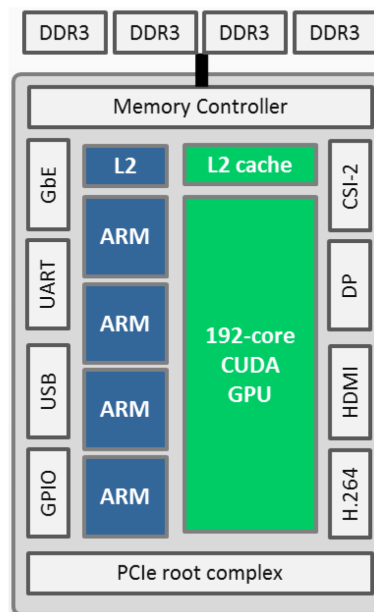


Figura 2.5: Arquitectura de la tarjeta Jetson TK1

Independientemente de los componentes ya mencionados, se pueden adaptar dispositivos

[6]mini tarjeta de cómputo con SoC integrado (CPU ARM, GPU y memoria)

Característica	Descripción
Dimensión	127 mm x 127 mm
SoC	Tegra K1 con CPU + GPU + ISP en el mismo chip
Consumo de potencia en SoC	1 $\sim$ 5 Watts
CPU	NVIDIA 4-Plus-1 ARM Cortex-A15 Quad-Core a 2.32GHz (con core de ahorro de energía)
GPU	NVIDIA Kepler “GK20a” con 192 SM3.2 núcleos CUDA
Memoria	2GB DDR3L a 933MHz EMC x16
Almacenamiento	16 GB fast eMMC 4.51
mini-PCIe	ranura de extensión para WiFi, Firewire o Ethernet
SD/MMC	ranura para lectura de tarjetas SD/MMC
USB 3.0	conector hembra tipo-A
USB 2.0	conector hembra micro-AB
HDMI	puerto de conexión HDMI
RS232	puerto serial DB9
Audio	códec Realtek HD ALC5639 con jacks de entrada y salida
Ethernet	puerto LAN RTL8111GS Realtek 10/100/1000 Base-T Gigabit
SATA	puerto que soporta discos de 2.5” y 3.5”
JTAG	puerto 2x10-pin 0.1”
Energía	jack de 12V en corriente directa y conector de 4-pin para IDE
Ventilador	ventilador trabajando con 12V

Tabla 2.3: Características incluídas en el kit de desarrollo Jetson TK1

extra a través de un puerto de expansión de 125 pines integrado a la tarjeta. En la tabla 2.4 se muestran las posibles opciones de expansión de hardware.

Característica extra	Descripción
Puertos de cámara	2 puertos de alta velocidad CSI-2 MIPI
Puerto LCD	panel LVDS y eDP
Puertos touchscreen	touch SPI 1x4-lane + 1x1-lane SI-2
I2C	3 puertos
GPIO	7 pines GPIO de 1.8V
UART	-
HSIC	-

Tabla 2.4: Posibles expansiones extra a través del puerto de 125 pines

2.3.3. JetPack TK1

Otra característica importante a mencionar es el JetPack TK1, el cual es un conjunto de paquetes de software compatibles con el sistema L4T, que contiene una Interfaz de Programación de Aplicaciones (API, del inglés Application Programming Interface) especializada

en aceleración de hardware y herramientas extra necesarias para desarrollar aplicaciones en esta plataforma. Dentro del contenido se encuentran imágenes del sistema operativo L4T en distintas versiones para actualizaciones o restauraciones, ejemplos de códigos y grandes cantidades de documentación para impulsar a los desarrolladores a iniciar nuevos proyectos en esta plataforma, disponible para ser cargado tanto en la tarjeta como en el sistema anfitrión. La tabla 2.5 lista el software proporcionado en el JetPack TK1, mietras que en la tabla 2.6 se listan las APIs de aceleración que pueden ser cargadas en el sistema para facilitar el desarrollo de aplicaciones.

Software	**Descripción**
Bibliotecas	CUDA Toolkit para anfitrión Ubuntu, CUDA Toolkit para Jetson TK1, OpenCV, etc.
Imágenes de sistemas operativos	Sistema de archivos muestra derivado de Ubuntu
Ejemplos	Ejemplos de código para NVIDIA GameWorks OpenGL
Documentación	Grandes cantidades de documentación para orientar a los nuevos desarrolladores en el uso de las herramientas y APIs incluídas
Tegra Graphics Debugger	Consola que facilita el proceso de depuración en OpenGL en sus distintas versiones, permitiendo a los desarrolladores de gráficos obtener mejores resultados
Tegra System Profiler	Muestreo de uso de CPU con vista interactiva de datos capturados
PerfKit	Herramientas de rendimiento para ayudar en el proceso de depuración en alto y bajo nivel en aplicaciones OpenGL y Direct3D

Tabla 2.5: Software incluído en el JetPack TK1

	API
1	CUDA 6.0 o superior
2	OpenGL 4.4 o superior
3	OpenGL ES 3.1 o superior
4	OpenMAX IL multimedia codec incluyendo soporte H.264, VC-1 y VP8
5	NPP (primitivas optimizadas para CUDA)
6	OpenCV4Tegra (NEON + GLSL + optimizaciones en la CPU)
7	VisionWorks

Tabla 2.6: APIs soportadas en la Jetson TK1

Cabe destacar que este conjunto de paquetes está diseñado para ser instalado exclusivamente en sistemas con un anfitrión Ubuntu 12.04/14.04 con arquitecturas de 64 bits. Al cargar esta variedad de software especializado en el sistema es más factible generar aplicaciones eficientes capaces de aprovechar óptimamente los recursos de la tarjeta.

CAPÍTULO 3

Modos de operación de cifradores por bloque

Un sistema criptográfico puede ocultar el contenido de un mensaje cualquiera transformándolo antes de transmitirlo o almacenarlo. Las técnicas requeridas para proteger los datos son parte de la criptografía. De acuerdo a [?], la criptografía es definida como:

Disciplina que estudia las técnicas matemáticas relacionadas a la seguridad de la información, tal que es capaz de proveer servicios de seguridad como confidencialidad, integridad de datos, autenticación y no-repudio.

Particularmente, en esta tesis se hará un enfoque sobre la *criptografía simétrica* o *criptografía de llave privada*. La criptografía simétrica es un esquema criptográfico que provee confidencialidad e integridad de datos. Utiliza una misma llave para cifrar y descifrar datos. Las entidades que se intentan comunicar deben establecer un acuerdo sobre la llave a utilizar. Una vez que las entidades tienen acceso a esta llave, el emisor cifra los datos a transmitir con un algoritmo criptográfico de llave simétrica, lo manda al receptor y éste lo descifra con el mismo algoritmo criptográfico en modo de descifrado y la misma llave.

Como consecuencia del uso de Internet en diversas áreas, la demanda de soluciones criptográficas eficientes ha crecido continuamente en la última década. La criptografía de llave simétrica se utiliza en muchos escenarios distintos, como puede ser *e-banking* o *e-commerce*, ya que este tipo de organizaciones requieren de altos estándares para garantizar la seguridad de la información con que se trabaja. Sin embargo, el rendimiento computacional se ve decrementado al ejecutar este tipo de algoritmos ya que se requiere procesar volúmenes de datos muy grandes. Con el fin de reducir tiempos de cifrado/descifrado de datos y aprovechar mayormente los componentes de los dispositivos actuales se debe considerar construir aceleradores dedicados a la criptografía.

Ahora bien, un modo de operación es un algoritmo que utiliza un cifrador por bloques (i.e. AES, DES) para transformar mensajes de tamaño arbitrario, de tal forma que es capaz de proporcionar confidencialidad o autenticidad. Dado que el cifrador por bloques sólo se aplica para asegurar la transformación criptográfica de un grupo de bits con tamaño fijo (un bloque), ya sea para cifrar o descifrar, se requiere de un modo de operación para describir cómo aplicar repetidamente la transformación sobre un conjunto de bloques y asegurar grandes cantidades de datos.

Una característica en varios modos de operación es que necesitan de una secuencia de bits aleatorios, llamado Vector de Inicialización (VI), para el proceso de transformación de

datos. Comúnmente, el VI es utilizado en un paso inicial del proceso de cifrado y en el paso correspondiente en el descifrado de datos. El VI no tiene que ser secreto, pero debe ser diferente e impredecible en cada nueva ejecución de una transformación, ya que se ocupa para asegurar que mensajes cifrados diferentes son producidos, incluso cuando el mensaje en claro es el mismo. Esto significa que si el mismo mensaje en claro es transformado múltiples veces con la misma llave, el resultado siempre será distinto uno de otro debido al VI. Es de suma importancia nunca reutilizar el mismo VI junto con la misma llave, ya que esto revela información sobre el primer bloque del mensaje a cifrar y esto implica pérdida de seguridad en algunos modos de operación; en otros modos, implica la destrucción total de la seguridad.

Los cifradores por bloques pueden tener varios tamaños de bloques, pero durante la transformación el tamaño siempre es fijo. Algunos modos de operación necesitan que cada bloque ingresado sea del tamaño especificado, pero puede llegar a ocurrir que el último bloque a transformar no esté completo. En caso de ocurrir esto, el último bloque debe ser completado. Existen varias formas de realizar este proceso, siendo la más simple rellenar con bytes nulos hasta completar el tamaño correcto u otra un poco más compleja, agregar un bit 1 seguido de varios bits 0, entre otras más. En cada forma de relleno se debe tener cuidado al recobrar la longitud original del mensaje, ya que al no manejar adecuadamente el reintegro del mensaje se puede perder o revelar información no deseada. Sin embargo, existen modos que no requieren este trabajo extra sobre el último bloque y por lo tanto, son capaces de transformar secuencias de bits de longitudes arbitrarias.

A través de los años, los modos de operación han sido estudiados para considerar errores de propagación que pueden llegar a existir, y en investigaciones más actuales también se considera la protección de integridad. Varios modos estandarizados son capaces de proporcionar confidencialidad, pero no son capaces de proteger contra modificación accidental o intencional de datos, las cuales pueden ser detectadas con un Código de Autenticación de Mensaje (MAC, del inglés Message Authentication Code) por separado, o con una firma digital[1]. Cuando los investigadores se dieron cuenta que es muy complicado generar un modo de operación capaz de proveer confidencialidad y autenticidad a la vez, comenzaron a crear modos que proveen confidencialidad e integridad, y además brindaron soluciones alternativas como el Código de Autenticación de Mensaje Basado en Hash (HMAC, del inglés Hash-based Message Authentication Code), el Código de Autenticación de Mensaje Basado en Cifrado (CMAC, del inglés Cipher-based Message Authentication Code) y el Código de Autenticación de Mensaje de Galois (GMAC, del inglés Galois Message Authentication Code), los cuales fueron siendo aprobados al paso del tiempo. En el 2001, el NIST revisó su lista de modos de operación aprobados [**?**] para incluir a AES como el cifrador por bloques predeterminado.

Hoy en día los modos de operación son revisados y definidos por un gran número de reconocidas organizaciones internacionales para garantizar la seguridad de la información que circula a través de los canales de comunicación de Internet. En las secciones siguientes se describirán algunos modos de operación estandarizados por el NIST, más otros capaces de proporcionar autenticidad.

[1] mecanismo que permite al receptor de un mensaje firmado digitalmente determinar la entidad que originó dicho mensaje y confirmar que el mensaje no ha sido alterado desde que fue firmado

3.1. Definiciones

Sean X y Y cadenas binarias, su concatenación se denota como $X||Y$ y su longitud en bits como $|X|$. El conjunto de todas las cadenas binarias se denota como $\{0,1\}^*$, y $\{0,1\}^n$ es el conjunto que contiene las cadenas binarias de longitud n. Los elementos de $\{0,1\}^n$ pueden ser vistos como elementos del campo finito GF(2^n), en polinomios de grado a los más $n-1$ y con coeficientes en $\{0,1\}$ o símplemente como la representación hexadecimal. La suma en dicho campo está definida como una $\oplus$ de bits, mientras que el producto se define como $A(x)B(x) \mod p(x)$, donde $p(x)$ es un polinomio de grado n irreducible en GF(2). En construcción de algoritmos de criptografía simétrica se utiliza frecuentemente la operación $xA(X) \mod p(x)$, que es la multiplicación de un polinomio $A(x)$ por la variable x.

El polinomio $A(x)$ por la variable x se denota como $xtimes(A)$. Dicha operación puede ser calculada muy eficientemente utilizando únicamente un corrimiento a la izquierda y algunas $\oplus$, ya que representa una multiplicación por 2 con reducción modular, por lo que en ocasiones se denota como $2L$, donde L es una cadena binaria de n-bits. Un cifrador por bloques es una función $E_K : \{0,1\}^n \times \{0,1\}^k \rightarrow \{0,1\}^n$, donde n es su tamaño de bloque y k es el tamaño de la llave. Un cifrador por bloques tiene una función inversa denotada como E_K^{-1}, por lo tanto es una permutación de valores de $\{0,1\}^n$.

Un cifrador por bloques sólo puede cifrar cadenas de bits P_i de longitud fija n-bits, donde P_i representa una cadena de bits en claro. La operación inversa de un cifrador por bloques sólo puede descifrar cadenas de bits C_i de longitud fija n-bits, donde C_i representa una cadena de bits cifrados. En aplicaciones de la vida real por lo general los mensajes procesados son mucho mayores de n-bits y en casos muy especiales más pequeños. Para poder manejar mensajes mayores al tamaño de bloque del cifrador por bloques se utilizan los modos de operación.

3.2. Modo de Encadenamiento de Bloques de Cifrado

El modo de Encadenamiento de Bloques de Cifrado (CBC, del inglés Cipher Block Chaining) [?][?][?] ha sido el modo de operación más utilizado. CBC se utiliza en distribuciones de OpenSSL. Para cifrar un conjunto de bloques se requiere aplicar una operación O Exclusiva (XOR) entre un bloque en claro i y el bloque cifrado anterior $i-1$, para después ingresar el resultado al cifrador por bloques (modo cifrado) con la llave secreta K y obtener el bloque cifrado i resultante. El único bloque cifrado que no depende del bloque anterior es el primer bloque, donde en vez de utilizar el bloque cifrado anterior, se utiliza un VI generado pseudo-aleatoriamente. De esta forma, cada bloque cifrado depende de todos los bloques procesados anteriormente y esta dependencia de datos provoca que el CBC en modo cifrado no pueda ser paralelizado. Para descifrar un conjunto de bloques se toma un bloque cifrado i y se ingresa directamente al cifrador por bloques (modo descifrado) con la llave secreta K, para después aplicar una operación XOR entre el bloque cifrado anterior $i-1$ y el bloque resultante del cifrador por bloques, recuperando así el bloque en claro i. El primer bloque se maneja de forma un poco diferente; la operación XOR se realiza entre el bloque resultante del cifrador por bloques al ingresar el primer bloque y el VI generado con anterioridad. A diferencia del proceso de cifrado, el descifrado sí puede ser paralelizado, debido a que un bloque en claro i no depende de ningún otro bloque en claro, si no sólamente de dos bloques cifrados

adyacentes. Los procesos de cifrado y descifrado de este modo de operación son descritos con las ecuaciones (3.1) y de igual manera, las figuras 3.1 y 3.2 muestran diagramas por bloque que dejan ver más claramente el funcionamiento.

$$
\begin{aligned}
C_i &= \begin{cases} E_K(P_i \oplus VI), \text{si } i = 0 \\ E_K(P_i \oplus C_{i-1}), \text{si } i > 0 \end{cases} \\
P_i &= \begin{cases} E_K^{-1}(C_i) \oplus VI, \text{si } i = 0 \\ E_K^{-1}(C_i) \oplus C_{i-1}, \text{si } i > 0 \end{cases}
\end{aligned}
\tag{3.1}
$$

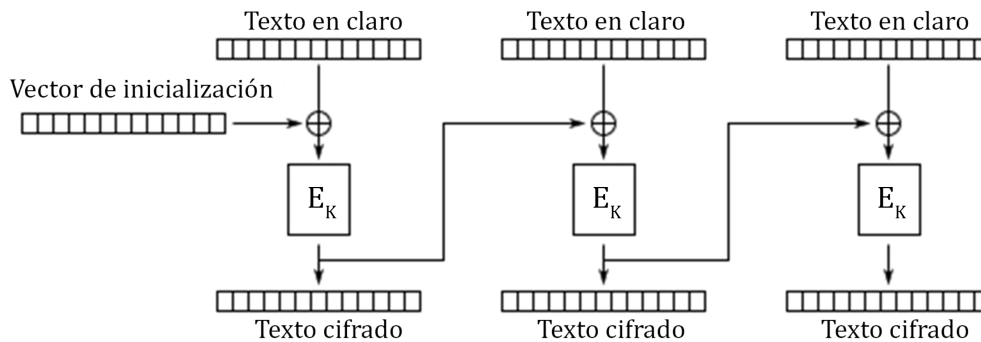


Figura 3.1: CBC en modo cifrado

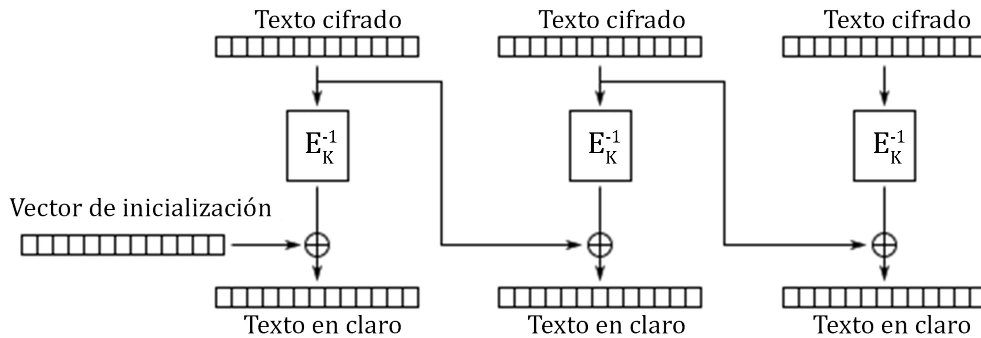


Figura 3.2: CBC en modo descifrado

Una característica de este modo de operación es que al modificar (intencionalmente o accidentalmente) un solo bit en un bloque cifrado, será imposible recuperar dos bloques del mensaje en claro original. Esto es consecuencia de la dependencia de datos existente entre dos bloques cifrados. Un detalle más de este modo de operación es que si el último bloque de entrada no está completo, éste debe ser rellenado forzosamente, provocando trabajo computacional extra.

3.3. Modo de Encadenamiento de Bloques de Cifrado en Propagación

El modo de Encadenamiento de Bloques de Cifrado en Propagación (PCBC, del inglés Propagating Cipher Block Chaining) [?] fue diseñado para propagar indefinidamente cualquier cambio efectuado sobre los bloques cifrados, de manera tal que al intentar descifrarlos, éstos cambios no permitan obtener los bloques en claro originales. Para cifrar un conjunto de bloques se aplica una operación XOR entre cada bloque en claro i y el resultado de otra operación XOR entre el bloque en claro anterior $i-1$ y el bloque cifrado anterior $i-1$, para después ingresar el bloque resultante al cifrador por bloques (modo cifrado) con la llave secreta K y así obtener el bloque cifrado i. Al igual que el modo CBC, en el proceso de descifrado se ingresan directamente los bloques cifrados al cifrador por bloques (modo descifrado) con la llave secreta K y al resultado se le aplica una operación XOR entre el bloque en claro anterior $i-1$ y el bloque cifrado anterior $i-1$, para así generar el bloque en claro i. En ambos procesos, para obtener el primer bloque, ya sea en claro o cifrado, se utiliza un VI. En este modo de operación la dependencia de datos es aún mayor que en el modo CBC, debido a que la generación de un bloque i depende forzosamente de la generación correcta de los dos bloques anteriores $i-1$, y es por esto que resulta imposible paralelizar tanto el cifrado como el descifrado de datos. Ambos procesos de este modo de operación son descritos con las ecuaciones (3.2), y las figuras 3.3 y 3.4 describen gráficamente los algoritmos mencionados.

$$
\begin{aligned}
C_i &= \begin{cases} E_K(P_i \oplus VI), \text{si } i = 0 \\ E_K(P_i \oplus P_{i-1} \oplus C_{i-1}), \text{si } i > 0 \end{cases} \\
P_i &= \begin{cases} E_K^{-1}(C_i) \oplus VI, \text{si } i = 0 \\ E_K^{-1}(C_i) \oplus P_{i-1} \oplus C_{i-1}, \text{si } i > 0 \end{cases}
\end{aligned}
\tag{3.2}
$$

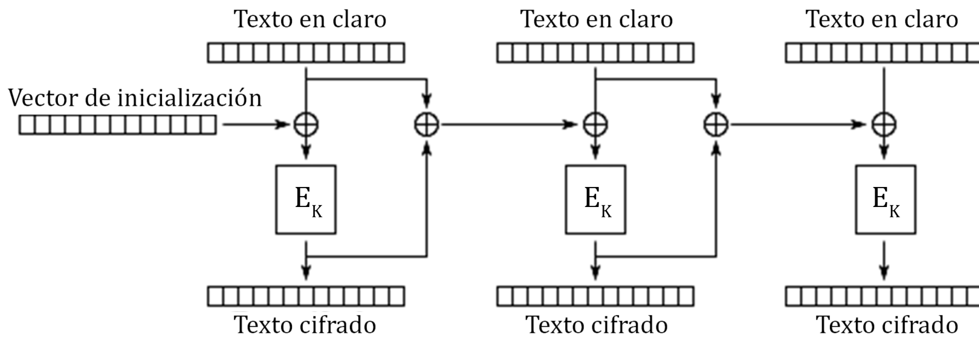


Figura 3.3: PCBC en modo cifrado

PCBC es un modo de operación poco común. Fue utilizado en *Kerberos v4* [2] y *WASTE* [3],

[2] protocolo de autenticación de red basado en "tickets" que permite a los nodos dentro de una red no segura establecer una comunicación para probar su identidad uno a otro de una forma segura

[3] protocolo "punto a punto" con un cifrado fuerte para asegurar que la información transferida, ya sea de mensajería instantánea o compartición de archivos, sea inaccesible a terceras partes

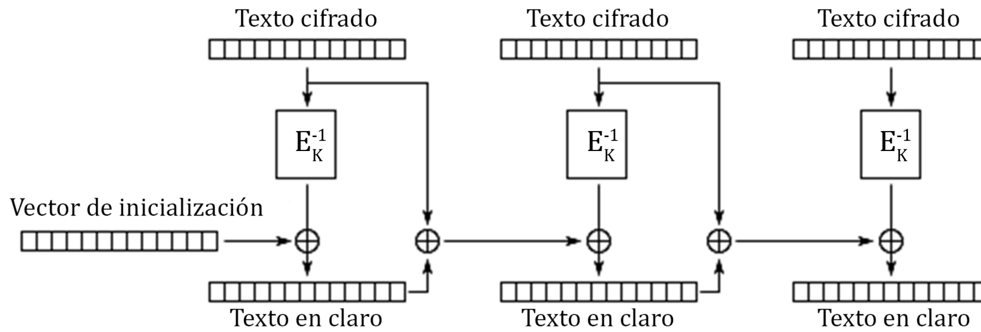


Figura 3.4: PCBC en modo descifrado

pero fue descontinuado debido a que la permutación de dos bloques adyacentes no afecta el proceso de descifrado de los bloques siguientes.

3.4. Modo de Retroalimentación Cifrada

El modo de Retroalimentación Cifrada (CFB, del inglés Cipher Feedback) [?][?] hace que el cifrador por bloques opere como un cifrador por flujo; esto es, si una parte de un bloque cifrado se pierde por algún motivo (i.e. error de transmisión), el receptor perderá solamente parte del mensaje original, considerando esta parte como *contenido ilegible* y sin embargo, podrá continuar con el proceso de descifrado después de procesar una cantidad determinada de bloques. El proceso de cifrado toma el bloque cifrado anterior $i-1$, lo ingresa al cifrador por bloques (modo cifrado) con la llave secreta K y aplica una operación XOR entre el bloque resultante (bloque de flujo) y el bloque en claro i, obteneniendo así el bloque cifrado i. Dado que cada bloque cifrado i depende del bloque cifrado anterior $i-1$, el proceso de cifrado no puede ser ejecutado en paralelo. El proceso de descifrado es básicamente igual al cifrado: se toma el bloque cifrado anterior $i-1$, se ingresa al cifrador por bloques (modo cifrado) con la llave secreta K y se aplica una operación XOR entre el bloque de flujo resultante y el bloque cifrado i para obtener el bloque en claro i. El descifrado de datos si puede ser paralelizado, debido a que un bloque en claro i depende solamente de dos bloques cifrados adyacentes. Al igual que los modos de operación anteriores, para procesar el primer bloque (en claro o cifrado) se utiliza un VI. Los algoritmos de este modo de operación son descritos con las ecuaciones (3.3) y las figuras 3.5 y 3.6 muestran los diagramas por bloque correspondientes.

$$
\begin{aligned}
C_i &= \begin{cases} E_K(VI) \oplus P_i, \text{si } i = 0 \\ E_K(C_{i-1}) \oplus P_i, \text{si } i > 0 \end{cases} \\
P_i &= \begin{cases} E_K(VI) \oplus C_i, \text{si } i = 0 \\ E_K(C_{i-1}) \oplus C_i, \text{si } i > 0 \end{cases}
\end{aligned} \tag{3.3}
$$

Al igual que el modo CBC, todos los cambios realizados en el mensaje original serán propagados en el mensaje cifrado. Al momento de descifrar un bloque, el cambio de un solo

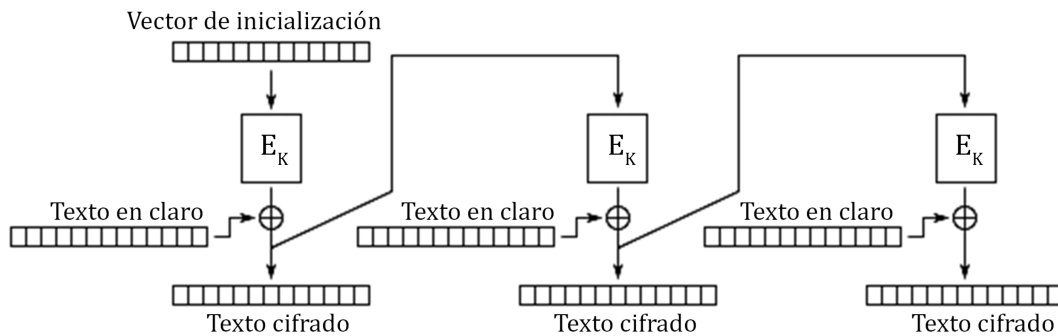


Figura 3.5: CFB en modo cifrado

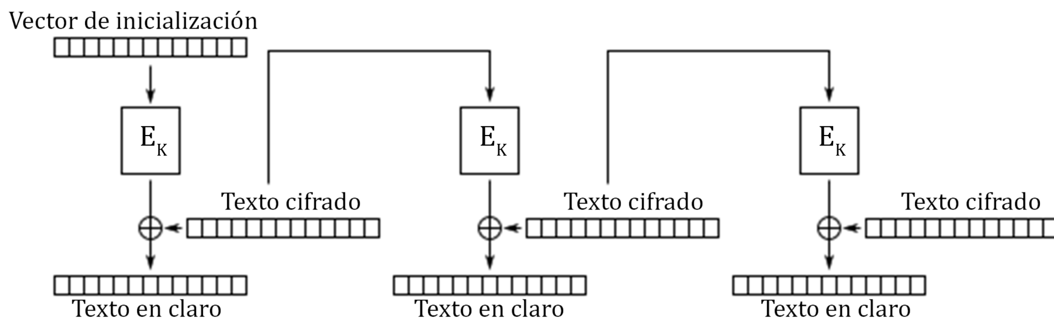


Figura 3.6: CFB en modo descifrado

bit afectará dos bloques en claro: el primer bloque se verá afectado solamente en un bit y el segundo bloque estará completamente corrupto. Una ventaja que tiene CFB sobre CBC es que el último bloque no necesita ser rellenado en caso de estar incompleto, con lo que se puede ahorrar trabajo computacional.

3.5. Modo de Retroalimentación de Salida

El modo de Retroalimentación de Salida (OFB, del inglés Output Feedback) [**?**][**?**] es muy similar al modo CFB, ya que también opera como un cifrador por flujo. El proceso de cifrado genera un bloque de flujo O_i ingresando el bloque de flujo anterior O_{i-1} al cifrador por bloques (modo cifrado) con la llave secreta K. Al tener el bloque de flujo O_i se le aplica una operación XOR con el bloque en claro i para producir el bloque cifrado i. Dado que cada bloque de flujo O_i depende del bloque de flujo anterior O_{i-1}, en algún momento se necesitará generar el primer bloque de flujo O_0; dicho bloque es generado ingresando un VI al cifrador por bloques. Ya que la operación XOR es simétrica, el proceso de descifrado es exáctamnte igual al proceso de cifrado. La dependencia de datos en cadena impide que el modo de operación OFB pueda ser paralelizado, tanto en cifrado como en descifrado de datos. Sin embargo, como los bloques de entrada son utilizados solamente en la operación XOR final, todas las operaciones del cifrador por bloques pueden ser ejecutadas secuencialmente, y el último paso puede ser ejecutado en paralelo cuando todos los bloques de flujo se encuentren

disponibles. Las ecuaciones (3.4) describen los algoritmos de cifrado y descifrado de este modo de operación, y las figuras 3.7 y 3.8 muestran los diagramas por bloque correspondientes.

$$
\begin{aligned}
C_i &= \begin{cases} O_i \oplus P_i, \text{con } O_i = E_K(VI), \text{si } i = 0 \\ E_K(O_i) \oplus P_i, \text{con } O_i = E_K(O_{i-1}), \text{si } i > 0 \end{cases} \\
P_i &= \begin{cases} O_i \oplus C_i, \text{con } O_i = E_K(VI), \text{si } i = 0 \\ E_K(O_i) \oplus C_i, \text{con } O_i = E_K(O_{i-1}), \text{si } i > 0 \end{cases}
\end{aligned}
\tag{3.4}
$$

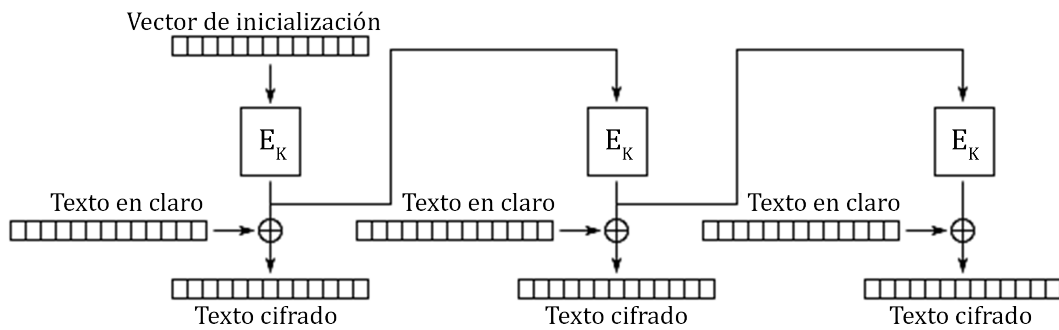


Figura 3.7: OFB en modo cifrado

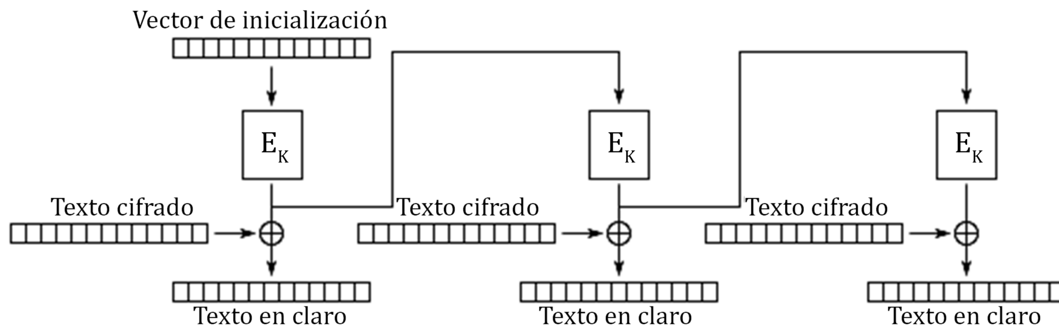


Figura 3.8: OFB en modo descifrado

Al igual que CFB, modificar un solo bit en el mensaje cifrado producirá un mensaje en claro modificado exactamente en la misma ubicación. Esta propiedad permite que muchos códigos de detección de errores funcionen normalmente incluso cuando son aplicados antes del cifrado de datos.

3.6. Modo de Libro de Códigos Electrónico

El modo de Libro de Códigos Electrónico (ECB, del inglés Electronic Codebook) [?][?] es el más simple de todos los modos de operación para cifrado y descifrado de datos. La entrada

es dividida en bloques del mismo tamaño, los cuales son transformados de tal forma que no depende un bloque de otro. El proceso de cifrado ingresa directamente un bloque en claro i al cifrador por bloques (modo cifrado) con la llave secreta K; el resultado obtenido es el bloque cifrado i. El proceso de descifrado ingresa un bloque cifrado i al cifrador por bloques (modo descifrado) con la llave secreta K y el resultado es el bloque en claro i. En este modo de operación no existe ningún tipo de dependencia de datos, por lo que ECB puede ser paralelizado fácilmente, tanto en cifrado como en descifrado de datos. Los procesos de cifrado y descifrado son descritos con las ecuaciones (3.5); las figuras 3.9 y 3.10 ejemplifican los procedimientos recién descritos.

$$\begin{aligned} C_i = E_K(P_i), \forall i \in [0, n-1] \\ P_i = E_K^{-1}(C_i), \forall i \in [0, n-1] \end{aligned} \tag{3.5}$$

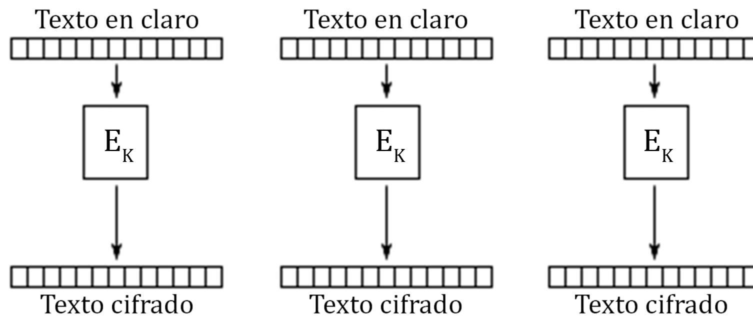


Figura 3.9: ECB en modo cifrado

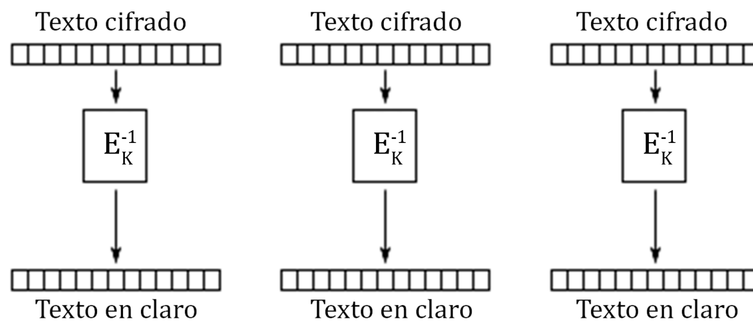


Figura 3.10: ECB en modo descifrado

La desventaja de este modo de operación es que bloques en claro idénticos producirán exáctamente el mismo bloque cifrado, y esto en realidad no oculta debidamente los patrones de la información; en cierta forma, no provee la confidencialidad necesaria para los datos y no es recomendable utilizarlo en protocolos criptográgicos. Un claro ejemplo que muestra cómo ECB puede dejar visibles patrones de la información en los bloques cifrados finales es cifrando un mapa de bits; en este tipo de imágenes suelen existir grandes áreas del mismo

color y al cifrar píxel por píxel, aún podrá ser visualizado el patrón de la imagen original en los píxeles coloreados idénticamente.

3.7. Modo Contador

El modo Contador (CTR) [?][?], al igual que CFB y OFB, opera como un cifrador por flujo. Genera los bloques de flujo ingresando un valor aleatorio N (llamado *nonce*) concatenado con valores sucesivos de un contador ctr_i al cifrador por bloques (modo cifrado) con la llave secreta K. El valor de N puede ser generado aleatoriamente o no. El valor del contador ctr_i puede ser el resultado de cualquier función que garantice producir una secuencia de bits que sea poco probable volver a repetir en mucho tiempo, aunque incrementarlo en uno es el método más simple y más popular actualmente. Si el valor de N también es aleatorio, entonces puede ser combinado con el contador ctr_i utilizando cualquier operación que no provoque pérdidas de datos (i.e. concatenación o XOR) para producir el bloque contador único que será utilizado en los procesos de cifrado y descifrado de datos. Si el valor de N no es aleatorio, entonces lo más recomendable es concatenarlo con el contador ctr_i. Al igual que el modo ECB, la entrada recibida es dividida en bloques del mismo tamaño y posteriormente son procesados de forma que no dependen uno de otro. El cifrado de datos realiza una operación XOR entre un bloque de flujo y un bloque en claro i para obtener el bloque cifrado i. El proceso de descifrado es idéntico al cifrado, nuevamente por la simetría de la operación XOR. El modo de operación puede ser ejecutado en paralelo, pero se debe procurar actualizar correctamente el valor del contador ctr_i para garantizar la confidencialidad provista por el algoritmo. Los procesos de cifrado y descifrado de CTR son descritos con las ecuaciones (3.6), y las figuras 3.11 y 3.12 muestran los diagramas por bloque del modo de operación.

$$\begin{aligned} C_i = E_K(N||ctr_i) \oplus P_i, \forall i \in [0, n-1] \\ P_i = E_K(N||ctr_i) \oplus C_i, \forall i \in [0, n-1] \end{aligned} \tag{3.6}$$

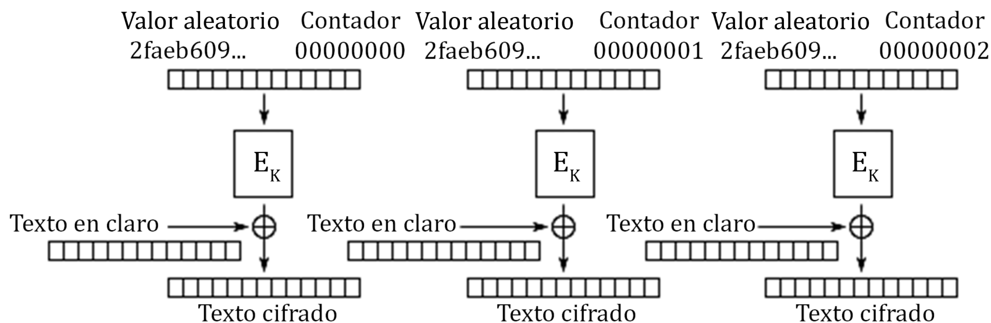


Figura 3.11: CTR en modo cifrado

El uso del contador ha sido debatido a lo largo de los años; expertos aseguran que el hecho de exponer un cripto-sistema dando a conocer parte de la entrada representa un riesgo de seguridad innecesario [?]. Sin embargo, el modo CTR ha sido ampliamente aceptado y los

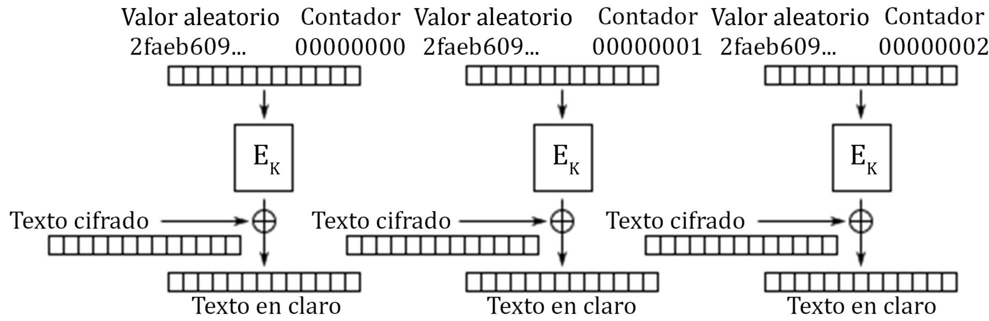


Figura 3.12: CTR en modo descifrado

problemas de seguridad que puedan llegar a existir son considerados debilidades del cifrador por bloques, más no del modo de operación. Otro detalle sobre CTR es que se considera más seguro que ECB debido a que dos bloques de entrada iguales no producen la misma salida; el contador provoca que un bloque de flujo sea siempre diferente de otros y, por ende, los bloques de salida también serán diferentes al aplicarse la operación XOR. Nótese que en este caso, el valor aleatorio N concatenado con el contador ctr_i es equivalente al VI de otros modos de operación.

3.8. Modo Contador de Galois

El modo Contador de Galois (GCM, del inglés Galois Counter Mode) [?] opera como un cifrador por bloques que además de proveer cifrado de datos, también provee la autenticación de los mismos; esta característica es conocida como *cifrado autenticado*. En la actualidad ha surgido la necesidad de crear un modo de operación que pueda proveer cifrado autenticado a altas velocidades, tanto en hardware como en software y que además, sea fácilmente programable en cualquier arquitectura; GCM es capaz de realizar estas tareas. En la transformación de datos, GCM es capaz de alcanzar velocidades similares a CTR debido a que opera de la misma forma, con un valor aleatorio N concatenado con un contador ctr_i. La diferencia con CTR es que el primer valor del contador (i.e. ctr_0) se reserva para generar parte de un bloque especial llamado *etiqueta de autenticación*, el cual servirá para autenticar los datos provistos. Los valores del siguiente contador ctr_{i+1} son utilizados para transformar los bloques de entrada. Las ecuaciones (3.7) describen los procesos de cifrado y descifrado de este modo de operación.

$$\begin{aligned} C_i = E_K(N||ctr_{i+1}) \oplus P_i, \forall i \in [0, n-1] \\ P_i = E_K(N||ctr_{i+1}) \oplus C_i, \forall i \in [0, n-1] \end{aligned} \tag{3.7}$$

Al igual que CTR, las transformaciones de bloques en GCM pueden ser paralelizadas debido a que no existe dependencia de datos entre una transformación y otra. Sin embargo, el rendimiento de GCM se ve decrementado en la generación de la etiqueta de autenticación, la cual es generada con las ecuaciones (3.8).

$$
\begin{aligned}
H &= E_K(0^{128}) \\
ctr_0 &= \begin{cases} VI||0^{31}1, \text{si } longitud(VI) = 96 \\ GHASH(H, \{\}, VI), \text{de otro modo} \end{cases} \\
T &= MSB_t(GHASH(H, A, C) \oplus E_K(ctr_0))
\end{aligned} \tag{3.8}
$$

La función $GHASH$ está definida como:

$$GHASH(H, A, C) = X_{m+n+1} \tag{3.9}$$

donde A representa los datos de autenticación, C representa el texto cifrado y las variables $X_i, i \in [0, m+n+1]$ están definidas por:

$$
X_i = \begin{cases} 0, \text{si } i = 0 \\ (X_{i-1} \oplus A_i) \cdot H, \text{si } i \in [1, m-1] \\ (X_{m-1} \oplus (A_m^*||0^{128-v})) \cdot H, \text{si } i = m \\ (X_{i-1} \oplus C_{i-m}) \cdot H, \text{si } i \in [m+1, m+n-1] \\ (X_{m+n-1} \oplus (C_n^*||0^{128-u})) \cdot H, \text{si } i = m+n \\ (X_{m+n} \oplus (longitud(A)||longitud(C))) \cdot H, \text{si } i = m+n+1 \end{cases} \tag{3.10}
$$

donde:

- A_m^*: último bloque de autenticación
- m: longitud en bits de los datos de autenticación
- C_n^*: último bloque del texto cifrado
- n: longitud en bits del texto en claro
- u: longitud en bits del último bloque cifrado
- v: longitud en bits del último bloque de autenticación

Como se puede observar en las ecuaciones (3.8), (3.9) y (3.10), el costo computacional requerido por la generación de la etiqueta de autenticación es elevado, ya que se necesita de varias multiplicaciones en el campo finito GF(2^{128}) para el cálculo de la función $GHASH$ (excepto si $i = 0$). Realizar estas operaciones puede degradar significativamente el desempeño del equipo de cómputo. Las figuras 3.13 y 3.14 muestran los diagramas por bloque del modo de operación, dejando ver más claramente el funcionamiento del algoritmo.

El cálculo de la etiqueta de autenticación debe ejecutarse forzosamente de una manera secuencial, ya que el resultado de cada multiplicación es adicionado a un bloque cifrado i para después ser multiplicado nuevamente por el bloque H. Es debido a esta dependencia de datos que GCM resulta ser un modo de operación que no es cien por ciento paralelizable y por ende, no se podrá obtener una aceleración satisfactoria en una arquitectura altamente paralela.

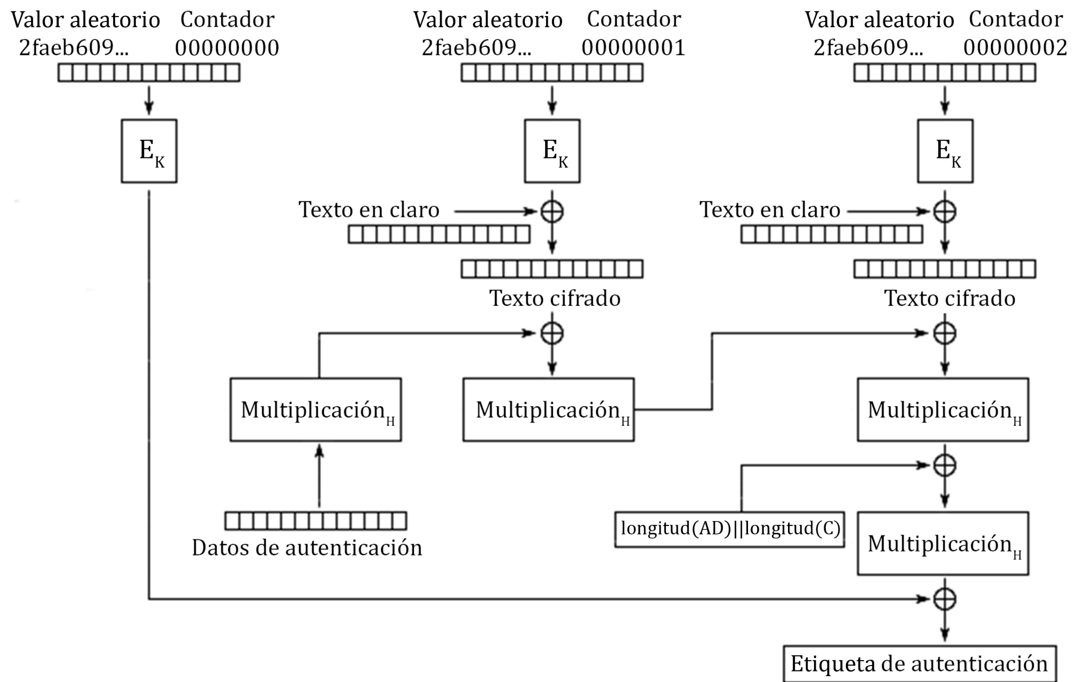


Figura 3.13: GCM en modo cifrado

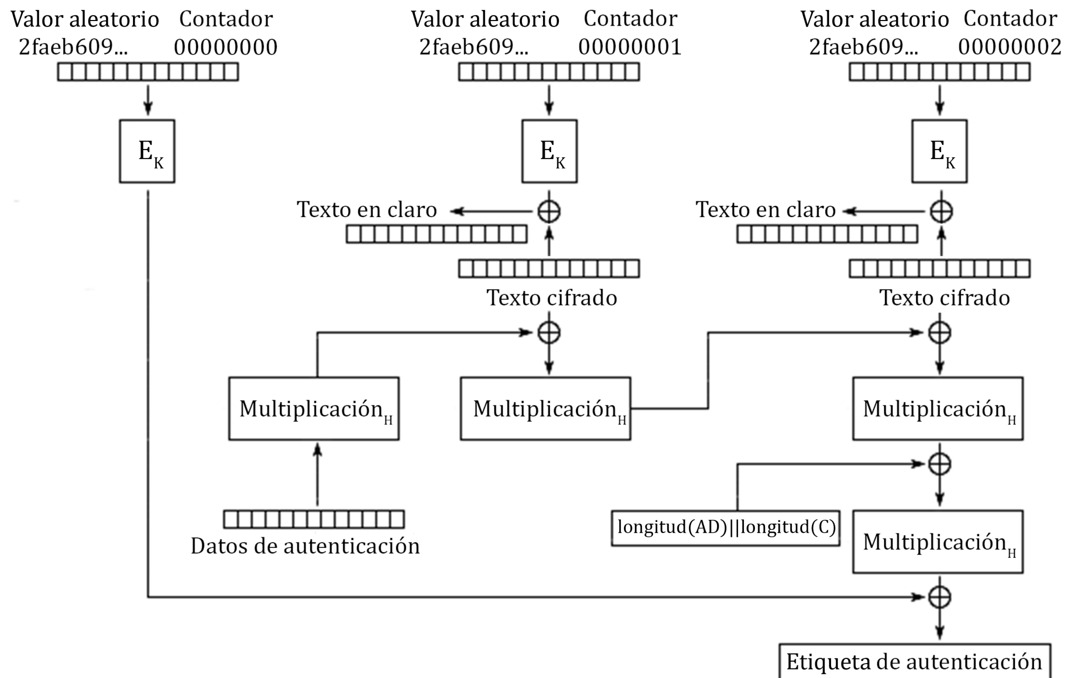


Figura 3.14: GCM en modo descifrado

3.9. Modo de Dos-Rondas de Desplazamiento

El modo de Dos-Rondas de Desplazamiento (OTR) [?] opera como un cifrador por bloques y, así como GCM, también es capaz de proveer cifrado autenticado con datos asociados. Posee una característica muy particular que podría ser considerada como una desventaja: necesita de particiones conformadas por dos bloques de entrada, con lo que se genera una ligera dependencia de datos entre pares de bloques. En los procesos de cifrado y descifrado de datos, OTR ocupa secuencias de bytes llamadas *máscaras* que, como su nombre lo sugiere, son utilizadas para enmascarar bloques antes de ser ingresados al cifrador por bloques. La primer máscara L se obtiene ingresando un valor aleatorio N (con un formato en particular) al cifrador por bloques (modo cifrado) con la llave secreta K. La ecuación (3.11) en conjunto con la figura 3.15 muestran la forma en la que se genera la máscara L inicial. Con esta primer máscara L podrán ser calculados futuros valores de las máscaras a ser utilizadas en las transformaciones de particiones posteriores, simplemente doblando valores de L y realizando operaciones de adición (XOR). Por ejemplo, en la transformación de la primer partición, conformada por el primer y segundo bloque de entrada, se requieren las máscaras L y $3L$. Las máscaras posteriores son calculadas con base en L y $3L$, nuevamente doblando sus valores (ver apéndice A.6.3).

$$L = E_K(\text{formato}(N)) \tag{3.11}$$

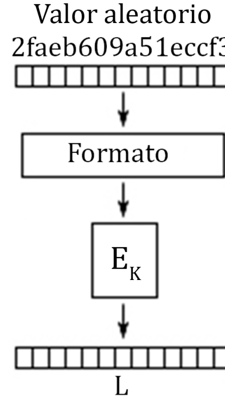


Figura 3.15: Generación de máscara L inicial en el modo OTR

Nótese que por cada partición de bloques se requiere un nuevo doblado de L y $3L$, por lo que es necesario calcular $\frac{m}{2}$ doblados de L y $\frac{m}{2}$ doblados de $3L$, dando un total de m doblados.

Los valores de las máscaras deberán ser calculados a lo largo de la transformación como:

$$\{2^0 L, 2^0 \cdot 3L\}, \{2^1 L, 2^1 \cdot 3L\}, \ldots, \{2^{l-2} L, 2^{l-2} \cdot 3L\}, \{2^{l-1} L, 2^{l-1} \cdot 3L\}$$

donde $l = \lfloor \frac{m}{2} \rfloor$ representa el número total de particiones.

En el proceso de cifrado de datos, para obtener el bloque cifrado i, se aplica una operación XOR entre el bloque en claro i y la máscara L correspondiente a la partición, ingresando

el bloque resultante al cifrador por bloques (modo cifrado) con la llave secreta K y posteriormente, aplicar nuevamente una operación XOR entre el bloque resultante y el bloque en claro $i+1$. Para obtener el bloque cifrado $i+1$, se aplica una operación XOR entre el bloque cifrado i y la máscara $3L$ correspondiente, para después ingresar el resultado al cifrador por bloques (modo cifrado) con la llave secreta K y finalmente, aplicar una última operación XOR entre el bloque resultante y el bloque en claro i. Es muy importante resaltar que el procedimiento recién descrito se aplica a todas las particiones excepto la última, que puede estar conformada por uno o dos bloques, y el último bloque puede ser o no ser completo. En el procedimiento de descifrado se debe permutar la máscara L con la máscara $3L$, lo que implica que el enmascaramiento del bloque cifrado i, antes de ingresar al cifrador por bloques, se realiza con la máscara $3L$, y el enmascaramiento al bloque en claro i se realiza con la máscara L. Las ecuaciones (3.12) describen las funciones de cifrado y descifrado de bloques para particiones conformadas por dos bloques, no incluyendo la última; la figura 3.16 muestra el diagrama correspondiente al algoritmo de cifrado.

$$
\begin{aligned}
C_{2i} &= E_K(P_{2i} \oplus 2^i L) \oplus P_{2i+1}, \forall i \in [0, l-2] \\
C_{2i+1} &= E_K(C_{2i} \oplus 2^i 3L) \oplus P_{2i}, \forall i \in [0, l-2] \\
P_{2i} &= E_K(C_{2i} \oplus 2^i 3L) \oplus C_{2i+1}, \forall i \in [0, l-2] \\
P_{2i+1} &= E_K(P_{2i} \oplus 2^i L) \oplus C_{2i}, \forall i \in [0, l-2]
\end{aligned}
\tag{3.12}
$$

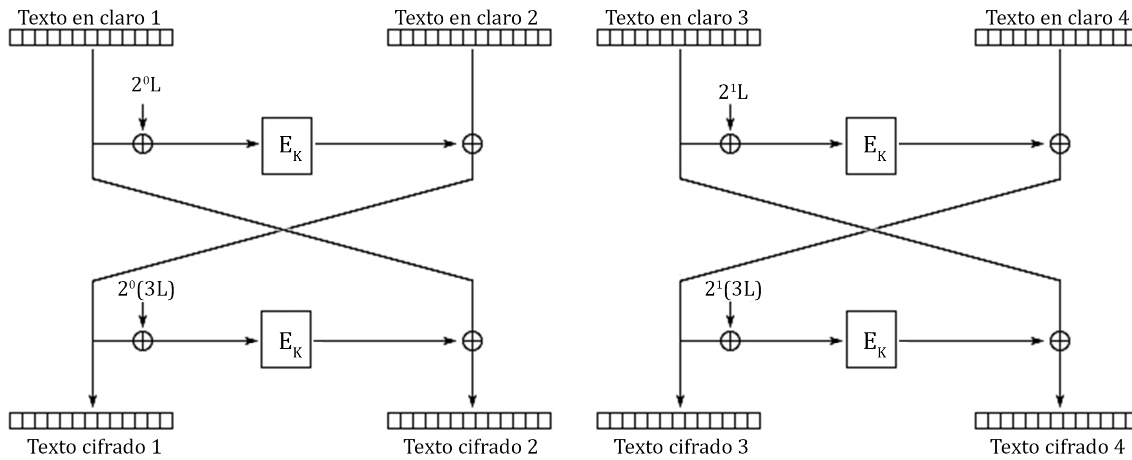


Figura 3.16: OTR en modo cifrado

Como se mencionó anteriormente, las ecuaciones (3.12) no son aplicadas en la última partición; ésta se maneja de forma diferente dependiendo del número de bloques:

- si el número de bloques m es par: para obtener el último bloque cifrado $m-1$, se aplica una operación XOR entre el bloque en claro $m-2$ y la última máscara $2^{l-1}L$, ingresando el bloque resultante al cifrador por bloques (modo cifrado) con la llave secreta K; posteriormente, hay que considerar solamente los t bits más significativos y aplicar nuevamente una operación XOR entre el bloque resultante y el bloque en claro

$m-1$; en el caso del bloque cifrado $m-2$, se aplica una operación XOR entre el bloque cifrado $m-1$ (rellenado si es necesario) y la última máscara $2^{l-1}3L$, para después ingresar el resultado al cifrador por bloques (modo cifrado) con la llave secreta K y finalmente, aplicar una última operación XOR entre el bloque resultante y el bloque en claro $m-2$. Para descifrar, al igual que con las particiones anteriores, solo se permutan las máscaras L y $3L$.

- si el número de bloques m es impar: para obtener el último bloque cifrado $m-1$, se ingresa la última máscara $2^{l-1}L$ directamente al cifrador por bloques (modo cifrado) con la llave secreta K; del bloque resultante se consideran los t bits más significativos y finalmente se aplica una operación XOR con el bloque en claro $m-1$. El proceso de descifrado es idéntico.

Las ecuaciones (3.13) describen las funciones de cifrado y descifrado para la última partición; las figuras 3.17 y 3.18 muestran los diagramas correspondientes para ambos casos del algoritmo.

$$m \text{ par}: \begin{cases} C_{m-1} = MSB_t(E_K(P_{m-2} \oplus 2^{l-1}L)) \oplus P_{m-1} \\ C_{m-2} = E_K(\underline{C_{m-1}} \oplus 2^{l-1}3L) \oplus P_{m-2} \\ P_{m-1} = MSB_t(E_K(C_{m-2} \oplus 2^{l-1}3L)) \oplus C_{m-1} \\ P_{m-2} = E_K(\underline{P_{m-1}} \oplus 2^{l-1}L) \oplus C_{m-2} \end{cases} \tag{3.13}$$

$$m \text{ impar}: \begin{cases} C_{m-1} = MSB_t(E_K(2^{l-1}L)) \oplus P_{m-1} \\ P_{m-1} = MSB_t(E_K(2^{l-1}L)) \oplus C_{m-1} \end{cases}$$

El modo de operación OTR puede ser paralelizado considerando que cada hilo creado debe procesar una partición de dos bloques y además, debe tener una forma de generar o acceder fácilmente a las máscaras correspondientes a la partición por procesar; este es el principal inconveniente en la paralelización: todos los valores de enmascaramiento de datos dependen de L y se calculan con base en el valor anterior. Al encontrar una forma de calcular eficientemente estos valores, la paralelización del algoritmo será más fácil de efectuar. En la sección correspondiente a la implementación se discutirá una forma de optimizar ligeramente el cálculo de las máscaras.

A diferencia de GCM, OTR no requiere de un multiplicador en GF(2^{128}) para proporcionar autenticación de datos; tomando en cuenta que un multiplicador en GF(2^{128}) consume muchos recursos de cómputo, el costo computacional de OTR, comparado con el costo computacional de GCM en la generación de la etiqueta de autenticación se ve bastante reducido, tanto que puede llegar a ser comparado con el costo computacional de otros modos de operación, como por ejemplo CTR. Dado que no es posible obtener un cifrado autenticado con una cantidad menor de llamadas de AES de las que requiere CTR, el costo computacional de OTR es considerado mínimo entre los modos de operación. Sin embargo, el proceso de transformación de datos puede llegar a consumir muchos recursos computacionales a causa de los doblados en las máscaras L y $3L$, por lo que es recomendable optimizar esta operación para evitar perder rendimiento computacional en cifrado y descifrado datos.

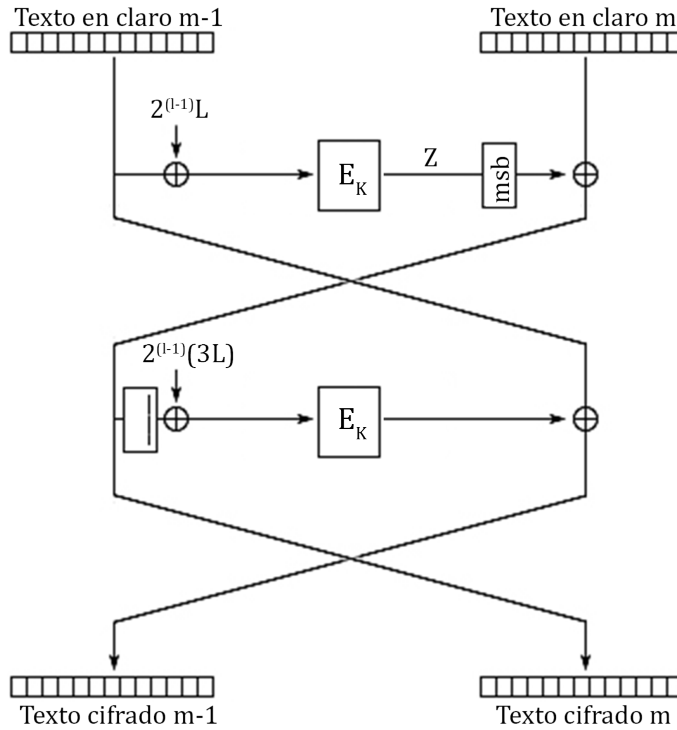


Figura 3.17: Cifrado OTR para la última partición compuesta por un número de bloques par

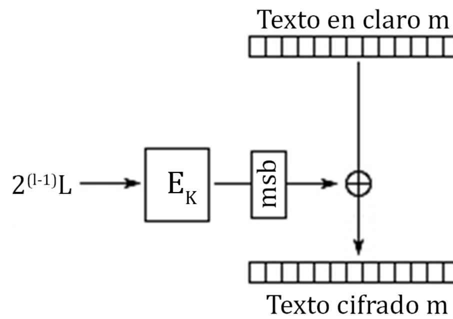


Figura 3.18: Cifrado OTR para la última partición compuesta por un número de bloques impar

En el caso de la autenticación de datos, OTR genera una etiqueta de autenticación T a partir de dos etiquetas de autenticación más:

- TE: requiere calcular dos variables extra, que son Σ y L^*; ambas variables dependen del número de bloques que componen el mensaje en claro. Σ es la suma de todos los bloques en claro pares (en este caso, bloques con índice impar) más el último bloque rellenado si es necesario; es posible calcularla con la ecuación (3.14). L^* es una de las máscaras utilizadas en la transformación de la última partición. La ecuación (3.15) muestra los dos posibles valores para L^*.

$$\Sigma = \begin{cases} \begin{cases} Z = E_K(P_{m-2} \oplus 2^{l-1}L) \\ P_1 \oplus P_3 \oplus \cdots \oplus P_{m-3} \oplus Z \oplus \underline{C_{m-1}} \end{cases} , \text{si m par} \\ P_1 \oplus P_3 \oplus \cdots \oplus P_{m-2} \oplus P_{m-1}, \text{si m impar} \end{cases} \tag{3.14}$$

$$L^* = \begin{cases} 2^{l-1}3L, \text{si m par} \\ 2^{l-1}L, \text{si m impar} \end{cases} \tag{3.15}$$

El valor final de la etiqueta de autenticación TE dependerá del número de bytes que componen el último bloque cifrado. La ecuación (3.16) muestra los dos posibles cálculos de TE.

$$TE = \begin{cases} E_K(\Sigma \oplus 3^2L^*), \text{si } |C_{m-1}| \neq n \\ E_K(\Sigma \oplus 7L^*), \text{si } |C_{m-1}| = n \end{cases} \tag{3.16}$$

En la ecuación (3.16) se puede apreciar que si el último bloque no está completo, el cálculo final de TE será más simple, debido a que sólo se requerirá calcular el valor $3L^*$ y doblarlo. En caso contrario, si el último bloque está completo, se requerirá calcular el valor $3L^*$ y 2^2L^*, para luego sumarlos, lo que implicará más costo computacional.

- TA: esta variable es calculada a partir de datos asociados, los cuales pueden ser proporcionados como entrada o no. En caso de no contar con datos asociados, el valor de TA se asigna directamente a un bloque compuesto por valores cero (bytes nulos). De otra forma, se requerirá calcular una variable Q cifrando un bloque de bytes nulos, ya que será necesario enmascarar los bloques de datos asociados de la misma forma en como son enmascarados los bloques a cifrar o descifrar. El cálculo de Q es descrito con la ecuación (3.17) y a su vez, con la figura 3.19; la ecuación (3.18) muestra los posibles cálculos de TA y la figura 3.20 muestra el diagrama correspondiente al algoritmo.

$$Q = E_K(0^{128}) \tag{3.17}$$

$$TA = \begin{cases} 0^n, \text{si } |A| = 0 \\ E_K(\bigoplus_{i=0}^{a-2} E_K(A_i \oplus 2^iQ) \oplus \underline{A_{a-1}} \oplus 2^{a-1}3Q), \text{si } |A_{a-1}| \neq n \\ E_K(\bigoplus_{i=0}^{a-2} E_K(A_i \oplus 2^iQ) \oplus \underline{A_{a-1}} \oplus 2^{a-1}3^2Q), \text{si } |A_{a-1}| = n \end{cases} \tag{3.18}$$

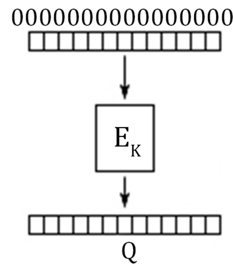


Figura 3.19: Generación de Q inicial en el modo OTR

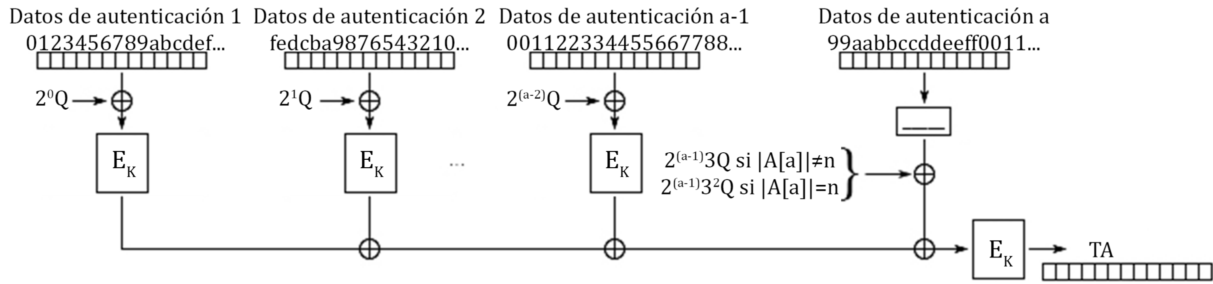


Figura 3.20: Generación de la etiqueta de autenticación TA en el modo OTR

De acuerdo a la ecuación (3.18), se puede llegar a pensar que el cálculo de TA es mucho más complejo que el cálculo de TE; esto puede ser cierto si la cantidad de bloques proporcionados para transformar es similar a la cantidad de datos asociados. Si la cantidad de bloques a cifrar o descifrar es mucho mayor a la de los datos asociados, este cálculo no representará gran pérdida en el rendimiento computacional y, ya que normalmente los datos asociados son una cantidad pequeña de información, el cálculo de esta etiqueta de autenticación TA no afectará seriamente a las aplicaciones que lo implementen.

Contando con estas dos etiquetas, la etiqueta de autenticación T final simplemente podrá ser calculado con la operación XOR entre TE y TA, tal como se muestra en la ecuación (3.19). De la etiqueta de autenticación final se considerarán solamente los t bits más significativos del bloque resultante. La figura 3.21 muestra un diagrama de flujo con el cálculo tanto de TE como de T.

$$T = MSB_t(TA \oplus TE) \tag{3.19}$$

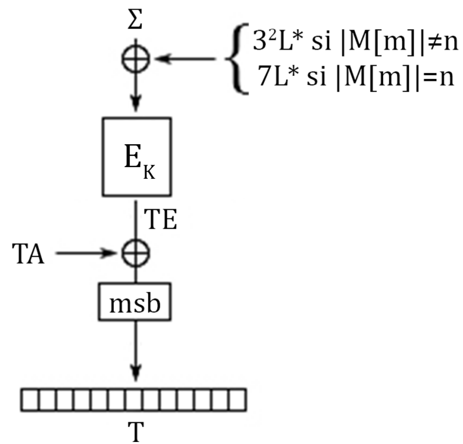


Figura 3.21: Generación de las etiquetas de autenticación TE y T en el modo OTR

3.10. Resumen de características

La tabla 3.1 muestra las características más importantes de los modos de operación recién descritos.

Modo	Cifrado paralelo	Descifrado paralelo	Autenticado de datos	Autenticado paralelo	Rellenado de último bloque
CBC	No	Si	No	-	Si
PCBC	No	No	No	-	Si
CFB	No	Si	No	-	No
OFB	No	No	No	-	No
ECB	Si	Si	No	-	No
CTR	Si	Si	No	-	No
GCM	Si	Si	Si	No	No
OTR	Si	Si	Si	Si	No

Tabla 3.1: Características resumidas de los modos de operación estandarizados por el NIST y del modo de operación en competencia OTR

CAPÍTULO 4

Estándar de Cifrado Avanzado

En enero de 1997, el NIST de los Estados Unidos de América convocó una iniciativa para desarrollar un nuevo estándar de cifrado que sería el reemplazo del Data Encryption Standard (DES) y triple-DES: el AES [?]. En el año 2000, el NIST anunció al algoritmo Rijndael [?] como el ganador debido a la seguridad que es capaz de proporcionar y al hecho de que es fácil de implementar en un amplio rango de plataformas con recursos limitados; posteriormente, en noviembre de 2001 publicó la estandarización oficial de Rijndael, con una modificación, como AES. La modificación que se efectuó sobre Rijndael fue la longitud de bloque y la longitud de la llave. Rijndael es un cifrador por bloques con dos variables: longitud de bloque y longitud de llave, las cuales pueden ser especificadas como un múltiplo de 32 bits, con un mínimo de 128 y un máximo de 256. Es posible definir versiones de Rijndael con longitudes mayores de bloque y llave, pero actualmente no es necesario. En cambio, AES especifica la longitud de bloque a 128 bits y tres posibles longitudes de llave: 128, 192 o 256 bits. Bloques de mayor longitud no fueron evaluados en el proceso de selección de AES y por ende, no fueron adoptados en la estandarización. Al haber sido aceptado por el NIST, el nuevo estándar AES fue enviado a diversas organizaciones y compañias alrededor del mundo para ser adoptado de igual forma que en los Estados Unidos de América.

Ahora bien, AES es un algoritmo criptográfico de llave simétrica que se utiliza para proteger datos electrónicos. Es capaz de cifrar o descifrar información de tal forma que, en el cifrado convierte los datos provistos a una forma ilegible, a lo que se le conoce como *texto cifrado* o *mensaje cifrado*, y en el descifrado convierte los datos cifrados provistos a su forma original, a lo que se le conoce como *texto en claro* o *mensaje en claro*. Como se ha mencionado anteriormente, AES utiliza una longitud de bloque de 128 bits y una longitud de llave secreta de 128, 192 o 256 bits. Con la llave secreta inicial se genera una serie de llaves de ronda por medio de un algoritmo llamado *expansión de llave*; las llaves generadas son utilizadas a lo largo de la transformación. El número de llaves de ronda a generar depende del tamaño de la llave, al igual que el número de rondas. Para cifrar una entrada de bits proporcionada por un usuario (e.g. imagen, audio, video, documentos, etc.) con la llave secreta, lo primero es obtener un bloque de 128 bits y después ejecutar una serie de transformaciones en un número de rondas fijas. Si se cuenta con una llave de 128 bits el número de rondas requeridas para llevar a cabo la transformación será de 10; para una llave de 192 bits las rondas serán 12, y si el tamaño de la llave es 256 se ejecutarán 14 rondas. Cada ronda consiste de la ejecución de cuatro transformaciones especiales sobre un bloque de bits, con excepción de la primer

y última ronda. Dichas transformaciones son: *suma de la llave de ronda* (*AddRoundKey*), *sustitución de bytes* (*SubBytes*), *corrimiento de filas* (*ShiftRows*) y *mezclado de columnas* (*MixColumns*); cada transformación será descrita en secciones posteriores.

4.1. Definiciones básicas

En el algoritmo 1 se presenta el pseudo-código del cifrador AES estandarizado por el NIST.

Algoritmo 1 Cifrador AES

Entrada: byte $entrada[4 \times Nb]$, byte $salida[4 \times Nb]$, palabra $w[Nb \times (Nr + 1)]$
Salida: byte $salida[4 \times Nb]$
1: byte $estado[4, Nb]$
2: $estado \leftarrow entrada$
3: ARK($estado, w[0, Nb - 1]$)
4: **para** $ronda = 1$ incremento en 1 hasta $ronda = Nr - 1$ **hacer**
5: SB($estado$)
6: SR($estado$)
7: MC($estado$)
8: ARK($estado, w[ronda \times Nb, (ronda + 1) \times Nb - 1]$)
9: **fin para**
10: SB($estado$)
11: SR($estado$)
12: ARK($estado, w[Nr \times Nb, (Nr + 1) \times Nb - 1]$)
13: $salida \leftarrow estado$

Se puede apreciar que son utilizadas varias variables y cuatro funciones; cada una de ellas se describe a continuación:

- $estado[4, Nb]$: resultado intermedio en una transformación que puede ser representado como una matriz de $4 \times Nb$ bytes. Por ejemplo, dado un arreglo de bytes como entrada:

$$b = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}\hline b_0 & b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7 & b_8 & b_9 & b_{10} & b_{11} & b_{12} & b_{13} & b_{14} & b_{15} \\ \hline\end{array}$$

  El arreglo puede ser representado como una matriz de 4×4:

$$b = \begin{array}{|c|c|c|c|}\hline b_0 & b_4 & b_8 & b_{12} \\ \hline b_1 & b_5 & b_9 & b_{13} \\ \hline b_2 & b_6 & b_{10} & b_{14} \\ \hline b_3 & b_7 & b_{11} & b_{15} \\ \hline\end{array}$$

  Al iniciar la transformación el arreglo de bytes será modificado; a esta nueva matriz de 4×4 se le llama el *estado*:

$$b = \begin{array}{|c|c|c|c|}\hline b_0 & b_4 & b_8 & b_{12} \\ \hline b_1 & b_5 & b_9 & b_{13} \\ \hline b_2 & b_6 & b_{10} & b_{14} \\ \hline b_3 & b_7 & b_{11} & b_{15} \\ \hline\end{array} \rightarrow \begin{array}{|c|c|c|c|}\hline s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ \hline s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ \hline s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ \hline s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \\ \hline\end{array} = s$$

- Nb: representa el número de columnas (palabras) de 32 bits (4 bytes) que conforman el estado (en este caso, $Nb = 4$).
- Nr: representa el número de rondas que son necesarias para transformar la entrada; Nr depende enteramente del tamaño de la llave y puede tomar los valores 10, 12 o 14.
- $entrada[4 \times Nb]$: arreglo de bytes de tamaño $4 \times Nb$ representando el texto en claro que será transformado a texto cifrado.

$entrada_0$	$entrada_4$	$entrada_8$	$entrada_{12}$
$entrada_1$	$entrada_5$	$entrada_9$	$entrada_{13}$
$entrada_2$	$entrada_6$	$entrada_{10}$	$entrada_{14}$
$entrada_3$	$entrada_7$	$entrada_{11}$	$entrada_{15}$

$\rightarrow$

$s_{0,0}$	$s_{0,1}$	$s_{0,2}$	$s_{0,3}$
$s_{1,0}$	$s_{1,1}$	$s_{1,2}$	$s_{1,3}$
$s_{2,0}$	$s_{2,1}$	$s_{2,2}$	$s_{2,3}$
$s_{3,0}$	$s_{3,1}$	$s_{3,2}$	$s_{3,3}$

- $salida[4 \times Nb]$: arreglo de bytes de tamaño $4 \times Nb$ representando el texto cifrado resultado después de haber transformado el texto en claro.

$s_{0,0}$	$s_{0,1}$	$s_{0,2}$	$s_{0,3}$
$s_{1,0}$	$s_{1,1}$	$s_{1,2}$	$s_{1,3}$
$s_{2,0}$	$s_{2,1}$	$s_{2,2}$	$s_{2,3}$
$s_{3,0}$	$s_{3,1}$	$s_{3,2}$	$s_{3,3}$

$\rightarrow$

$salida_0$	$salida_4$	$salida_8$	$salida_{12}$
$salida_1$	$salida_5$	$salida_9$	$salida_{13}$
$salida_2$	$salida_6$	$salida_{10}$	$salida_{14}$
$salida_3$	$salida_7$	$salida_{11}$	$salida_{15}$

- $w[Nb \times (Nr + 1)]$: arreglo de bytes de tamaño $Nb \times (Nr + 1)$ representando las llaves a utilizar en cada ronda del algoritmo. Para un mayor entendimiento, el arreglo será representado de la siguiente forma:

$k_{0,0}$	$k_{0,4}$	$k_{0,8}$	$k_{0,12}$	$k_{1,0}$	$k_{1,4}$	$k_{1,8}$	$k_{1,12}$	$\ldots$	$k_{i,0}$	$k_{i,4}$	$k_{i,8}$	$k_{i,12}$
$k_{0,1}$	$k_{0,5}$	$k_{0,9}$	$k_{0,13}$	$k_{1,1}$	$k_{1,5}$	$k_{1,9}$	$k_{1,13}$	$\ldots$	$k_{i,1}$	$k_{i,5}$	$k_{i,9}$	$k_{i,13}$
$k_{0,2}$	$k_{0,6}$	$k_{0,10}$	$k_{0,14}$	$k_{1,2}$	$k_{1,6}$	$k_{1,10}$	$k_{1,14}$	$\ldots$	$k_{i,2}$	$k_{i,6}$	$k_{i,10}$	$k_{i,14}$
$k_{0,3}$	$k_{0,7}$	$k_{0,11}$	$k_{0,15}$	$k_{1,3}$	$k_{1,7}$	$k_{1,11}$	$k_{1,15}$	$\ldots$	$k_{i,3}$	$k_{i,7}$	$k_{i,11}$	$k_{i,15}$

con $i \in [0, Nr + 1]$ y la llave $k_{0,j}$ representa la llave inicial.

- $ronda$: contador de número de rondas. $1 \leq ronda \leq Nr - 1$
- ARK: función que realiza la transformación de *suma de la llave de ronda.*
- SB: función que realiza la transformación de *sustitución de bytes.*
- SR: función que realiza la transformación de *corrimiento de filas.*
- MC: función que realiza la transformación de *mezclado de columnas.*

Como se muestra en el algoritmo 1, cada ronda se compone de los mismos pasos, exceptuando la primer ronda en la que se adiciona la llave inicial al bloque de bytes (línea 3), y la última, en la que se omite la transformación de mezclado de columnas (líneas 10 a 12). Al término de las rondas el resultado que se obtiene es un bloque de bits cifrados del mismo tamaño que la entrada. El proceso de descifrado se consigue aplicando las operaciones inversas del cifrado con la misma llave utilizada para cifrar.

Las cuatro transformaciones serán descritas en la siguiente sección.

4.2. Transformaciones

4.2.1. Suma de la llave de ronda

La transformación de suma de la llave de ronda (ARK, de AddRoundKey) realiza una adición en el campo GF(2^8) entre una llave de ronda y el estado, aplicando la ecuación (4.1) sobre cada byte.

$$s'_{r,c} = s_{r,c} \oplus k_{i,j}, \forall\ r, c \in [0,3],\ i \in [0, Nr],\ j \in [0, Nb-1] \tag{4.1}$$

Otra forma de representar la transformación es con una suma matricial como se muestra a continuación:

$$\begin{array}{|c|c|c|c|}\hline s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ \hline s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ \hline s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ \hline s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \\ \hline \end{array} = \begin{array}{|c|c|c|c|}\hline s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ \hline s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ \hline s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ \hline s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \\ \hline \end{array} \oplus \begin{array}{|c|c|c|c|}\hline k_{ronda,0} & k_{ronda,4} & k_{ronda,8} & k_{ronda,12} \\ \hline k_{ronda,1} & k_{ronda,5} & k_{ronda,9} & k_{ronda,13} \\ \hline k_{ronda,2} & k_{ronda,6} & k_{ronda,10} & k_{ronda,14} \\ \hline k_{ronda,3} & k_{ronda,7} & k_{ronda,11} & k_{ronda,15} \\ \hline \end{array}$$

4.2.2. Sustitución de bytes

En la transformación de sustitución de bytes (SB, de SubBytes), como su nombre lo indica, se realiza una sustitución de bytes en el estado, donde la sustitución de cada byte es independiente una de otra. Para ello, se utiliza una tabla especial de sustitución llamada *caja-S* (*S-box*), la cual es considerada invertible y es generada con las siguientes dos transformaciones:

1. Se obtiene el inverso multiplicativo de un valor dado, el cual pertenece al campo finito GF(2^8)

2. Se aplica la transformación

$$b'_i = b_i \oplus b_{(i+4) \mod 8} \oplus b_{(i+5) \mod 8} \oplus b_{(i+6) \mod 8} \oplus b_{(i+7) \mod 8} \oplus c_i \tag{4.2}$$

   donde $0 \le i \le 7$, b_i es el i-ésimo bit del byte b y c_i es el i-ésimo bit del byte c, el cual consta del valor constante {63} (hexadecimal) o {01100011} (binario).

Estas mismas operaciones para generar la caja-S pueden ser visualizadas en forma matricial como se muestra en la ecuación (4.3):

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1&0&0&0&1&1&1&1 \\ 1&1&0&0&0&1&1&1 \\ 1&1&1&0&0&0&1&1 \\ 1&1&1&1&0&0&0&1 \\ 1&1&1&1&1&0&0&0 \\ 0&1&1&1&1&1&0&0 \\ 0&0&1&1&1&1&1&0 \\ 0&0&0&1&1&1&1&1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \tag{4.3}$$

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Tabla 4.1: Valores hexadecimales de la caja-S

Tomando cada elemento del campo y aplicando las operaciones matriciales de la ecuación (4.3) se podrán obtener los valores de la caja-S. En la tabla 4.1 son mostrados todos los valores hexadecimales finales de la caja-S.

Teniendo un estado cualquiera, la transformación de sustitución de bytes utilizará la caja-S para realizar un mapeo entre elementos del campo finito:

$s_{0,0}$	$s_{0,1}$	$s_{0,2}$	$s_{0,3}$
$s_{1,0}$	$s_{1,1}$	$s_{1,2}$	$s_{1,3}$
$s_{2,0}$	$s_{2,1}$	$s_{2,2}$	$s_{2,3}$
$s_{3,0}$	$s_{3,1}$	$s_{3,2}$	$s_{3,3}$

$\rightarrow$ SB(s) $\rightarrow$

$s'_{0,0}$	$s'_{0,1}$	$s'_{0,2}$	$s'_{0,3}$
$s'_{1,0}$	$s'_{1,1}$	$s'_{1,2}$	$s'_{1,3}$
$s'_{2,0}$	$s'_{2,1}$	$s'_{2,2}$	$s'_{2,3}$
$s'_{3,0}$	$s'_{3,1}$	$s'_{3,2}$	$s'_{3,3}$

Por ejemplo, si en el estado se cuenta con el elemento $s_{0,0} = \{6a\}$, la forma adecuada de efectuar la sustitución es: buscar la intersección de la fila 6 con la columna a en la caja-S, tomar el valor contenido en ella ($\{02\}$ en este caso) y realizar la sustitución del elemento $s_{0,0} = \{6a\}$ por el elemento $s'_{0,0} = \{02\}$.

4.2.3. Corrimiento de filas

La transformación de corrimiento de filas (SR, de ShiftRows) rota los bytes de las tres últimas filas del estado cíclicamente dependiendo del número de fila. La primer fila ($r = 0$) no se rota. Más específicamente, la transformación se realiza como:

$$s'_{r,c} = s_{r,c+\text{corrimiento}(r,Nb) \mod Nb} \tag{4.4}$$

donde $0 < r < 4$, $0 \leq c < Nb$ y corrimiento(r, Nb) depende de r (recordar que $Nb = 4$) como sigue:

$$\text{corrimiento}(1,4) = 1;\ \text{corrimiento}(2,4) = 2;\ \text{corrimiento}(3,4) = 3 \tag{4.5}$$

El resultado que se obtiene al aplicar la ecuación (4.4) sobre el estado se ilustra a continuación:

$$\begin{array}{|c|c|c|c|}\hline s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ \hline s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ \hline s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ \hline s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \\ \hline \end{array} \rightarrow \boxed{\text{SR}(s)} \rightarrow \begin{array}{|c|c|c|c|}\hline s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ \hline s_{1,1} & s_{1,2} & s_{1,3} & s_{1,0} \\ \hline s_{2,2} & s_{2,3} & s_{2,0} & s_{2,1} \\ \hline s_{3,3} & s_{3,0} & s_{3,1} & s_{3,2} \\ \hline \end{array}$$

4.2.4. Mezclado de columnas

La transformación de mezclado de columnas (MC, de MixColumns) implica mayor costo computacional. Opera sobre el estado columna por columna, considerando cada una de ellas como un polinomio de cuatro términos en el campo GF(2^8). Cada polinomio es multiplicado módulo $x^4 + 1$ con un polinomio fijo $a(x)$, dado por:

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\} \tag{4.6}$$

La multiplicación de polinomios puede ser representada como una multiplicación de matrices:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}, \text{con } 0 \leq c < Nb \tag{4.7}$$

Como resultado de esta multiplicación, los bytes de cada columna en el estado son reemplazados por medio de las siguientes cuatro operaciones:

$$\begin{aligned} s'_{0,c} &= (\{02\} \cdot s_{0,c}) \oplus (\{03\} \cdot s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\ s'_{1,c} &= s_{0,c} \oplus (\{02\} \cdot s_{1,c}) \oplus (\{03\} \cdot s_{2,c}) \oplus s_{3,c} \\ s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \cdot s_{2,c}) \oplus (\{03\} \cdot s_{3,c}) \\ s'_{3,c} &= (\{03\} \cdot s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \cdot s_{3,c}) \end{aligned}$$

4.3. Transformaciones inversas

Las transformaciones de sustitución de bytes inversa (*InvSubBytes*), de corrimiento de filas inverso (*InvShiftRows*) y de mezclado de columnas inverso (*InvMixColumns*) son las inversas de las transformaciones de AES. Pueden ser implementadas en reversa para producir el algoritmo de descifrado.

Los algoritmos estudiados en esta tesis no requirieron de estas transformaciones inversas, por lo que se refiere al lector a [?] y [?] para una descripción detallada de ellas.

4.4. Expansión de llave

Como se ha mencionado anteriormente, el algoritmo de expansión de llave es una rutina que genera una serie de llaves de ronda para ser utilizadas en el proceso de cifrado y/o descifrado de datos en AES. Inicialmente, el algoritmo toma la llave k_0 para generar Nb columnas de la llave k_1; al finalizar, toma la llave k_1 para generar Nb columnas de la llave k_2 y así sucesivamente hasta generar Nr llaves con Nb columnas cada una. En el algoritmo 2 se presenta el pseudo-código para llevar a cabo la expansión de llave en AES.

Algoritmo 2 Expansión de llave

Entrada: byte $llave[4 \times Nk]$, palabra $w[Nb \times (Nr+1)]$, Nk
Salida: palabra $w[Nb \times (Nr+1)]$

1: palabra $temp$
2: $i \leftarrow 0$
3: **mientras** $i < Nk$ **hacer**
4: $w[i] \leftarrow palabra(llave[4 \times i], llave[4 \times i+1], llave[4 \times i+2], llave[4 \times i+3])$
5: $i \leftarrow i+1$
6: **fin mientras**
7: $i \leftarrow Nk$
8: **mientras** $i < Nb \times (Nr+1)$ **hacer**
9: $temp \leftarrow w[i-1]$
10: **si** $i \mod Nk = 0$ **entonces**
11: $temp \leftarrow \mathrm{RW}(temp)$
12: $temp \leftarrow \mathrm{SW}(temp) \oplus Rcon[i/Nk]$
13: **si no si** $Nk > 6 \wedge i \mod Nk = 4$ **entonces**
14: $temp \leftarrow \mathrm{SW}(temp)$
15: **fin si**
16: $w[i] \leftarrow w[i-Nk] \oplus temp$
17: $i \leftarrow i+1$
18: **fin mientras**

En la línea 11 se presenta la transformación de rotación de palabra (RW, de RotWord). Esta transformación toma una palabra de 4 bytes $w = [w_0, w_1, w_2, w_3]$ como entrada, ejecuta una permutación cíclica y regresa la palabra $w' = [w_1, w_2, w_3, w_0]$.

En la línea 12 se muestra la transformación de sustitución de palabra (SW, de SubWord) y un arreglo especial de bytes llamado *Rcon*. La transformación SW toma una palabra $w = [w_0, w_1, w_2, w_3]$ como entrada y, como en la transformación de sustitución de bytes, sustituye cada byte en la palabra w con ayuda de la caja-S, para producir la palabra de salida w'. El arreglo *Rcon* es un arreglo de palabras constante, en la que están contenidos los valores dados por la palabra $\{x^{i-1}, 00, 00, 00\}$, con $x = 02$, $1 \leq i \leq Nr$, donde x^{i-1} representa una potencia de x dentro del campo finito $\mathrm{GF}(2^8)$.

El arreglo puede ser representado de la siguiente manera:

Rcon=

01	02	04	...	x^{i-1}
00	00	00	...	00
00	00	00	...	00
00	00	00	...	00

Por otro lado, la condición mostrada en la línea 10 indica que solamente la primer palabra de cada nueva llave será generada aplicando las transformaciones SW, RW, una operación XOR con la palabra correspondiente del arreglo *Rcon* y una operación XOR más con la primer palabra de la llave de ronda anterior. La generación de la primer palabra de una llave i es ejemplificada de la siguiente forma:

$$\begin{bmatrix} k_{i,0} \\ k_{i,1} \\ k_{i,2} \\ k_{i,3} \end{bmatrix} = \begin{bmatrix} k_{i-1,12} \\ k_{i-1,13} \\ k_{i-1,14} \\ k_{i-1,15} \end{bmatrix} \rightarrow \boxed{\text{RW()}} \rightarrow \begin{bmatrix} k_{i-1,13} \\ k_{i-1,14} \\ k_{i-1,15} \\ k_{i-1,12} \end{bmatrix} \rightarrow \boxed{\text{SW()}} \rightarrow \begin{bmatrix} k'_{i-1,13} \\ k'_{i-1,14} \\ k'_{i-1,15} \\ k'_{i-1,12} \end{bmatrix} \oplus \begin{bmatrix} k_{i-1,0} \\ k_{i-1,1} \\ k_{i-1,2} \\ k_{i-1,3} \end{bmatrix} \oplus \begin{bmatrix} Rcon_{i-1,0} \\ Rcon_{i-1,1} \\ Rcon_{i-1,2} \\ Rcon_{i-1,3} \end{bmatrix}$$

con $i \in [1, Nr]$. En caso de que la palabra a generar no sea la primera, su cálculo será más sencillo; simplemente se debe realizar una operación XOR entre la palabra anterior de la llave i y la palabra equivalente de la llave anterior $i - 1$. Más formalmente, el cálculo puede ser representado de la siguiente forma:

$$\begin{bmatrix} k_{i,j} \\ k_{i,j+1} \\ k_{i,j+2} \\ k_{i,j+3} \end{bmatrix} = \begin{bmatrix} k_{i,j-4} \\ k_{i,j-3} \\ k_{i,j-2} \\ k_{i,j-1} \end{bmatrix} \oplus \begin{bmatrix} k_{i-1,j} \\ k_{i-1,j+1} \\ k_{i-1,j+2} \\ k_{i-1,j+3} \end{bmatrix}$$

con $i \in [1, Nr]$ y $j \in [Nb, 12]$.

4.5. Optimización con cajas-T

En la optimización cajas-T, dos transformaciones de AES, sustitución de bytes y mezclado de columnas, son combinadas dentro de cuatro tablas que consisten de 256 palabras de 32 bits cada una. Para obtenerlas, los valores contenidos en la caja-S son multiplicados por valores constantes dentro del campo GF(2^8) como se muestra a continuación:

$$T_0[a_{i,j}] = \begin{bmatrix} S[a_{i,j}] \cdot 02 \\ S[a_{i,j}] \\ S[a_{i,j}] \\ S[a_{i,j}] \cdot 03 \end{bmatrix} \quad T_1[a_{i,j}] = \begin{bmatrix} S[a_{i,j}] \cdot 03 \\ S[a_{i,j}] \cdot 02 \\ S[a_{i,j}] \\ S[a_{i,j}] \end{bmatrix}$$

$$T_2[a_{i,j}] = \begin{bmatrix} S[a_{i,j}] \\ S[a_{i,j}] \cdot 03 \\ S[a_{i,j}] \cdot 02 \\ S[a_{i,j}] \end{bmatrix} \quad T_3[a_{i,j}] = \begin{bmatrix} S[a_{i,j}] \\ S[a_{i,j}] \\ S[a_{i,j}] \cdot 03 \\ S[a_{i,j}] \cdot 02 \end{bmatrix}$$

De esta forma, al pre-calcular y almacenar las tablas T_0, T_1, T_2 y T_3 el cálculo de una columna del estado se reduce a la siguiente operación:

$$e_j = T_0[a_{0,j}] \oplus T_1[a_{1,j-1}] \oplus T_2[a_{2,j-2}] \oplus T_3[a_{3,j-3}] \oplus k_j \quad (4.8)$$

donde e_j representa la columna a ser calculada y k_j representa la llave de ronda j a ser adicionada.

CAPÍTULO 5

Implementaciones clásicas

Como se ha mencionado anteriormente, antes de proceder al diseño de los algoritmos paralelos, primero hay que realizar la implementación de los algoritmos AES-CTR y AES-OTR secuenciales, que son usualmente los algoritmos codificados en sistemas de cómputo. Con las implementaciones secuenciales se pueden establecer los tiempos de referencia base que servirán para efectuar las comparaciones correspondientes entre algoritmos; además, se puede tener una mejor idea de cómo realizar los diseños paralelos.

En este capítulo se describen las estructuras que fueron diseñadas especialmente para trabajar con AES. También se discute la forma en que se implementan las versiones clásicas de los algoritmos, mostrando los pseudo-códigos y explicándolos en las partes consideradas más importantes.

5.1. Variables constantes y estructuras

Como se mencionó en el capítulo anterior, AES trabaja con los arreglos de bytes entrada, salida y estado, en los que se almacenan los mensajes a transformar (algoritmo 1). Tambien trabaja con el arreglo w en el que se almacenan la llave inicial y las llaves de ronda a utilizar en el proceso de transformación, además de las variables Nb, Nr y $ronda$. Todas estas variables son definidas de la siguiente forma:

i. Nb: valor constante asignado a 16, ya que en el caso de esta tesis, los tamaños de todos los mensajes a procesar son potencia de 2; por lo tanto, cada bloque AES leído contendrá 128 bits (o 16 bytes).

ii. Nr: valor constante asignado a 10, ya que la versión de AES con la que se trabajó es AES-128.

iii. $ronda$: variable para contar el número de ronda en que se trabaja.

Para codificar más fácilmente los arreglos de bytes, se creó la estructura *bloque*, que contiene un arreglo de cuatro columnas (palabra) de bytes, y cada columna está compuesta por cuatro bytes, por lo que se tiene como resultado 16 bytes contenidos por bloque. La estructura puede ser visualizada de la siguiente forma:

$b =$

w_0	w_1	w_2	w_3
b_0	b_4	b_8	b_{12}
b_1	b_5	b_9	b_{13}
b_2	b_6	b_{10}	b_{14}
b_3	b_7	b_{11}	b_{15}

donde cada w_i representa la palabra i en el bloque.

Con ayuda de esta estructura se mejora el almacenamiento tanto de los mensajes en claro o cifrados como de las llaves. Si se cuenta con un mensaje cuyo tamaño es m, con $m > 1$, será necesario almacenarlo en m estructuras *bloque*. Por ejemplo, es posible almacenar la llave inicial en un *bloque*, ya que cuenta con 16 bytes de longitud. El arreglo de llaves también se puede almacenar sobre estructuras *bloque*, pero a diferencia de la llave inicial, se necesita un arreglo de *bloque* de tamaño Nr para conseguirlo; esto es $16 \times Nr$ bytes. La llave inicial y las llaves de ronda se representan en *bloque* como se muestra a continuación:

$k_{ini} =$

w_0	w_1	w_2	w_3
$k_{ini,0}$	$k_{ini,4}$	$k_{ini,8}$	$k_{ini,12}$
$k_{ini,1}$	$k_{ini,5}$	$k_{ini,9}$	$k_{ini,13}$
$k_{ini,2}$	$k_{ini,6}$	$k_{ini,10}$	$k_{ini,14}$
$k_{ini,3}$	$k_{ini,7}$	$k_{ini,11}$	$k_{ini,15}$

$k_r[Nr] =$

w_0	w_1	w_2	w_3	$\dots$	w_0	w_1	w_2	w_3
$k_{1,0}$	$k_{1,4}$	$k_{1,8}$	$k_{1,12}$	$\dots$	$k_{Nr,0}$	$k_{Nr,4}$	$k_{Nr,8}$	$k_{Nr,12}$
$k_{1,1}$	$k_{1,5}$	$k_{1,9}$	$k_{1,13}$	$\dots$	$k_{Nr,1}$	$k_{Nr,5}$	$k_{Nr,9}$	$k_{Nr,13}$
$k_{1,2}$	$k_{1,6}$	$k_{1,10}$	$k_{1,14}$	$\dots$	$k_{Nr,2}$	$k_{Nr,6}$	$k_{Nr,10}$	$k_{Nr,14}$
$k_{1,3}$	$k_{1,7}$	$k_{1,11}$	$k_{1,15}$	$\dots$	$k_{Nr,3}$	$k_{Nr,7}$	$k_{Nr,11}$	$k_{Nr,15}$

Además de los mensajes y llaves, hay otros arreglos de bytes que se considera conveniente almacenarlos en un *bloque*; por ejemplo el VI para AES-CTR, y los datos de autenticación o máscaras L y Q para AES-OTR. Cualquier variable en los algoritmos cuyo tamaño sea 16 bytes podrá representarse con una estructura *bloque*.

También se crearon los arreglos de bytes para almacenar los valores constantes de las cuatro tablas cajas-T, representadas como $T_0[w_i]$, $T_1[w_i]$, $T_2[w_i]$ y $T_3[w_i]$, donde w_i representa el byte i de una palabra w de algún *bloque*. Cada tabla consta de 256 palabras. En total se requieren 1024 bytes por tabla; por lo tanto el almacenamiento de las cuatro tablas se efectúa en 4096 bytes o 4 KB, lo cual no representa una gran cantidad de espacio para la arquitectura en que se trabaja.

5.2. Cifrador AES optimizado con cajas-T

Al trabajar solamente con la versión AES de 128 bits se omitieron algunos pasos del algoritmo, con lo que se aceleró el cifrado de datos. También se modificó el algoritmo para utilizar las cajas-T descritas previamente y la estructura *bloque*. Con esta optimización es posible ejecutar las cuatro transformaciones de AES sobre un bloque de datos palabra por palabra al mismo tiempo, como se especificó en la ecuación (4.8). En el algoritmo 3 se muestra

Algoritmo 3 Cifrador AES cajas-T

Entrada: bloque b, bloque k_{ini}, *bloque* $k_r[Nr]$
Salida: bloque b

1: **palabra** $w_0 \leftarrow b.w_0 \oplus k_{ini}.w_0$
2: **palabra** $w_1 \leftarrow b.w_1 \oplus k_{ini}.w_1$
3: **palabra** $w_2 \leftarrow b.w_2 \oplus k_{ini}.w_2$
4: **palabra** $w_3 \leftarrow b.w_3 \oplus k_{ini}.w_3$
5: **para** $ronda = 0$ incremento en 1 hasta $ronda = Nr - 1$ **hacer**
6: $b.w_0 \leftarrow T_0[w_{0,3}] \oplus T_1[w_{1,2}] \oplus T_2[w_{2,1}] \oplus T_3[w_{3,0}] \oplus k_r[i].w_0$
7: $b.w_1 \leftarrow T_0[w_{1,3}] \oplus T_1[w_{2,2}] \oplus T_2[w_{3,1}] \oplus T_3[w_{0,0}] \oplus k_r[i].w_1$
8: $b.w_2 \leftarrow T_0[w_{2,3}] \oplus T_1[w_{3,2}] \oplus T_2[w_{0,1}] \oplus T_3[w_{1,0}] \oplus k_r[i].w_2$
9: $b.w_3 \leftarrow T_0[w_{3,3}] \oplus T_1[w_{0,2}] \oplus T_2[w_{1,1}] \oplus T_3[w_{2,0}] \oplus k_r[i].w_3$
10: $w_0 \leftarrow b.w_0$
11: $w_1 \leftarrow b.w_1$
12: $w_2 \leftarrow b.w_2$
13: $w_3 \leftarrow b.w_3$
14: **fin para**
15: $b.w_0 \leftarrow T_2[w_{0,3}] \oplus T_3[w_{1,2}] \oplus T_0[w_{2,1}] \oplus T_1[w_{3,0}] \oplus k_r[Nr].w_0$
16: $b.w_1 \leftarrow T_2[w_{1,3}] \oplus T_3[w_{2,2}] \oplus T_0[w_{3,1}] \oplus T_1[w_{0,0}] \oplus k_r[Nr].w_1$
17: $b.w_2 \leftarrow T_2[w_{2,3}] \oplus T_3[w_{3,2}] \oplus T_0[w_{0,1}] \oplus T_1[w_{1,0}] \oplus k_r[Nr].w_2$
18: $b.w_3 \leftarrow T_2[w_{3,3}] \oplus T_3[w_{0,2}] \oplus T_0[w_{1,1}] \oplus T_1[w_{2,0}] \oplus k_r[Nr].w_3$

el pseudo-código del cifrador AES-128 optimizado con cajas-T y con estructuras de datos *bloque*.

En las líneas 1-4 se adiciona la llave de ronda inicial al bloque; esto es la transformación de suma de la llave de ronda inicial; en las líneas 6-9 las cuatro transformaciones se llevan a cabo: la sustitución de bytes y el mezclado de columnas van implícitos en las cajas-T, el corrimiento de filas se genera en las líneas 7-9, rotando las palabras de acuerdo a la especificación de la transformación, y la suma de la llave de ronda, al igual que la ronda inicial, se calcula con una operación XOR con las palabras de la llave de ronda. Como se mostró en el algoritmo 1, en la última ronda se omite la transformación de mezclado de columnas, pero ésta ya fue calculada al generar las cajas-T; para omitirla nuevamente lo que se hace es rotar dos veces a la derecha las cajas-T en el cálculo de la última ronda. El resultado de esta rotación se muestra en las líneas 15-18. Al término del algoritmo 3 el bloque resultante será el bloque cifrado.

Normalmente, cuando se implementa AES se debe codificar tanto la función de cifrado como la de descifrado. En este caso, los modos de operación que se trabajaron necesitan solamente la función de cifrado para realizan ambas transformaciones. Por esta razón no se muestra la función de descifrado de AES-128, pero en las siguientes secciones sí son mostradas tanto las funciones de cifrado como de descifrado de datos.

5.3. Expansión de llave

Al igual que el algoritmo de cifrado, el algoritmo de expansión de llave también se optimizó con las cajas-T y con estructuras *bloque*. Por conveniencia de la implementación se creó la función *expandir*, la cual se encarga de generar una llave j a partir de una llave i en una ronda determinada. Dicha función se muestra en el algoritmo 4.

Algoritmo 4 Expandir k_j a partir de k_i

Entrada: bloque k_i, bloque k_j, entero *ronda*
Salida: bloque k_j
1: **palabra** $w \leftarrow k_i.w_3$
2: $k_j.w_0 \leftarrow k_i.w_0 \oplus T_2[w_2] \oplus T_3[w_1] \oplus T_0[w_0] \oplus T_1[w_3] \oplus Rcon_{ronda}$
3: $k_j.w_1 \leftarrow k_i.w_1 \oplus k_j.w_0$
4: $k_j.w_2 \leftarrow k_i.w_2 \oplus k_j.w_1$
5: $k_j.w_3 \leftarrow k_i.w_3 \oplus k_j.w_2$

La línea 2 del algoritmo indica cómo se calcula la primer palabra de la nueva llave, aplicando la transformación de sustitución de palabra con ayuda de las cajas-T y la transformación de rotación de palabra, rotando los bytes de la tercer palabra de la llave i, para finalmente adicionar la palabra correspondiente del arreglo *Rcon*; en las líneas 3-5 se calculan las palabras restantes.

Teniendo esta función, el algoritmo 2 de expansión de llaves se reduce significativamente. El pseudo-código en el algoritmo 5 ejemplifica la expansión de llaves final. En la línea 1, la llave $k_r[0]$ correspondiente a la ronda 0, se calcula con la llave inicial k_{ini}. El ciclo de la línea 2 muestra cómo generar las llaves de ronda restantes, expandiendo la llave $k_r[i]$ a partir de la llave $k_r[i-1]$.

Algoritmo 5 Expansión de llave optimizado

Entrada: bloque k_{ini}, bloque $k_r[Nr]$, entero Nr
Salida: bloque $k_r[Nr]$
1: EXPANDIR$(k_{ini}, k_r[0], 0)$
2: **para** $i = 1$ incremento en 1 hasta $i = Nr$ **hacer**
3: EXPANDIR$(k_r[i-1], k_r[i], i)$
4: **fin para**

5.4. AES-CTR secuencial

Las transformaciones secuenciales de AES-CTR son simples de codificar, ya que existen bibliotecas estandarizadas por OpenSSL en lenguaje C disponibles para todo público. En esta tesis solo se han ocupado dos de ellas: *openssl/aes.h* y *openssl/rand.h*. Entre las funciones definidas en estas bibliotecas se encuentra una función llamada *AES_ctr128_encrypt*, con la que es posible efectuar el cifrado de un bloque de datos con el algoritmo AES-CTR descrito en la ecuación (3.6), en la página 28. La función recibe siete argumentos: el arreglo de bloques a cifrar *entrada*$[m]$, el arreglo destino *entrada*$[m]$, el número de bytes por bloque *Nb*, la

llave inicial k_{ini}, el vector de inicialización iv, la variable *contador* y un identificador *byte* que sirve para saber qué número de byte dentro del bloque está en proceso de modificación. Además, las variables se actualizan dentro de la función, por lo que no es necesario actualizar constantemente su valor. En el algoritmo 6 se muestra la forma en que se implementó el cifrado AES-CTR con ayuda de la función descrita y en el algoritmo 7 se muestra el descifrado AES-CTR.

Algoritmo 6 Cifrador AES-CTR OpenSSL

Entrada: bloque $entrada[m]$, bloque $salida[m]$, bloque k_{ini}
Salida: bloque $salida[m]$
1: **entero** $contador \leftarrow 0$
2: **entero** $byte \leftarrow 0$
3: **bloque** $vi \leftarrow$ bytes_aleatorios()
4: **para** $i = 0$ incremento en 1 hasta $i = m - 1$ **hacer**
5: AES_CTR128_ENCRYPT($entrada[i], salida[i], Nb, k_{ini}, vi, contador, byte$)
6: **fin para**

Algoritmo 7 Descifrador AES-CTR OpenSSL

Entrada: bloque $entrada[m]$, bloque $salida[m]$, bloque k_{ini}, bloque iv
Salida: bloque $salida[m]$
1: **entero** $contador \leftarrow 0$
2: **entero** $byte \leftarrow 0$
3: **para** $i = 0$ incremento en 1 hasta $i = m - 1$ **hacer**
4: AES_CTR128_ENCRYPT($entrada[i], salida[i], Nb, k_{ini}, vi, contador, byte$)
5: **fin para**

Al revisar ambos algoritmos se puede ver que son muy similares. La diferencia radica en los siguientes detalles:

1. En el cifrador se genera el VI pseudo-aleatoriamente con la función *bytes_aleatorios()*, que puede ser cualquier función que devuelva una cadena de bytes pseudo-aleatorios. Al generarlo, éste es enviado en el mensaje cifrado resultante; en el descifrador se lee el VI recibido.

2. En el cifrador el arreglo $entrada[m]$ contiene el texto en claro y el arreglo $salida[m]$ contendrá el mensaje cifrado; en el descifrador el arreglo $entrada[m]$ contiene el texto cifrado y el arreglo $salida[m]$ contendrá el mensaje en claro.

Nótese que la función *AES_ctr128_encrypt* no recibe como argumento las llaves de ronda, porque dentro de la función se ejecuta la expansión de llave; tampoco acepta argumentos de tipo *bloque*, por lo que se tuvo que adaptar la implementación para trabajar con los tipos de dato adecuados.

5.5. AES-OTR secuencial

Al comenzar el desarrollo del algoritmo se notó que almacenar los arreglos $entrada[m]$, $salida[m]$ y $estado[m]$ es muy costoso en memoria para mensajes de gran tamaño, por lo que se optó por omitir los arreglos $salida[m]$ y $estado[m]$, copiar el par de bloques a procesar en dos bloques extra B_1 y B_2 y realizar la transformación directamente sobre el arreglo $entrada[m]$. Además del arreglo $entrada[m]$, se ocupa uno extra: $A[a]$; este arreglo sirve para almacenar los a datos de autenticación que se usan para generar la etiqueta de autenticación final. El algoritmo 8 ejemplifica claramente los pasos a seguir para llevar a cabo el cifrado y autenticado de datos en AES-OTR y en el algoritmo 9 se muestra el descifrado y validación de la etiqueta.

Las diferencias entre ambos algoritmos son:

1. En el cifrador se genera el valor aleatorio N y la etiqueta de autenticación T, los cuales se envían al destino en el mensaje cifrado resultante; en el descifrador se lee el valor aleatorio N y la etiqueta T; dentro del algoritmo de descifrado se calcula la etiqueta T' para comparar ambas etiquetas T y T'; si son distintas significa que el mensaje ha sido corrompido y por lo tanto no se entrega el mensaje en claro.

2. En el cifrador, la suma de bloques pares se realiza antes de transformar el mensaje en claro contenido en el arreglo $entrada[m]$; en el descifrador es lo inverso, la suma se calcula después de realizar la transformación del mensaje cifrado del arreglo $entrada[m]$.

Además, se puede apreciar que AES-OTR es más complejo que AES-CTR debido al enmascaramiento de datos previo al cifrado, el cual necesita de operaciones de doblado sobre la máscara L, y a la generación de la etiqueta de autenticación T, que requiere del cálculo de TE y TA. Para obtener estas variables son necesarias operaciones extra que pueden reducir el rendimiento del algoritmo. En el caso de TE, las operaciones de adición en GF(2^{128}) de los bloques en claro pares en mensajes de gran tamaño pueden reducir el desempeño computacional, además de que se requiere de un cifrado extra sobre la suma total enmascarada con un doblado de L. En contraste, el cálculo de TA resulta más sencillo, ya que normalmente el tamaño de los datos asociados que se proveen es muy pequeño. Sin embargo, también puede requerir de operaciones de cifrado extra sobre una suma en GF(2^{128}) de los datos de autenticación enmascarados con un doblado de Q. Para una explicación más clara, en el algoritmo 10 se muestra el pseudo-código con las operaciones necesarias para calcular la etiqueta de los datos asociados TA en AES-OTR.

Finalmente, para el almacenamiento de todas estas variables auxiliares se ocupan estructuras *bloque*, ya que cada una de ellas tiene un tamaño de 128 bits y resulta conveniente guardar sus valores en este tipo de dato, además de que las funciones de adición y doblado se codificaron para trabajar eficientemente con la estructura.

Algoritmo 8 Cifrador AES-OTR secuencial

Entrada: bloque $entrada[m]$, bloque $A[a]$, bloque k_{ini}
Salida: bloque $entrada[m]$, bloque T, bloque N

1: **entero** $l \leftarrow 0$
2: **bloque** $Z, T, TE, TA, L^*, B_1, B_2$
3: **bloque** $\Sigma \leftarrow 0$
4: **bloque** $N \leftarrow$ bytes_aleatorios()
5: **bloque** $k_r[Nr] \leftarrow$ EXPANSION_DE_LLAVE_OPT$(k_{ini}, k_r[Nr], Nr)$
6: **bloque** $L \leftarrow$ FORMATO(N)
7: $L \leftarrow$ CIFRADOR_AES_CAJAS_T$(L, k_{ini}, k_r[Nr])$
8: **para** $i = 0$ incremento en 2 hasta $i = m - 3$ **hacer**
9: $B_1 \leftarrow entrada[i]$
10: $B_2 \leftarrow entrada[i+1]$
11: $\Sigma \leftarrow \Sigma \oplus entrada[i+1]$
12: $entrada[i] \leftarrow$ CIFRADOR_AES_CAJAS_T$(entrada[i] \oplus 2^l L, k_{ini}, k_r[Nr]) \oplus B_2$
13: $entrada[i+1] \leftarrow$ CIFRADOR_AES_CAJAS_T$(entrada[i] \oplus 2^l 3L, k_{ini}, k_r[Nr]) \oplus B_1$
14: $l \leftarrow l + 1$
15: **fin para**
16: **si** m par **entonces**
17: $B_1 \leftarrow entrada[m-2]$
18: $B_2 \leftarrow entrada[m-1]$
19: $Z \leftarrow$ CIFRADOR_AES_CAJAS_T$(entrada[m-2] \oplus 2^{l-1} L, k_{ini}, k_r[Nr])$
20: $Z \leftarrow msb_t(Z)$
21: $\Sigma \leftarrow \Sigma \oplus Z \oplus \underline{entrada[m-1]}$
22: $entrada[m-1] \leftarrow Z \oplus B_2$
23: $entrada[m-2] \leftarrow$ CIFRADOR_AES_CAJAS_T$(\underline{entrada[m-1]} \oplus 2^{l-1} 3L, k_{ini}, k_r[Nr]) \oplus B_1$
24: $L^* \leftarrow 2^{l-1} 3L$
25: **si no si** m impar **entonces**
26: $B_2 \leftarrow entrada[m-1]$
27: $\Sigma \leftarrow \Sigma \oplus \underline{entrada[m-1]}$
28: $entrada[m-1] \leftarrow$ CIFRADOR_AES_CAJAS_T$(2^{l-1} L, k_{ini}, k_r[Nr])$
29: $entrada[m-1] \leftarrow msb_t(entrada[m-1]) \oplus B_2$
30: $L^* \leftarrow 2^{l-1} L$
31: **fin si**
32: **si** $|B_2| \neq 128$ bits **entonces**
33: $TE \leftarrow$ CIFRADOR_AES_CAJAS_T$(\Sigma \oplus 3^2 L^*, k_{ini}, k_r[Nr])$
34: **si no si** $|B_2| = 128$ bits **entonces**
35: $TE \leftarrow$ CIFRADOR_AES_CAJAS_T$(\Sigma \oplus 7L^*, k_{ini}, k_r[Nr])$
36: **fin si**
37: $TA \leftarrow$ AES_OTR_TA$(A[a], k_{ini}, k_r[Nr])$
38: $T \leftarrow TE \oplus TA$

Algoritmo 9 Descifrador AES-OTR secuencial

Entrada: bloque $entrada[m]$, bloque $A[a]$, bloque k_{ini}, bloque T, bloque N
Salida: bloque $entrada[m]$ ó etiqueta T no válida
1: **entero** $l \leftarrow 0$
2: **bloque** $Z, T', TE, TA, L^*, B_1, B_2$
3: **bloque** $\Sigma \leftarrow 0$
4: **bloque** $k_r[Nr] \leftarrow$ EXPANSION_DE_LLAVE_OPT$(k_{ini}, k_r[Nr], Nr)$
5: **bloque** $L \leftarrow$ FORMATO(N)
6: $L \leftarrow$ CIFRADOR_AES_CAJAS_T$(L, k_{ini}, k_r[Nr])$
7: **para** $i = 0$ incremento en 2 hasta $i = m - 3$ **hacer**
8: $B_1 \leftarrow entrada[i]$
9: $B_2 \leftarrow entrada[i+1]$
10: $entrada[i] \leftarrow$ CIFRADOR_AES_CAJAS_T$(entrada[i] \oplus 2^l 3L, k_{ini}, k_r[Nr]) \oplus B_2$
11: $entrada[i+1] \leftarrow$ CIFRADOR_AES_CAJAS_T$(entrada[i] \oplus 2^l L, k_{ini}, k_r[Nr]) \oplus B_1$
12: $\Sigma \leftarrow \Sigma \oplus entrada[i+1]$
13: $l \leftarrow l + 1$
14: **fin para**
15: **si** m par **entonces**
16: $B_1 \leftarrow entrada[m-2]$
17: $B_2 \leftarrow entrada[m-1]$
18: $Z \leftarrow$ CIFRADOR_AES_CAJAS_T$(entrada[m-2] \oplus 2^{l-1} 3L, k_{ini}, k_r[Nr])$
19: $Z \leftarrow msb_t(Z)$
20: $entrada[m-1] \leftarrow Z \oplus B_2$
21: $entrada[m-2] \leftarrow$ CIFRADOR_AES_CAJAS_T$(\underline{entrada[m-1]} \oplus 2^{l-1} L, k_{ini}, k_r[Nr]) \oplus B_1$
22: $\Sigma \leftarrow \Sigma \oplus Z \oplus \underline{entrada[m-1]}$
23: $L^* \leftarrow 2^{l-1} 3L$
24: **si no si** m impar **entonces**
25: $B_2 \leftarrow entrada[m-1]$
26: $entrada[m-1] \leftarrow$ CIFRADOR_AES_CAJAS_T$(2^{l-1} L, k_{ini}, k_r[Nr])$
27: $entrada[m-1] \leftarrow msb_t(entrada[m-1]) \oplus B_2$
28: $\Sigma \leftarrow \Sigma \oplus \underline{entrada[m-1]}$
29: $L^* \leftarrow 2^{l-1} L$
30: **fin si**
31: **si** $|entrada[m-1]| \neq 128$ bits **entonces**
32: $TE \leftarrow$ CIFRADOR_AES_CAJAS_T$(\Sigma \oplus 3^2 L^*, k_{ini}, k_r[Nr])$
33: **si no si** $|entrada[m-1]| = 128$ bits **entonces**
34: $TE \leftarrow$ CIFRADOR_AES_CAJAS_T$(\Sigma \oplus 7L^*, k_{ini}, k_r[Nr])$
35: **fin si**
36: $TA \leftarrow$ AES_OTR_TA$(A[a], k_{ini}, k_r[Nr])$
37: $T' \leftarrow TE \oplus TA$
38: **si** $T \neq T'$ **entonces Etiqueta** T **no válida**
39: **fin si**

Algoritmo 10 Generador de la etiqueta de autenticación TA en OTR

Entrada: bloque $A[a]$, bloque k_{ini}, bloque $k_r[Nr]$
Salida: bloque TA

1: **bloque** Q
2: **si** $A[a] = \emptyset$ **entonces**
3: $\quad TA \leftarrow 0$
4: **si no si** $A[a] \neq \emptyset$ **entonces**
5: $\quad Q \leftarrow$ CIFRADOR_AES_CAJAS_T$(0, k_{ini}, k_r[Nr])$
6: $\quad A[0] \leftarrow$ CIFRADOR_AES_CAJAS_T$(A[0] \oplus Q, k_{ini}, k_r[Nr])$
7: $\quad$ **para** $i = 1$ incremento en 1 hasta $i = a - 2$ **hacer**
8: $\quad\quad A[i] \leftarrow$ CIFRADOR_AES_CAJAS_T$(A[i] \oplus 2^i Q, k_{ini}, k_r[Nr])$
9: $\quad\quad A[i] \leftarrow A[i-1] \oplus A[i]$
10: $\quad$ **fin para**
11: $\quad$ **si** $|A[a-1]| \neq 128$ bits **entonces**
12: $\quad\quad A[a-1] \leftarrow \underline{A[a-1]} \oplus 2^{a-1}3Q$
13: $\quad$ **si no si** $|A[a-1]| = 128$ bits **entonces**
14: $\quad\quad A[a-1] \leftarrow \underline{A[a-1]} \oplus 2^{a-1}3^2Q$
15: $\quad$ **fin si**
16: $\quad TA \leftarrow$ CIFRADOR_AES_CAJAS_T$(A[a-1] \oplus A[a-2], k_{ini}, k_r[Nr])$
17: **fin si**

CAPÍTULO 6

Algoritmos paralelos y sus implementaciones

Al finalizar el desarrollo de las versiones secuenciales de los modos de operación CTR y OTR es posible analizarlos más detalladamente para así diseñar los algoritmos paralelos en una forma eficiente, manejando de la mejor manera las dependencias de datos encontradas.

En este capítulo se muestran los algoritmos paralelos AES-CTR y AES-OTR diseñados, tanto la versión multinúcleo con OpenMP (OMP) como la versión muchos núcleos con CUDA, discutiendo la forma en que se decidió implementarlos.

6.1. AES-CTR multinúcleo

A diferencia de las versión secuencial del algoritmo, en la versión paralela no se ocupó la estandarización de OpenSSL; se prefirió codificar una versión personal de AES-CTR que utiliza solamente el arreglo $entrada[m]$ con la finalidad de ahorrar espacio en memoria y además, ejecutar el algoritmo de expansión de llave antes de comenzar la transformación de datos, ya que si no se hace de esta manera, cada hilo generado en la versión multinúcleo deberá ejecutar su propia expansión de llaves, restando rendimiento al algoritmo. En otras palabras, esta implementación sirve como reemplazo de la función *AES_ctr128_encrypt* utilizada en los algoritmos 6 y 7, en la página 53. Los algoritmos 11 y 12 ejemplifican tanto el cifrado como el descifrado de las versiones multinúcleo de AES-CTR diseñadas en la tesis.

Algoritmo 11 Cifrador AES-CTR multinúcleo OMP

Entrada: bloque $entrada[m]$, bloque k_{ini}, entero NT
Salida: bloque $entrada[m]$
1: **bloque** $k_r[Nr] \leftarrow$ EXPANSION_DE_LLAVE_OPT$(k_{ini}, k_r[Nr], Nr)$
2: **bloque** $N \leftarrow$ bytes_aleatorios()
3: **bloque paralelo** compartido: $entrada[m], k_{ini}, k_r[Nr], N, NT$ **:**
4:     **bloque** b
5:     **entero** $ctr \leftarrow$ NUMERO_HILO()
6:     **para** $i = ctr$ incremento en NT hasta $i = m - 1$ **hacer**
7:         $b \leftarrow$ CIFRADOR_AES_CAJAS_T$(N||ctr, k_{ini}, k_r[Nr])$
8:         $entrada[i] \leftarrow entrada[i] \oplus b$
9:     **fin para**
10: **fin bloque paralelo**

Algoritmo 12 Descifrador AES-CTR multinúcleo OMP

Entrada: bloque $entrada[m]$, bloque k_{ini}, bloque N, entero NT
Salida: bloque $entrada[m]$
1: **bloque** $k_r[Nr] \leftarrow$ EXPANSION_DE_LLAVE_OPT($k_{ini}, k_r[Nr], Nr$)
2: **bloque paralelo** compartido: $entrada[m], k_{ini}, k_r[Nr], N, NT$ **:**
3: **bloque** b
4: **entero** $ctr \leftarrow$ NUMERO_HILO()
5: **para** $i = ctr$ incremento en NT hasta $i = m - 1$ **hacer**
6: $b \leftarrow$ CIFRADOR_AES_CAJAS_T($N||ctr, k_{ini}, k_r[Nr]$)
7: $entrada[i] \leftarrow entrada[i] \oplus b$
8: **fin para**
9: **fin bloque paralelo**

Las diferencias entre los algoritmos cambian con respecto a la versión secuencial debido a la nueva implementación:

1. En el cifrador se genera el valor aleatorio N pseudo-aleatoriamente, lo que es equivalente al VI de la versión secuencial. Al generarlo, éste es enviado en el mensaje cifrado; en el descifrador se lee el valor aleatorio N recibido.

2. Inicialmente en el cifrador, el arreglo $entrada[m]$ contiene el texto en claro y durante la ejecución del algoritmo el texto cifrado será generado "al vuelo" dentro del mismo arreglo; en el descifrador ocurre lo contrario.

En las líneas 3 a 10 del algoritmo 11 y en las líneas 2 a 9 del algoritmo 12, se muestra (en color rojo) el inicio y fin del bloque de pseudo-código que se ejecuta en paralelo con una cantidad NT de hilos, los cuales realizan la transformación sobre el arreglo de bloques $entrada[m]$. Esta nueva variable entera NT está incluída en los argumentos de la función. En el caso de esta tesis, puede oscilar entre los valores 2-4, ya que la cantidad mínima de hilos para ejecutar el algoritmo en paralelo es 2, y la cantidad máxima de hilos con la que es factible alcanzar una aceleración cercana a la óptima es 4, ya que la arquitectura seleccionada cuenta con cuatro núcleos. Si se desea realizar pruebas de funcionamiento en una arquitectura con mayores capacidades, el valor de NT debe modificarse.

También, en las líneas 6 a 8 del algoritmo 11 y en las líneas 5 a 7 del algoritmo 12, se muestra (en color azul) las variables que son compartidas entre hilos en tiempo de ejecución y se especifican al inicio del bloque paralelo. Dentro del bloque paralelo se usan dos variables locales por hilo: un bloque b que se utiliza para almacenar temporalmente el cifrado del valor aleatorio N concatenado con el contador y el entero ctr, cuyo valor inicial se consigue con la función *numero_hilo()*, que puede ser cualquier función que devuelva el número de hilo en ejecución, por ejemplo la función *omp_get_thread_num()* de OMP que regresa valores entre 0 y $NT - 1$. Su incremento se realiza en NT unidades ya que el arreglo de bloques $entrada[m]$ es procesado intercaladamente en vez de dividirlo en NT particiones, lo cual resulta más complejo. Por ejemplo, en un arreglo $entrada[m]$ con $m = 8$ y $NT = 4$, el hilo 0 procesa los bloques $\{entrada[0], entrada[4]\}$, el hilo 1 los bloques $\{entrada[1], entrada[5]\}$, el hilo 2 los bloques $\{entrada[2], entrada[6]\}$ y el hilo 3 los bloques $\{entrada[3], entrada[7]\}$. Nótese que

el valor de *ctr* sirve tanto para indexar el arreglo $entrada[m]$ como para llevar correctamente el valor del contador. El texto cifrado resultante se consigue al aplicar un XOR entre el resultado del bloque b y el bloque $entrada[i]$, almacenando el valor final directamente sobre el bloque $entrada[i]$.

Finalmente se hace mención sobre el grado de paralelismo con que cuenta el modo de operación CTR; toda transformación sobre el arreglo $entrada[m]$ se ejecuta dentro del bloque de pseudo-código paralelo. Con esto es posible alcanzar una aceleración cercana a la máxima aceleración esperada sobre el algoritmo, debido a que el procesamiento con mayor costo computacional se realiza dentro de este bloque paralelo. La única parte del algoritmo que se ejecuta secuencialmente es la expasión de llave, ya que este trabajo está hecho para ser enteramente secuencial; sin embargo, comparado con el trabajo que se realiza en el bloque paralelo, la expasión de llave no representa una gran carga de trabajo en el algoritmo.

6.2. AES-CTR CUDA

Al igual que en la versión multinúcleo, en la versión CUDA se adaptó el modo de operación para que pueda ejecutarse en tarjetas gráficas con CUDA, ya que la estandarización de OpenSSL no está hecha para esta tarea. En esta nueva versión CUDA se codificó el algoritmo 1 para trabajar directamente en la GPU; de no hacerlo no sería posible utilizar este componente como un acelerador criptográfico masivo. Los algoritmos 13 y 14 muestran los procesos de cifrado y descifrado AES-CTR en su versión CUDA.

Algoritmo 13 Cifrador AES-CTR CUDA

Entrada: bloque $entrada[m]$, bloque k_{ini}, entero NBL, entero NT
Salida: bloque $entrada[m]$

1: **bloque** $k_r[Nr] \leftarrow$ EXPANSION_DE_LLAVE_OPT$(k_{ini}, k_r[Nr], Nr)$
2: **bloque** $N \leftarrow$ bytes_aleatorios()
3: **bloque cuda :**
4: SUBIR_DATOS_ANFITRION_GPU()
5: **nucleo cuda** $<<< NBL, NT >>>$, compartido: $entrada[m], k_{ini}, k_r[Nr], N$ **:**
6: **bloque** b
7: **entero** $ctr \leftarrow hilo_ID + (bloque_ID \times NBL)$
8: **para** $i = ctr$ incremento en $NBL \times malla_DIM$ hasta $i = m - 1$ **hacer**
9: $b \leftarrow$ CIFRADOR_AES_CAJAS_T$(N||ctr, k_{ini}, k_r[Nr])$
10: $entrada[i] \leftarrow entrada[i] \oplus b$
11: **fin para**
12: **fin nucleo cuda**
13: BAJAR_DATOS_GPU_ANFITRION()
14: **fin bloque cuda**

Las diferencias entre el cifrador y el descifrador son exactamente las mismas que en la versión multinúcleo.

En las líneas 3 a 14 del algoritmo 13 y en las líneas 2 a 13 del algoritmo 14 se muestra (en color verde azulado) el inicio y fin del bloque CUDA. Dentro de este bloque CUDA, en las líneas 4 y 13 del algoritmo 13 y en las líneas 3 y 12 del algoritmo 14 se muestran (en

Algoritmo 14 Descifrador AES-CTR CUDA

Entrada: bloque $entrada[m]$, bloque k_{ini}, bloque N, entero NBL, entero NT
Salida: bloque $entrada[m]$
1: **bloque** $k_r[Nr] \leftarrow$ EXPANSION_DE_LLAVE_OPT($k_{ini}, k_r[Nr], Nr$)
2: **bloque cuda :**
3: SUBIR_DATOS_ANFITRION_GPU()
4: **nucleo cuda** $<<< NBL, NT >>>$, compartido: $entrada[m], k_{ini}, k_r[Nr], N$ **:**
5: **bloque** b
6: **entero** $ctr \leftarrow hilo_ID + (bloque_ID \times NBL)$
7: **para** $i = ctr$ incremento en $NBL \times malla_DIM$ hasta $i = m - 1$ **hacer**
8: $b \leftarrow$ CIFRADOR_AES_CAJAS_T($N||ctr, k_{ini}, k_r[Nr]$)
9: $entrada[i] \leftarrow entrada[i] \oplus b$
10: **fin para**
11: **fin nucleo cuda**
12: BAJAR_DATOS_GPU_ANFITRION()
13: **fin bloque cuda**

color olivo) dos nuevas funciones:

1. *subir_datos_anfitrion_GPU()*: implica cargar el arreglo $entrada[m]$ en la memoria global de la GPU; la llave inicial, las llaves de ronda, el valor aleatorio N y las cuatro cajas-T se cargan en la memoria compartida. Lo ideal sería cargar todos los datos en memoria compartida para acelerar el acceso, pero como se trabaja con mensajes de grandes tamaños esto resulta imposible, ya que la arquitectura Kepler GK210 cuenta sólamente con 128 KB de memoria compartida. Esta cantidad es suficiente para almacenar únicamente las variables constantes del algoritmo.

2. *bajar_datos_GPU_anfitrion()*: implica descargar el arreglo $entrada[m]$ resultante desde la memoria global en la GPU al terminar la transformación. Esta función resulta más eficiente que *subir_datos_anfitrion_GPU()* ya que no es necesario manipular nuevamente la memoria compartida.

En los argumentos de la función se incluye una variable entera más llamada NBL, que indica la cantidad de bloques de hilos que se invocan en la ejecución del núcleo CUDA. Sus posibles valores son: {1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024}. La variable NT indica la cantidad de hilos por cada NBL bloques y sus posibles valores son: {16, 32, 64, 128, 256, 512, 1024}; entonces, con estos valores asignados, las ejecuciones del núcleo CUDA se realizan con: $1 \times 16, \ldots, 1 \times 1024, \ldots, 2 \times 16, \ldots, 2 \times 1024, \ldots, 1024 \times 16, \ldots, 1024 \times 1024$, dando un total de 77 ejecuciones del núcleo sobre un arreglo $entrada[m]$ dado.

En las líneas 5 a 12 del algoritmo 13 y en las líneas 4 a 11 del algoritmo 14 se muestra (en color rojo) el inicio y fin del bloque que se ejecuta en paralelo (en este caso, el núcleo CUDA), junto con las variables compartidas entre hilos. Dentro del núcleo, en las líneas 7 a 10 del algoritmo 13 y en las líneas 6 a 9 del algoritmo 14 se muestra (en color azul) los accesos a las variables compartidas por cada hilo dentro del núcleo. También se incluyen las mismas variables locales b y ctr. Pero además de las variables especificadas al inicio del bloque paralelo, se destacan unas variables especiales propias de la GPU, necesarias tanto

para indexar correctamente el arreglo *entrada*[m] como para calcular el valor del contador *ctr*. Estas variables especiales son:

- *hilo_ID*: indica el número de hilo dentro del bloque
- *bloque_ID*: indica el número de bloque dentro de la malla
- *malla_DIM*: indica la dimensión de la malla

Por ejemplo, la función *numero_hilo()* del algoritmo multinúcleo se reemplaza por un cálculo equivalente en CUDA utilizando estas variables. El incremento del contador *ctr* también se realiza con base en ellas.

Este algoritmo paralelo es más complejo que la versión multinúcleo, debido a la carga y descarga de datos en la GPU. Esta operación se puede llevar grandes intervalos de tiempo en comparación con el proceso criptográfico, reduciendo así el rendimiento del algoritmo.

6.3. AES-OTR multinúcleo

Como se mencionó en la descripción del algoritmo secuencial AES-OTR, los arreglos *salida*[m] y *estado*[m] de la versión paralela también se omitieron para ahorrar espacio en memoria, por lo que las transformaciones se realizan sobre el arreglo *entrada*[m]. En esta nueva versión también se incluyó en los argumentos la variable entera NT para determinar la cantidad de hilos con que se ejecuta el algoritmo.

El algoritmo 15 muestra el diseño paralelo del cifrado AES-OTR y el algoritmo 16 muestra el descifrado paralelo. Las diferencias entre el cifrado y el descifrado son las mismas que en la versión secuencial del algoritmo.

Al analizar el modo de operación se puede ver que la parte más factible para paralelización es la transformación principal de datos, sin incluir la última partición de bloques. Los tiempos de ejecución se pueden reducir considerablemente si se acelera esta parte del algoritmo, ya que es la tarea que consume mayores recursos computacionales, tanto memoria como tiempo en CPU. Por lo tanto, si se comparan las dos versiones se puede ver que son muy similares debido a que sólamente se consideró el ciclo principal para ser paralelizado.

En las líneas 7 a 16 del algoritmo 15 y en las líneas 6 a 15 del algoritmo 16 se muestra (en color rojo) el inicio y fin del bloque paralelo junto con las variables compartidas, que en este caso son seis: *entrada*[m], k_{ini}, $k_r[Nr]$, L, Σ y NT. En las líneas 9 a 14 del algoritmo 15 y en las líneas 8 a 13 del algoritmo 16 se muestra (en color azul) los accesos a cada variable compartida dentro del bloque paralelo.

En el bloque paralelo se incluye la variable entera *nt* que representa el número de hilo que se ejecuta. Se calcula con la función *numero_hilo()*. Al iniciar el ciclo principal, *nt* debe multiplicarse por dos, ya que el algoritmo procesa $\frac{m}{2} - 1$ particiones de dos bloques por hilo. El incremento de *nt* se realiza en $NT \times 2$ para indexar intercaladamente el arreglo *entrada*[m], además de que también sirve para llevar la cuenta de la partición en proceso; esto es $\frac{i}{2}$. El número de partición en proceso sirve para calcular el doblado $2^{i/2}L$. Por ejemplo, con el arreglo *entrada*[m] con $m = 16$ y $NT = 4$, el hilo 0 calcula las particiones $\{p_0(entrada[0], entrada[1]),\ p_4(entrada[8], entrada[9])\}$, el hilo 1 las particiones $\{p_1(entrada[2], entrada[3]),\ p_5(entrada[10], entrada[11])\}$, el hilo 2 las

particiones $\{p_2(entrada[4],entrada[5]),\ p_6(entrada[12],entrada[13])\}$ y el hilo 3 la partición $\{p_3(entrada[6],entrada[7])\}$, ya que el ciclo omite la última partición debido a que se procesa de una forma particular.

Ahora bien, la expansión de llave, el cifrado de L, el cálculo de la última partición y el cálculo de la etiqueta de autenticación se realizaron secuencialmente, debido a que estas tareas no requieren de una gran cantidad de tiempo comparadas con la transformación principal de OTR. Sin embargo, el cálculo de la etiqueta TA también se puede paralelizar, aunque debido a que los datos asociados son cantidades reducidas de bytes, no se espera una aceleración en este procedimiento. Debido a esto se prefirió no realizar la paralelización en el cálculo de la etiqueta de autenticación.

Finalmente, los doblados de L se pueden calcular antes de iniciar el proceso de transformación, almacenándolos directamente en memoria. Esto puede resultar poco favorable si se cuenta con memoria reducida, pero a cambio se puede obtener mayor aceleración en el procesamiento del arreglo $entrada[m]$, ya que el doblado de L representa una operación costosa en tiempo dentro de OTR. Dado que los doblados de L se calculan conforme se procesa una partición de dos bloques, el número de cálculos a realizar previo al ciclo principal es $\frac{m}{2}$.

En las implementaciones de esta versión paralela se llevó a cabo este cálculo. El arreglo de bloques $L[\frac{m}{2}]$ se creó para almacenar todos los doblados de L. Un ejemplo de uso se describe con la línea 13 del algoritmo 15:

$$entrada[i] \leftarrow \text{CIFRADOR_AES_CAJAS_T}(entrada[i] \oplus 2^{i/2}L, k_{ini}, k_r[Nr]) \oplus B_2$$

En esta línea se realiza el doblado $2^{i/2}L$. Como ya se pre-calculó este valor y se almacenó en el arreglo $L[\frac{m}{2}]$, solamente se debe acceder a él. Entonces, la línea 13 puede reescribirse como:

$$entrada[i] \leftarrow \text{CIFRADOR_AES_CAJAS_T}(entrada[i] \oplus L[\tfrac{i}{2}], k_{ini}, k_r[Nr]) \oplus B_2$$

Un ejemplo más se da en la línea 14 del mismo algoritmo:

$$entrada[i+1] \leftarrow \text{CIFRADOR_AES_CAJAS_T}(entrada[i] \oplus 2^{i/2}3L, k_{ini}, k_r[Nr]) \oplus B_1$$

En esta línea se realiza el doblado $2^{i/2}3L$. Este valor se puede calcular rápidamente con los valores contenidos en $L[\frac{m}{2}]$ de la siguiente forma:

$$entrada[i+1] \leftarrow \text{CIFRADOR_AES_CAJAS_T}(entrada[i] \oplus L[\tfrac{i}{2}] \oplus L[\tfrac{i}{2}+1], k_{ini}, k_r[Nr]) \oplus B_1$$

Con esto se puede ver que los valores triplicados de L se calculan eficientemente con una operación XOR adicional entre el valor $2^{\frac{i}{2}}$ y el valor $2^{\frac{i}{2}+1}$, ganando así una aceleración significativa en el algoritmo.

Algoritmo 15 Cifrador AES-OTR multinúcleo OMP

Entrada: bloque $entrada[m]$, bloque $A[a]$, bloque k_{ini}, entero NT
Salida: bloque $entrada[m]$, bloque T, bloque N

1: **bloque** $Z, T, TE, TA, L^*, B_1, B_2$
2: **bloque** $\Sigma \leftarrow 0$
3: **bloque** $N \leftarrow$ bytes_aleatorios()
4: **bloque** $k_r[Nr] \leftarrow$ EXPANSION_DE_LLAVE_OPT$(k_{ini}, k_r[Nr], Nr)$
5: **bloque** $L \leftarrow$ FORMATO(N)
6: $L \leftarrow$ CIFRADOR_AES_CAJAS_T$(L, k_{ini}, k_r[Nr])$
7: **bloque paralelo** compartido: $entrada[m], k_{ini}, k_r[Nr], L, \Sigma, NT$ **:**
8: **entero** $nt \leftarrow$ NUMERO_HILO()
9: **para** $i = nt \times 2$ incremento en $NT \times 2$ hasta $i = m - 3$ **hacer**
10: $B_1 \leftarrow entrada[i]$
11: $B_2 \leftarrow entrada[i+1]$
12: $\Sigma \leftarrow \Sigma \oplus entrada[i+1]$
13: $entrada[i] \leftarrow$ CIFRADOR_AES_CAJAS_T$(entrada[i] \oplus 2^{i/2}L, k_{ini}, k_r[Nr]) \oplus B_2$
14: $entrada[i+1] \leftarrow$ CIFRADOR_AES_CAJAS_T$(entrada[i] \oplus 2^{i/2}3L, k_{ini}, k_r[Nr]) \oplus B_1$
15: **fin para**
16: **fin bloque paralelo**
17: **si** m par **entonces**
18: $B_1 \leftarrow entrada[m-2]$
19: $B_2 \leftarrow entrada[m-1]$
20: $Z \leftarrow$ CIFRADOR_AES_CAJAS_T$(entrada[m-2] \oplus 2^{(m/2)-1}L, k_{ini}, k_r[Nr])$
21: $Z \leftarrow msb_t(Z)$
22: $\Sigma \leftarrow \Sigma \oplus Z \oplus entrada[m-1]$
23: $entrada[m-1] \leftarrow Z \oplus B_2$
24: $entrada[m-2] \leftarrow$ CIFRADOR_AES_CAJAS_T$(entrada[m-1] \oplus 2^{(m/2)-1}3L, k_{ini}, k_r[Nr]) \oplus B_1$
25: $L^* \leftarrow 2^{(m/2)-1}3L$
26: **si no si** m impar **entonces**
27: $B_2 \leftarrow entrada[m-1]$
28: $\Sigma \leftarrow \Sigma \oplus entrada[m-1]$
29: $entrada[m-1] \leftarrow$ CIFRADOR_AES_CAJAS_T$(2^{(m/2)-1}L, k_{ini}, k_r[Nr])$
30: $entrada[m-1] \leftarrow msb_t(entrada[m-1]) \oplus B_2$
31: $L^* \leftarrow 2^{(m/2)-1}L$
32: **fin si**
33: **si** $|B_2| \neq 128$ bits **entonces**
34: $TE \leftarrow$ CIFRADOR_AES_CAJAS_T$(\Sigma \oplus 3^2L^*, k_{ini}, k_r[Nr])$
35: **si no si** $|B_2| = 128$ bits **entonces**
36: $TE \leftarrow$ CIFRADOR_AES_CAJAS_T$(\Sigma \oplus 7L^*, k_{ini}, k_r[Nr])$
37: **fin si**
38: $TA \leftarrow$ AES_OTR_TA$(A[a], k_{ini}, k_r[Nr])$
39: $T \leftarrow TE \oplus TA$

Algoritmo 16 Descifrador AES-OTR multinúcleo OMP

Entrada: bloque $entrada[m]$, bloque $A[a]$, bloque k_{ini}, bloque T, bloque N, entero NT
Salida: bloque $entrada[m]$ ó etiqueta T no válida

1: **bloque** $Z, T', TE, TA, L^*, B_1, B_2$
2: **bloque** $\Sigma \leftarrow 0$
3: **bloque** $k_r[Nr] \leftarrow$ EXPANSION_DE_LLAVE_OPT$(k_{ini}, k_r[Nr], Nr)$
4: **bloque** $L \leftarrow$ FORMATO(N)
5: $L \leftarrow$ CIFRADOR_AES_CAJAS_T$(L, k_{ini}, k_r[Nr])$
6: **bloque paralelo** compartido: $entrada[m], k_{ini}, k_r[Nr], L, \Sigma, NT$ **:**
7: **entero** $nt \leftarrow$ NUMERO_HILO()
8: **para** $i = nt \times 2$ incremento en $NT \times 2$ hasta $i = m - 3$ **hacer**
9: $B_1 \leftarrow entrada[i]$
10: $B_2 \leftarrow entrada[i+1]$
11: $entrada[i] \leftarrow$ CIFRADOR_AES_CAJAS_T$(entrada[i] \oplus 2^{i/2}3L, k_{ini}, k_r[Nr]) \oplus B_2$
12: $entrada[i+1] \leftarrow$ CIFRADOR_AES_CAJAS_T$(entrada[i] \oplus 2^{i/2}L, k_{ini}, k_r[Nr]) \oplus B_1$
13: $\Sigma \leftarrow \Sigma \oplus entrada[i+1]$
14: **fin para**
15: **fin bloque paralelo**
16: **si** m par **entonces**
17: $B_1 \leftarrow entrada[m-2]$
18: $B_2 \leftarrow entrada[m-1]$
19: $Z \leftarrow$ CIFRADOR_AES_CAJAS_T$(entrada[m-2] \oplus 2^{(m/2)-1}3L, k_{ini}, k_r[Nr])$
20: $Z \leftarrow msb_t(Z)$
21: $entrada[m-1] \leftarrow Z \oplus B_2$
22: $entrada[m-2] \leftarrow$ CIFRADOR_AES_CAJAS_T$(\underline{entrada[m-1]} \oplus 2^{(m/2)-1}L, k_{ini}, k_r[Nr]) \oplus B_1$
23: $\Sigma \leftarrow \Sigma \oplus Z \oplus \underline{entrada[m-1]}$
24: $L^* \leftarrow 2^{(m/2)-1}3L$
25: **si no si** m impar **entonces**
26: $B_2 \leftarrow entrada[m-1]$
27: $entrada[m-1] \leftarrow$ CIFRADOR_AES_CAJAS_T$(2^{(m/2)-1}L, k_{ini}, k_r[Nr])$
28: $entrada[m-1] \leftarrow msb_t(entrada[m-1]) \oplus B_2$
29: $\Sigma \leftarrow \Sigma \oplus \underline{entrada[m-1]}$
30: $L^* \leftarrow 2^{(m/2)-1}L$
31: **fin si**
32: **si** $|entrada[m-1]| \neq 128$ bits **entonces**
33: $TE \leftarrow$ CIFRADOR_AES_CAJAS_T$(\Sigma \oplus 3^2L^*, k_{ini}, k_r[Nr])$
34: **si no si** $|entrada[m-1]| = 128$ bits **entonces**
35: $TE \leftarrow$ CIFRADOR_AES_CAJAS_T$(\Sigma \oplus 7L^*, k_{ini}, k_r[Nr])$
36: **fin si**
37: $TA \leftarrow$ AES_OTR_TA$(A[a], k_{ini}, k_r[Nr])$
38: $T \leftarrow TE \oplus TA$
39: **si** $T \neq T'$ **entonces Etiqueta** T **no válida**
40: **fin si**

6.4. AES-OTR CUDA

Como en la versión CUDA de AES-CTR, el modo de operación OTR en su versión homónima se adaptó para operar en tarjetas gráficas que soportan CUDA. El modelo del algoritmo paralelo que se ocupó es el mismo que en la versión multinúcleo: paralelizar sólamente el ciclo principal de OTR. La misma función de cifrado directo en GPU que se desarrolló en la versión CUDA de AES-CTR se ocupó en este nuevo algoritmo paralelo. También se incluyó en los argumentos de la nueva función la variable entera NBL para indicar la cantidad de bloques con que se ejecuta el algoritmo. Esta variable mantiene los mismos valores que en la versión CUDA de AES-CTR, al igual que la variable NT. Los algoritmos 17 y 18 muestran el diseño paralelo de cifrado y descifrado OTR en la versión CUDA. Las diferencias entre ambos algoritmos son idénticas a las diferencias de la versión secuencial del algoritmo.

La función *subir_datos_anfitrion_GPU()* que se utiliza para cargar los datos necesarios en la GPU se modificó ligeramente para adaptarse al nuevo algoritmo. En esta función, además de cargarse las llaves necesarias, el valor aleatorio N y las cuatro cajas-T en memoria compartida, y el arreglo $entrada[m]$ en memoria global, también carga el arreglo $L[\frac{m}{2}]$ en memoria global para poder acceder a él durante la transformación. Esto reduce considerablemente la cantidad de información que puede modificarse en la GPU, pero es una operación necesaria debido a que el algoritmo así lo requiere. La función *bajar_datos_GPU_anfitrion()* no se modificó, ya que el único resultado que se debe recuperar al término de la transformación es el mensaje cifrado.

En las líneas 7 a 20 del algoritmo 17 y en las líneas 6 a 19 del algoritmo 18 se muestra (en color verde azulado) el inicio y fin del bloque CUDA. Dentro de este bloque CUDA, en las líneas 9 y 18 del algoritmo 17 y en las líneas 8 y 17 del algoritmo 18 se muestran (en color rojo) el inicio y fin del bloque paralelo, al igual que las variables compartidas entre hilos. En las líneas 10 a 16 del algoritmo 17 y en las líneas 9 a 15 del algoritmo 18 se muestra (en color azul) cada acceso a una variables compartida dentro del núcleo. Esta versión paralela resulta más compleja que la versión multinúcleo, ya que la carga de datos iniciales en GPU aumentó más del doble, incrementando así el intervalo de tiempo necesario para iniciar con la operación criptográfica.

Inicialmente, la variable entera nt dentro del núcleo se calcula con ayuda de las variables especiales $hilo_ID$ y $bloque_ID$, para después multiplicarse por dos en el inicio del ciclo principal, realizando sus incrementos en $NBL \times malla_DIM \times 2$ para indexar correctamente $entrada[m]$ y para llevar la cuenta correcta de la partición en proceso. La suma Σ se incluye dentro del ciclo, pero se puede calcular en un ciclo paralelo independiente para evitar pérdida en el rendimiento. En el caso de esta tesis, se calculó en un bloque paralelo aparte para evitar sincronizaciones innecesarias entre hilos durante la transformación dentro de la GPU.

Para finalizar, las partes secuenciales (expansión de llave, el cálculo de la última partición o el cálculo de la etiqueta de autenticación) se conservaron tal como el algoritmo multinúcleo.

Algoritmo 17 Cifrador AES-OTR CUDA

Entrada: bloque $entrada[m]$, bloque $A[a]$, bloque k_{ini}, entero NBL, entero NT
Salida: bloque $entrada[m]$, bloque T, bloque N

1: **bloque** $Z, T, TE, TA, L^*, B_1, B_2$
2: **bloque** $\Sigma \leftarrow 0$
3: **bloque** $N \leftarrow$ bytes_aleatorios()
4: **bloque** $k_r[Nr] \leftarrow$ EXPANSION_DE_LLAVE_OPT$(k_{ini}, k_r[Nr], Nr)$
5: **bloque** $L \leftarrow$ FORMATO(N)
6: $L \leftarrow$ CIFRADOR_AES_CAJAS_T$(L, k_{ini}, k_r[Nr])$
7: **bloque cuda :**
8: SUBIR_DATOS_ANFITRION_GPU()
9: **nucleo cuda** $<<< NBL, NT >>>$, compartido: $entrada[m], k_{ini}, k_r[Nr], L, \Sigma, NBL, NT$ **:**
10: **entero** $nt \leftarrow hilo_ID + (bloque_ID \times NBL)$
11: **para** $i = nt \times 2$ incremento en $NBL \times malla_DIM \times 2$ hasta $i = m - 3$ **hacer**
12: $B_1 \leftarrow entrada[i]$
13: $B_2 \leftarrow entrada[i+1]$
14: $\Sigma \leftarrow \Sigma \oplus entrada[i+1]$
15: $entrada[i] \leftarrow$ CIFRADOR_AES_CAJAS_T$(entrada[i] \oplus 2^{i/2}L, k_{ini}, k_r[Nr]) \oplus B_2$
16: $entrada[i+1] \leftarrow$ CIFRADOR_AES_CAJAS_T$(entrada[i] \oplus 2^{i/2}3L, k_{ini}, k_r[Nr]) \oplus B_1$
17: **fin para**
18: **fin nucleo cuda**
19: BAJAR_DATOS_GPU_ANFITRION()
20: **fin bloque cuda**
21: **si** m par **entonces**
22: $B_1 \leftarrow entrada[m-2]$
23: $B_2 \leftarrow entrada[m-1]$
24: $Z \leftarrow$ CIFRADOR_AES_CAJAS_T$(entrada[m-2] \oplus 2^{(m/2)-1}L, k_{ini}, k_r[Nr])$
25: $Z \leftarrow msb_t(Z)$
26: $\Sigma \leftarrow \Sigma \oplus Z \oplus \underline{entrada[m-1]}$
27: $entrada[m-1] \leftarrow Z \oplus B_2$
28: $entrada[m-2] \leftarrow$ CIFRADOR_AES_CAJAS_T$(\underline{entrada[m-1]} \oplus 2^{(m/2)-1}3L, k_{ini}, k_r[Nr]) \oplus B_1$
29: $L^* \leftarrow 2^{(m/2)-1}3L$
30: **si no si** m impar **entonces**
31: $B_2 \leftarrow entrada[m-1]$
32: $\Sigma \leftarrow \Sigma \oplus \underline{entrada[m-1]}$
33: $entrada[m-1] \leftarrow$ CIFRADOR_AES_CAJAS_T$(2^{(m/2)-1}L, k_{ini}, k_r[Nr])$
34: $entrada[m-1] \leftarrow msb_t(entrada[m-1]) \oplus B_2$
35: $L^* \leftarrow 2^{(m/2)-1}L$
36: **fin si**
37: **si** $|B_2| \neq 128$ bits **entonces**
38: $TE \leftarrow$ CIFRADOR_AES_CAJAS_T$(\Sigma \oplus 3^2L^*, k_{ini}, k_r[Nr])$
39: **si no si** $|B_2| = 128$ bits **entonces**
40: $TE \leftarrow$ CIFRADOR_AES_CAJAS_T$(\Sigma \oplus 7L^*, k_{ini}, k_r[Nr])$
41: **fin si**
42: $TA \leftarrow$ AES_OTR_TA$(A[a], k_{ini}, k_r[Nr])$
43: $T \leftarrow TE \oplus TA$

Algoritmo 18 Descifrador AES-OTR CUDA

Entrada: bloque $entrada[m]$, bloque $A[a]$, bloque k_{ini}, bloque T, bloque N, entero NBL, entero NT
Salida: bloque $entrada[m]$ ó etiqueta T no válida

1: **bloque** $Z, T', TE, TA, L^*, B_1, B_2$
2: **bloque** $\Sigma \leftarrow 0$
3: **bloque** $k_r[Nr] \leftarrow$ EXPANSION_DE_LLAVE_OPT$(k_{ini}, k_r[Nr], Nr)$
4: **bloque** $L \leftarrow$ FORMATO(N)
5: $L \leftarrow$ CIFRADOR_AES_CAJAS_T$(L, k_{ini}, k_r[Nr])$
6: **bloque cuda :**
7: SUBIR_DATOS_ANFITRION_GPU()
8: **nucleo cuda** $<<< NBL, NT >>>$, compartido: $entrada[m], k_{ini}, k_r[Nr], L, \Sigma, NBL, NT$ **:**
9: **entero** $nt \leftarrow hilo_ID + (bloque_ID \times NBL)$
10: **para** $i = nt \times 2$ incremento en $NBL \times malla_DIM \times 2$ hasta $i = m - 3$ **hacer**
11: $B_1 \leftarrow entrada[i]$
12: $B_2 \leftarrow entrada[i+1]$
13: $entrada[i] \leftarrow$ CIFRADOR_AES_CAJAS_T$(entrada[i] \oplus 2^{i/2}3L, k_{ini}, k_r[Nr]) \oplus B_2$
14: $entrada[i+1] \leftarrow$ CIFRADOR_AES_CAJAS_T$(entrada[i] \oplus 2^{i/2}L, k_{ini}, k_r[Nr]) \oplus B_1$
15: $\Sigma \leftarrow \Sigma \oplus entrada[i+1]$
16: **fin para**
17: **fin nucleo cuda**
18: BAJAR_DATOS_GPU_ANFITRION()
19: **fin bloque cuda**
20: **si** m par **entonces**
21: $B_1 \leftarrow entrada[m-2]$
22: $B_2 \leftarrow entrada[m-1]$
23: $Z \leftarrow$ CIFRADOR_AES_CAJAS_T$(entrada[m-2] \oplus 2^{(m/2)-1}3L, k_{ini}, k_r[Nr])$
24: $Z \leftarrow msb_t(Z)$
25: $entrada[m-1] \leftarrow Z \oplus B_2$
26: $entrada[m-2] \leftarrow$ CIFRADOR_AES_CAJAS_T$(\underline{entrada[m-1]} \oplus 2^{(m/2)-1}L, k_{ini}, k_r[Nr]) \oplus B_1$
27: $\Sigma \leftarrow \Sigma \oplus Z \oplus \underline{entrada[m-1]}$
28: $L^* \leftarrow 2^{(m/2)-1}3L$
29: **si no si** m impar **entonces**
30: $B_2 \leftarrow entrada[m-1]$
31: $entrada[m-1] \leftarrow$ CIFRADOR_AES_CAJAS_T$(2^{(m/2)-1}L, k_{ini}, k_r[Nr])$
32: $entrada[m-1] \leftarrow msb_t(entrada[m-1]) \oplus B_2$
33: $\Sigma \leftarrow \Sigma \oplus \underline{entrada[m-1]}$
34: $L^* \leftarrow 2^{(m/2)-1}L$
35: **fin si**
36: **si** $|entrada[m-1]| \neq 128$ bits **entonces**
37: $TE \leftarrow$ CIFRADOR_AES_CAJAS_T$(\Sigma \oplus 3^2L^*, k_{ini}, k_r[Nr])$
38: **si no si** $|entrada[m-1]| = 128$ bits **entonces**
39: $TE \leftarrow$ CIFRADOR_AES_CAJAS_T$(\Sigma \oplus 7L^*, k_{ini}, k_r[Nr])$
40: **fin si**
41: $TA \leftarrow$ AES_OTR_TA$(A[a], k_{ini}, k_r[Nr])$
42: $T' \leftarrow TE \oplus TA$
43: **si** $T \neq T'$ **entonces Etiqueta** T **no válida**
44: **fin si**

6.5. Optimización de AES-OTR

Durante el desarrollo de AES-OTR se comprobó que el modo de operación resulta práctico en los sistemas que necesitan proporcionar tanto confidencialidad como autenticación de datos, pero a pesar de que se puede paralelizar en cierto grado, la dependencia de datos que se encontró en los doblados de la máscara L impide que el algoritmo se pueda optimizar eficientemente. Si se desea acelerar adecuadamente el algoritmo se necesita eliminar la dependencia entre el cálculo de un doblado 2^iL con base en un doblado $2^{i-1}L$.

Para lograr esta tarea, los enmascaramientos previos a la transformación de datos se pueden combinar de cierta forma con el modo de operación CTR, reemplazando los doblados 2^iL y 2^i3L por dos nuevas variables α_i y β_i, que también son variables en el campo GF(2^{128}) y sirven como las nuevas máscaras. Por ejemplo, el procesamiento de una partición no final con dos bloques en OTR se realiza con la ecuación (3.12). En esta ecuación los valores 2^iL y 2^i3L se reemplazan por α_i y β_i, respectivamente.

El cálculo de estas variables no debe depender uno de otro, ni tampoco debe depender de un cálculo previo de las mismas variables. Una forma de obtener estas características es realizar el cálculo de α_i y β_i adicionando el número de partición en proceso (ctr_i) a L[1] y posteriormente, transformar el resultado de tal forma que se genere una máscara impredecible e irrepetible, ya que lo que se busca al cálcular una máscara es que su valor siempre sea distinto con respecto de otras máscaras. Esto se puede conseguir aplicando el algoritmo de cifrado AES sobre α_i o β_i, pero con una ligera modificación: en vez de aplicar diez rondas de cifrado, aplicar solo cuatro rondas completas y además, utilizar llaves diferentes en la transformación de cada variable [?]. Con esta operación se garantiza que los valores de las máscaras siempre serán distintos uno de otro, y el trabajo computacional requerido será mínimo.

Al definir esta transformación para calcular las nuevas máscaras α_i y β_i, también es necesario definir las llaves k_α y k_β para ser utilizadas en el cifrador AES de cuatro rondas. En el algoritmo 19 se muestra el pseudo-código de este nuevo cifrado AES y en la ecuación (6.1) se muestra el cálculo final de α_i y β_i.

$$\begin{aligned} \alpha_i &= \hat{E}_{k_\alpha}(L \oplus ctr_i) \\ \beta_i &= \hat{E}_{k_\beta}(L \oplus ctr_i) \end{aligned} \tag{6.1}$$

donde $\hat{E}_{k_\alpha}$ y $\hat{E}_{k_\beta}$ representan el cifrado AES con cuatro rondas utilizando las llaves k_α y k_β, respectivamente, y ctr_i representa el número de partición en proceso.

Al tener estas dos nuevas llaves y querer utilizarlas en el algoritmo 19, se deben realizar dos expansiones de llave más, pero en vez de expandirlas para diez rondas, expandirlas sólo para cuatro; esto es, generar cuatro llaves de ronda a partir de k_α y k_β. Con este procedimiento es posible excluir la dependencia de datos entre máscaras en OTR y por lo tanto, la paralelización del algoritmo se vuelve mucho más eficiente, tanto en requerimiento de memoria como en aceleración en las transformaciones de datos. En la figura 6.1 se ejemplifica el cálculo de las variables α_i y β_i, la ecuación (6.2) muestra las modificaciones hechas a la ecuación (3.12) para optimizar el modo de operación OTR y la figura 6.2 muestra el diagrama

[1] Realizar esta suma en GF(2^{128}) solo afecta a los cuatro bytes menos significativos de L, ya que el tamaño máximo que se trabaja en la tesis es 4 GB, y para llevar la cuenta de esta cantidad solo se necesitan estos cuatro bytes.

Algoritmo 19 Cifrador AES con 4 rondas

Entrada: bloque b, bloque k_{ini}, bloque $k_r[4]$
Salida: bloque b
1: **palabra** $w_0 \leftarrow b.w_0 \oplus k_{ini}.w_0$
2: **palabra** $w_1 \leftarrow b.w_1 \oplus k_{ini}.w_1$
3: **palabra** $w_2 \leftarrow b.w_2 \oplus k_{ini}.w_2$
4: **palabra** $w_3 \leftarrow b.w_3 \oplus k_{ini}.w_3$
5: **para** $ronda = 0$ incremento en 1 hasta $ronda = 3$ **hacer**
6: $b.w_0 \leftarrow T_0[w_{0,3}] \oplus T_1[w_{1,2}] \oplus T_2[w_{2,1}] \oplus T_3[w_{3,0}] \oplus k_r[i].w_0$
7: $b.w_1 \leftarrow T_0[w_{1,3}] \oplus T_1[w_{2,2}] \oplus T_2[w_{3,1}] \oplus T_3[w_{0,0}] \oplus k_r[i].w_1$
8: $b.w_2 \leftarrow T_0[w_{2,3}] \oplus T_1[w_{3,2}] \oplus T_2[w_{0,1}] \oplus T_3[w_{1,0}] \oplus k_r[i].w_2$
9: $b.w_3 \leftarrow T_0[w_{3,3}] \oplus T_1[w_{0,2}] \oplus T_2[w_{1,1}] \oplus T_3[w_{2,0}] \oplus k_r[i].w_3$
10: $w_0 \leftarrow b.w_0$
11: $w_1 \leftarrow b.w_1$
12: $w_2 \leftarrow b.w_2$
13: $w_3 \leftarrow b.w_3$
14: **fin para**

correspondiente al algoritmo de cifrado AES-OTR optimizado.

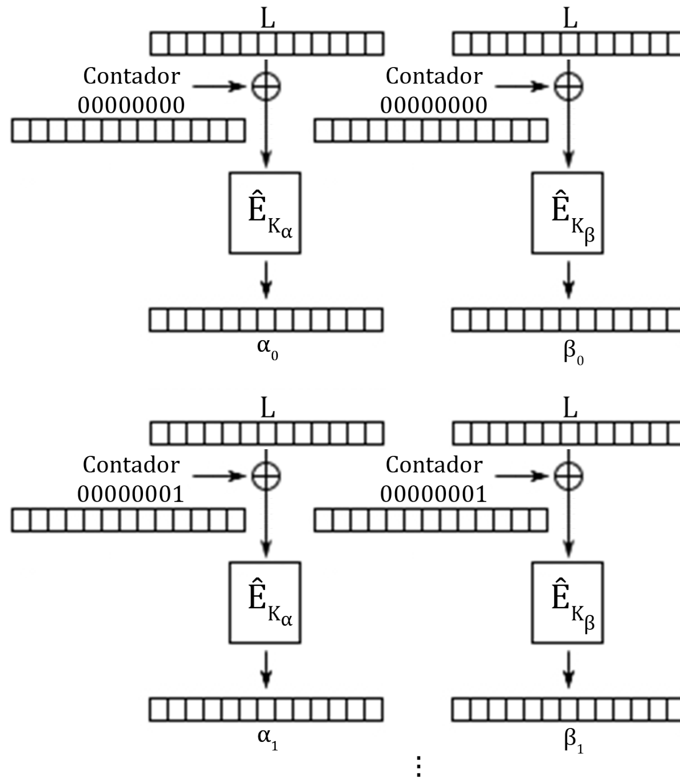


Figura 6.1: Generación de variables α_i y β_i en la optimización de OTR

$$
\begin{aligned}
&C_{2i} = E_K(P_{2i} \oplus \alpha_i) \oplus P_{2i+1}, \forall i \in [0, l-2]\\
&C_{2i+1} = E_K(C_{2i} \oplus \beta_i) \oplus P_{2i}, \forall i \in [0, l-2]\\
&P_{2i} = E_K(C_{2i} \oplus \beta_i) \oplus C_{2i+1}, \forall i \in [0, l-2]\\
&P_{2i+1} = E_K(P_{2i} \oplus \alpha_i) \oplus C_{2i}, \forall i \in [0, l-2]
\end{aligned}
\tag{6.2}
$$

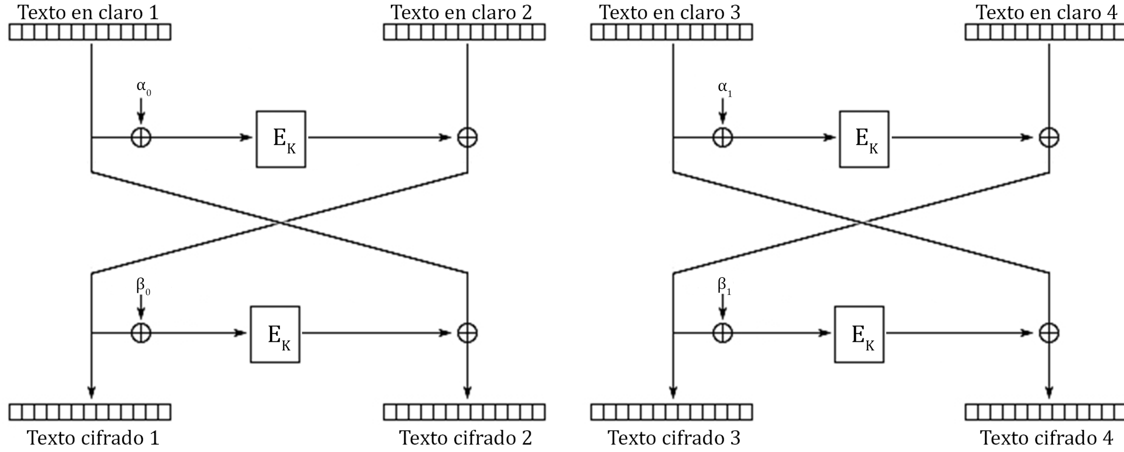


Figura 6.2: OTR optimizado en modo cifrado

Al modificar el modo de operación para calcular particiones no finales, también se debe modificar el cálculo de la partición final. Las ecuaciones (6.3) describen cómo se modificaron las ecuaciones (3.13), las cuales se utilizan en el cálculo de la última partición con un número de bloques par o impar; las figuras 6.3 y 6.4 muestran los diagramas correspondientes para ambos casos del algoritmo.

$$
m \text{ par}: \begin{cases}
C_{m-1} = MSB_t(E_K(P_{m-2} \oplus \alpha_{l-1})) \oplus P_{m-1}\\
C_{m-2} = E_K(\underline{C_{m-1}} \oplus \beta_{l-1}) \oplus P_{m-2}\\
P_{m-1} = MSB_t(E_K(C_{m-2} \oplus \beta_{l-1})) \oplus C_{m-1}\\
P_{m-2} = E_K(\underline{P_{m-1}} \oplus \alpha_{l-1}) \oplus C_{m-2}
\end{cases}
\tag{6.3}
$$

$$
m \text{ impar}: \begin{cases}
C_{m-1} = MSB_t(E_K(\alpha_{l-1})) \oplus P_{m-1}\\
P_{m-1} = MSB_t(E_K(\alpha_{l-1})) \oplus C_{m-1}
\end{cases}
$$

Al efectuar esta modificación, los recursos de cómputo necesarios para proveer integridad y autenticación de datos se decrementan considerablemente. Incluso el costo computacional de este nuevo algoritmo se puede comparar con el costo computacional de CTR original, considerándolo mínimo de entre varios modos de operación.

Ahora, en el caso de la autenticación de datos, los cambios que se realizaron fueron mínimos. Los cálculos de las etiquetas T y TE permanecen igual. La diferencia radica en

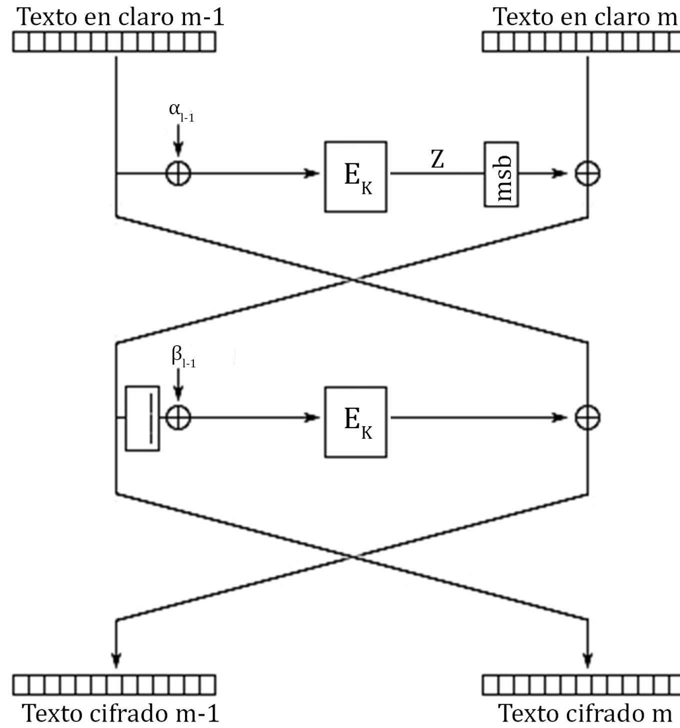


Figura 6.3: Cifrado OTR optimizado para la última partición compuesta por un número de bloques par

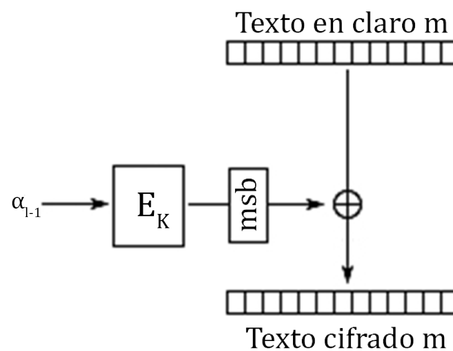


Figura 6.4: Cifrado OTR optimizado para la última partición compuesta por un número de bloques impar

las variables Σ, L^* y TA. Para calcular Σ y L^* se ocupan las nuevas máscaras α_{l-1} y β_{l-1}, como se muestra en las ecuaciones (6.4) y (6.5). El algoritmo de cálculo para TA también es el mismo, pero en vez de utilizar la llave inicial k_{ini} se utiliza una llave k_γ, con la intención de incrementar el nivel de seguridad en la etiqueta de autenticación.

$$\Sigma = \begin{cases} \begin{cases} Z = E_K(P_{m-2} \oplus \alpha_{l-1}) \\ P_1 \oplus P_3 \oplus \cdots \oplus P_{m-3} \oplus Z \oplus \underline{C_{m-1}} \end{cases} , \text{si m par} \\ P_1 \oplus P_3 \oplus \cdots \oplus P_{m-2} \oplus P_{m-1}, \text{si m impar} \end{cases} \tag{6.4}$$

$$L^* = \begin{cases} \beta_{l-1}, \text{si m par} \\ \alpha_{l-1}, \text{si m impar} \end{cases} \tag{6.5}$$

Finalmente, la operación de doblado sobre una variable en GF(2^{128}) no se excluyó totalmente. En el caso que el último bloque a procesar sea incompleto (no cuente con 128 bits de longitud), se realizarán operaciones de doblado sobre la máscara α_{l-1}; en caso contrario, los doblados serán sobre la máscara β_{l-1}. En cualquier caso, la cantidad de doblados que se ejecutarán sobre alguna de las máscaras es dos, por lo que el tiempo computacional necesario para llevarlos a cabo es mínimo y por lo tanto, no afecta el rendimiento del algoritmo[2].

[2] Las demostraciones de seguridad pertinentes se encuentran pendientes y se consideran trabajo futuro

CAPÍTULO 7

Ejecución de algoritmos paralelos y sus resultados

Al tener las implementaciones tanto de los algoritmos secuenciales como de los algoritmos paralelos completamente funcionales, el siguiente paso es realizar las pruebas necesarias para comprobar que se obtuvo un beneficio.

En este capítulo se especifica el medio de almacenamiento que se utilizó para proporcionar las entradas a los algoritmos y persistir los resultados. También se muestran las gráficas de tiempos y rendimientos obtenidos en las distintas versiones de los algoritmos. Los tiempos y rendimentos registrados que se utilizaron para generar las gráficas de este capítulo se incluyen en los apéndices B y C.

7.1. Almacenamiento y tiempos medidos

Para proporcionar las entradas de datos a los algoritmos se generaron archivos aleatorios con tamaños desde 16 bytes hasta 4 GB. El medio de almacenamiento que se utilizó es un disco duro de estado sólido *Kingston SSDNOW 300* de 60 GB, para acelerar los tiempos de lectura y escritura de datos. El sistema de archivos que se utilizó es *ext4*, ya que el dispositivo seleccionado funciona más eficientemente con él. Cada algoritmo se ejecutó sobre las mismas entradas, excepto los algoritmos OTR, en los que se omitió la entrada de 16 bytes para verificar los peores casos del modo de operación (procesamiento de particiones de dos bloques). En cada una de las pruebas se registraron los tiempos transcurridos para realizar una tarea en particular. Para las versiones multinúcleo y muchos núcleos implementadas con OMP y con CUDA, respectivamente, los tiempos que se midieron son:

1. *Tiempo de subida (carga) de datos a GPU*: el tiempo total para copiar los datos leídos desde disco a la memoria global de la GPU (solo CUDA)

2. *Tiempo de bajada (descarga) de datos desde GPU*: el tiempo total para copiar el resultado desde la memoria global de la GPU (solo CUDA)

3. *Tiempo inherentemente secuencial del algoritmo*: el tiempo total para realizar las operaciones secuenciales del algoritmo (generación de VIs o valores aleatorios, expansiones de llave, etc.)

4. *Tiempo paralelo del algoritmo*: el tiempo total para realizar operaciones en paralelo en el algoritmo (cifrado o descifrado)

5. *Tiempo de generación de la etiqueta de autenticación*: el tiempo total para generar y comprobar la etiqueta de autenticación (solo OTR y OTR optimizado)

En el caso de las versiones secuenciales, el tiempo total de ejecución representa el tiempo inherentemente secuencial del algoritmo, ya que el tiempo paralelo es cero. Además, para todas las versiones de los algoritmos, se midieron dos tiempos más: *tiempo de lectura*, que es el tiempo total que toma leer la entrada desde disco, y *tiempo de escritura*, que es el tiempo total que toma escribir el resultado en disco.

En las versiones paralelas se realizaron pruebas con diferentes cantidades de hilos. Para las versiones multinúcleo las pruebas se hicieron con 2, 3 y 4 hilos de ejecución, y para cada una de estas configuraciones de hilos se realizaron diez ejecuciones. Para las versiones muchos núcleos, las pruebas se hicieron con configuraciones de bloques e hilos de 1×16, 1×32, ..., 1024×16, 1024×32, ..., tal como se describió en el capítulo 6 y, de igual forma, cada unas estas configuraciones se ejecutó diez veces.

7.2. Lectura y escritura

En la ejecución de cada algoritmo sobre las entradas de bytes, se registraron los tiempos de lectura y escritura de datos. Se calculó el tiempo promedio que lleva leer o escribir cada entrada de bytes, tomando los tiempos registrados en cada ejecución de todos los algoritmos con sus distintas configuraciones de hilos. En la tabla B.1 se muestran los tiempos promedio de lectura y escritura de datos en el disco de estado sólido especificado. En los resultados se puede apreciar que las operaciones de escritura resultan más eficientes que las operaciones de lectura, a pesar de que en las especificaciones del disco se indica que sus velocidades son las mismas. Debido a que estas operaciones no forman parte de los algoritmos criptográficos, los tiempos registrados de lectura y escritura no se consideran en los tiempos totales de ejecución de los algoritmos, tanto secuenciales como paralelos.

7.3. Subida y bajada desde GPU

Las operaciones de subida y bajada de datos son exclusivas de las versiones CUDA de los algoritmos. En los tres modos de operación, las cuatro cajas-T, la llave inicial k_{ini} y las llaves de ronda $k_r[Nr]$ se almacenan en memoria constante para acelerar su acceso. En el caso de CTR y OTR optimizado, un bloque extra también se carga en memoria constante: el VI para CTR y la máscara inicial L para OTR optimizado. Todas las entradas $entrada[m]$ a procesar se cargan en la memoria global de la GPU, ya que es la memoria con mayor capacidad de almacenamiento. En el modo OTR, las máscaras pre-calculadas 2^iL también se cargan en memoria global, debido a que para entradas de gran tamaño, las memorias constante y compartida no son suficientes para almacenar estos valores. Esto reduce significativamente la cantidad de bloques de $entrada[m]$ que pueden cargarse en la GPU para ser procesados en una sola invocación del núcleo CUDA. Con esto se provoca que el número de invocaciones al núcleo se incremente y por ende, los tiempos de carga también lo hagan. En los modos de operación CTR y OTR optimizado esto no ocurre, ya que no es necesario cargar grandes cantidades de datos pre-calculados adicionales para realizar la transformación de $entrada[m]$.

Al finalizar la ejecución del núcleo, la cantidad de datos que se transfiere desde la memoria global de la GPU a la memoria de la CPU es sólamente el número de bloques que se procesaron, ya que no es necesario manipular nuevamente la memoria constante ni compartida de la GPU. El propio mecanismo de la GPU se encarga de liberar el espacio ocupado por las variables constantes de los algoritmos. En la tabla B.2 se muestran los tiempos promedio de subida y bajada que se registraron en las ejecuciones de cada algoritmo paralelo versión CUDA, para los tres modos de operación. En los valores se observa que los tiempos de subida y bajada del modo OTR son mayores a los tiempos de los modos CTR y OTR optimizado, debido a la carga extra de las máscaras pre-calculadas y al número mayor de invocaciones al núcleo. También se aprecia que los tiempos de OTR optimizado resultan similares a los tiempos de CTR, mostrando así la eliminación de dependencia de datos en el modo de operación OTR.

Estas operaciones no forman parte de los algoritmos criptográficos, pero resultan indispensables para realizar las transformaciones en las versiones CUDA. Por lo tanto, los tiempos de subida y bajada sí se consideran en los tiempos de ejecución totales de los algoritmos paralelos muchos núcleos.

7.4. Ejecuciones secuenciales y paralelas

Los tiempos de ejecución secuenciales y paralelos son muy importantes en los resultados, debido a que con ellos es posible saber qué tan eficiente es un algoritmo y cuanta aceleración se puede obtener de la paralelización del mismo. En las versiones secuenciales de los modos de operación CTR y OTR, los tiempos registrados incluyen: generación pseudo-aleatoria de un VI o valor aleatorio N (depende del modo de operación), expansión de llave y transformación de datos. En el caso de OTR, también se incluye el cálculo de la máscara inicial L y los tiempos de generación de la etiqueta de autenticación T. En las versiones multinúcleo, la transformación de datos se calcula como tiempo paralelo; en el modo OTR, los tiempos de pre-cálculo de los $\frac{m}{2}$ doblados de la máscara L se adicionan al tiempo secuencial del algoritmo, ya que estos cálculos se deben realizar secuencialmente. En las versiones muchos núcleos, las operaciones de subida y bajada de datos en la GPU se incluyen como tiempo secuencial.

Para generar la etiqueta de autenticación a partir de datos asociados se utilizó la cantidad de 1 KB. Esta cantidad se considera excesiva para los datos asociados, ya que normalmente este tipo de datos son muy reducidos. Al probar con esta gran cantidad de datos asociados se sometió al algoritmo a un caso extremo de generación de la etiqueta. Para cada entrada proporcionada, los datos asociados se consideraron constantes, por lo que los tiempos que se registraron en cada generación de la etiqueta son similares. El tiempo promedio que se calculó para generar la etiqueta de autenticación en los modos OTR y OTR optimizado es de 23,06 microsegundos, lo cual resulta mínimo en comparación con los tiempos de cifrado o descifrado de datos.

En las tablas B.3 y B.4 se muestran los mejores tiempos de ejecución que se registraron en el cifrado de datos utilizando el modo de operación CTR. En la versión multinúcleo, los mejores resultados se consiguieron con 4 hilos de ejecución y en la versión muchos núcleos, los mejores resultados los dio la configuración de 16 $\times$ 256. En la tabla B.4 se indican dos tiempos totales distintos: *tiempo total parcial* y *tiempo total*. El *tiempo total parcial* no incluye

los tiempos de subida y bajada de datos de la GPU y el *tiempo total* sí lo hace. Estos dos tiempos se calcularon de esta forma con la finalidad de medir exclusivamente el tiempo que tarda el algoritmo en finalizar el cifrado de datos y así, percatarnos que los tiempos de subida y bajada decrementan considerablemente el rendimiento del algoritmo. En la figura 7.1 se muestra la gráfica resultante (en escala logarítmica) de los tiempos promedio registrados para el modo de operación CTR.

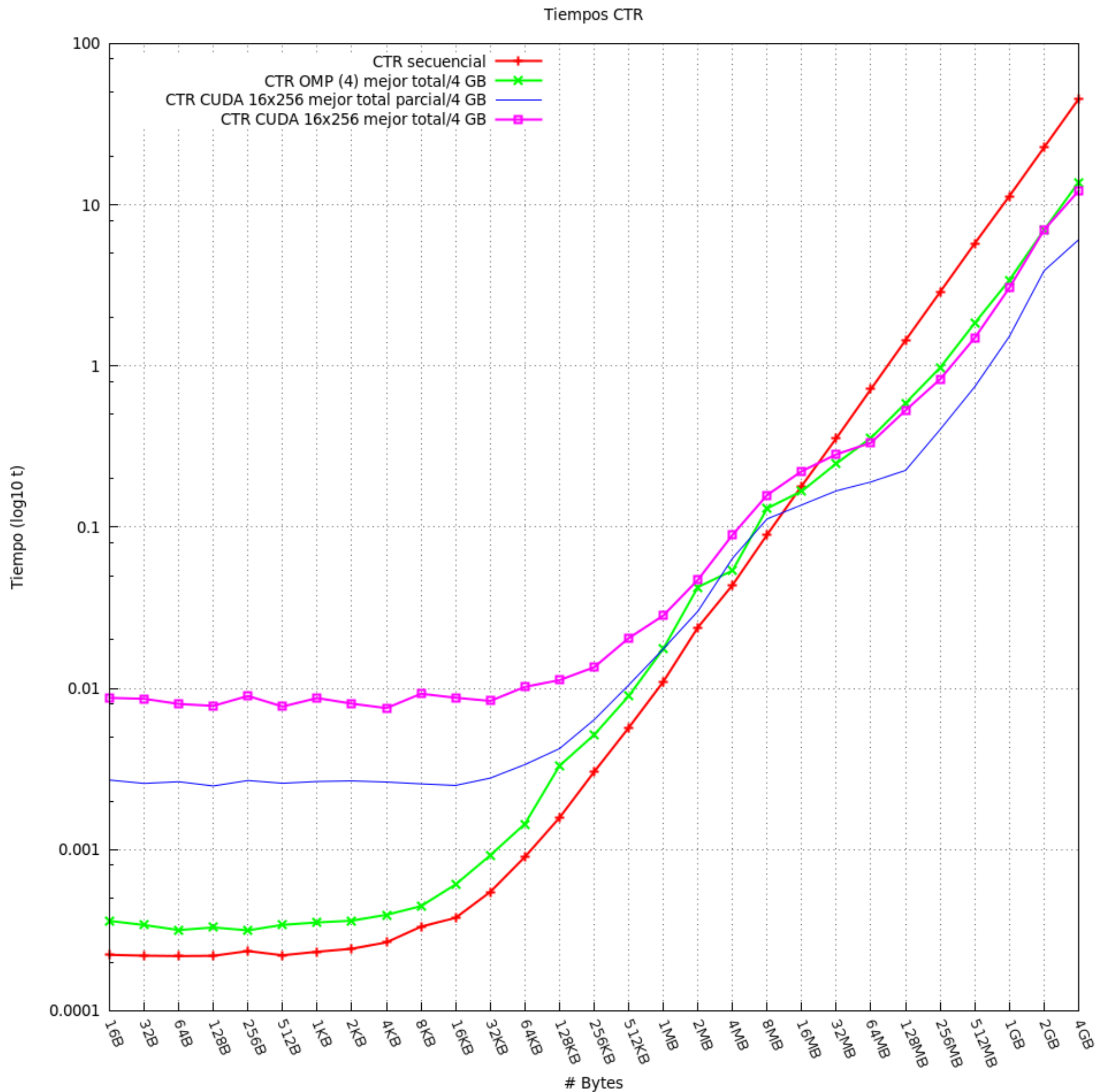


Figura 7.1: Gráfica de tiempos de ejecución para CTR (en escala logarítmica)

Revisando los resultados se puede apreciar que las versiones paralelas logran superar a la versión secuencial hecha con bibliotecas de OpenSSL. Comparando la versión multinúcleo contra la versión muchos núcleos, se observa que la implementación hecha con CUDA supera a la implementación hecha con OMP en entradas de 64 MB o mayores. Si se considera el

tiempo total parcial de la versión muchos núcleos, el cifrado de datos resulta mucho más eficiente que el de la versión multinúcleo pero, dado que las operaciones de subida y bajada de datos en la GPU forzosamente se deben realizar, se obliga al algoritmo a reducir su eficiencia y por lo tanto obtener resultados similares al algoritmo multinúcleo.

Además, en la tabla D.1 se muestran las mejores aceleraciones, tanto reales como parciales, que se consiguieron en el cifrado de datos utilizando el modo de operación CTR. Los resultados indican que las aceleraciones obtenidas superan a la aceleración constante secuencial, la cual es igual a 1. En la versión multinúcleo, la aceleración mayor que se logró fue en la entrada de 1 GB, siendo ésta 3,318 veces más rápida que la versión secuencial. Este resultado se consiguió con 4 hilos de ejecución. Tomando en cuenta que la aceleración máxima que es posible obtener de una ejecución paralela con 4 hilos es 4, se considera que el resultado es bastante eficiente. En la versión muchos núcleos, la aceleración máxima se obtuvo en la entrada de 512 MB con una configuración de 1024 $\times$ 64. Esta ejecución fue 3,921 veces más rápida que la secuencial e incluso, superó a la aceleración multinúcleo en 0,603. Si se considera solo la aceleración parcial, la ejecución alcanza una aceleración máxima 7,984 veces mayor que la secuencial. Dado que el modo de operación CTR es el más paralelizable tratado en este trabajo, se estima que este valor es la aceleración límite que se puede alcanzar en la arquitectura que se utilizó.

En el caso del modo de operación OTR, en la tabla B.5 se muestran los mejores tiempos de ejecución obtenidos de la versión multinúcleo utilizando 4 hilos de ejecución. A diferencia del modo CTR, se nota que los tiempos secuenciales de este modo de operación son mayores, debido a los pre-cálculos de las máscaras 2^iL y a la generación de la etiqueta de autenticación, que como se describió en el capítulo 6, se realizan secuencialmente. Realizar estos pre-cáculos adicionales decrementa el rendimiento del algoritmo tanto en tiempo de ejecución como en la cantidad de bloques que se almacenan en memoria para procesarse. También se incrementa el intervalo de tiempo en el que los hilos no están realizando algún tipo de procesamiento sobre la entrada. A pesar de esto, los resultados indican que la versión paralela multinúcleo supera notablemente a la versión secuencial, incrementando el tiempo secuencial pero también decrementando el tiempo paralelo del algoritmo.

En las tablas B.6 y B.7 se muestran los tiempos de ejecución promedio de la versión CUDA de OTR con las configuraciones 4 $\times$ 1024 y 256 $\times$ 1024. Los tiempos incluídos en la tabla B.6 representan los tiempos totales menores registrados; esto es, la suma de los tiempos totales es la menor de entre todas las configuraciones probadas. Los tiempos incluídos en la tabla B.7 se muestran porque con la configuración 256 $\times$ 1024 se obtuvo el cifrado más rápido sobre la entrada de 4 GB. Los resultados muestran que la implementación en CUDA también supera al algoritmo secuencial OTR, pero si se compara la versión muchos núcleos con la versión multinúcleo, se observa que la versión muchos núcleos es menos eficiente que la versión multinúcleo, incluso si se considera solamente el tiempo total parcial del algoritmo. Este resultado se produjo debido a que cada hilo de ejecución debe acceder mayormente a la memoria global de la GPU para obtener los valores de las máscaras pre-calculadas 2^iL y a su vez, a la partición de bloques que está en proceso de transformación, generando así accesos masivos a la memoria global y produciendo un cuello de botella en el bus de datos. Esto implica que solamente una cantidad reducida de hilos puede operar sobre la memoria global y por ende, se incrementa el número de hilos que permanecen en espera de acceder a los valores necesarios para realizar el cifrado de datos, decrementando considerablemente el rendimiento

del algoritmo muchos núcleos. Tomando en cuenta el tiempo total de ejecución, el algoritmo muchos núcleos se ve todavía menos eficiente que el algoritmo multinúcleo debido a las operaciones de subida y bajada de datos, con lo que se producen resultados aproximadamente 50 % menos eficientes que la versión multinúcleo. En la figura 7.2 se muestra la gráfica resultante (en escala logarítmica) de los tiempos promedio registrados para el modo de operación OTR.

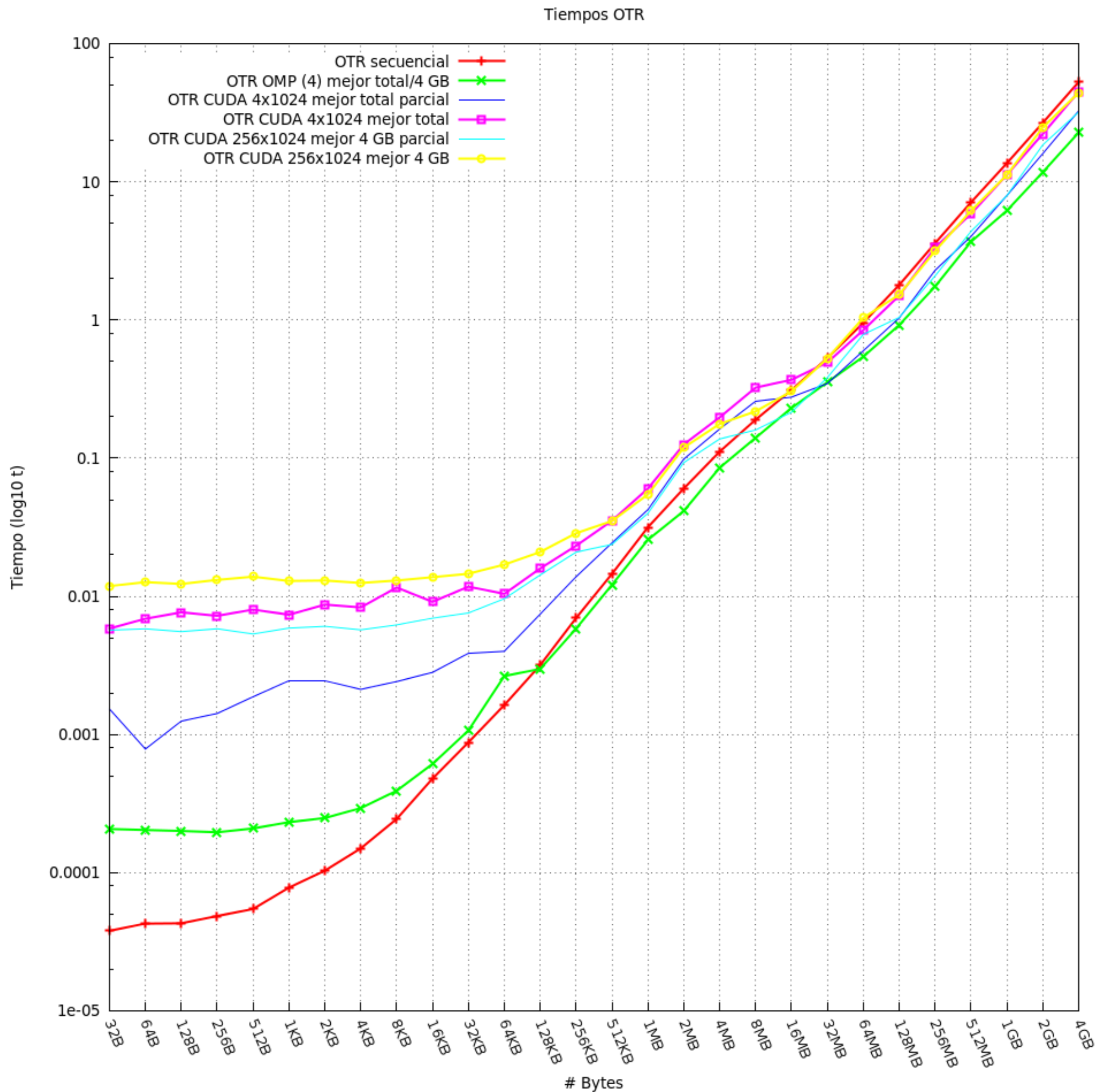


Figura 7.2: Gráfica de tiempos de ejecución para OTR (en escala logarítmica)

Con respecto a las aceleraciones, en la tabla D.2 se muestran las mejores aceleraciones obtenidas de la versión multinúcleo utilizando 4 hilos de ejecución. Se observa que las aceleraciones de OTR son menores que las de CTR, por las operaciones secuenciales extra que se realizan en este modo de operación. La aceleración máxima que se logró fue de 2,32 en la

entrada de 4 GB. No obstante, este resultado es mejor que el de la versión muchos núcleos. La implementación hecha con CUDA alcanzó una aceleración máxima de 1,27 en la entrada de 512 MB, lo cual resulta poco óptimo comparado con la versión multinúcleo, incluso si se considera solo la aceleración parcial (1,87).

En las tablas B.8 y B.9 se muestran los mejores tiempos de ejecución que se registraron en el cifrado de datos utilizando la optimización que se planteó al modo de operación OTR. Al igual que los algoritmos multinúcleo CTR y OTR, los mejores resultados se consiguieron con 4 hilos, y en la versión muchos núcleos los mejores resultados se consiguieron con una configuración de 64 × 1024. Los resultados indican que la optimización que se realizó al modo OTR es más eficiente que las versión secuencial original y a las versiones paralelas que se diseñaron. Al comparar las versiones multinúcleo de OTR y OTR optimizado, se observa que los tiempos inherentemente secuenciales contrastan significativamente en las ejecuciones con entradas grandes. Esto se debe a que los cálculos de las máscaras α_i y β_i en OTR optimizado se realizaron en paralelo independientes uno de otro. Con ello, se redujo el tiempo secuencial y se incrementó ligeramente el tiempo paralelo, el cual a pesar de incluir una cantidad mayor de cálculos que el modo OTR original, también se ve decrementado. Nótese que el tiempo paralelo influye mayormente en el tiempo total de ejecución, al igual que la versión multinúcleo del modo CTR. Esto confirma que la implementación multinúcleo de OTR optimizado es altamente paralela y por lo tanto, puede proporcionar resultados comparables al modo CTR.

La versión muchos núcleos de OTR optimizado también genera resultados más eficientes que la versión secuencial de OTR original, pero si se comparan los tiempos totales de ejecución contra los tiempos totales de la versión multinúcleo, se observa que, al igual que OTR original, resulta menos eficiente, debido a los accesos extra a los valores de las cuatro cajas-T en el cifrado AES de cuatro rondas, a los accesos a las llaves k_α y k_β con sus respectivas expansiones de llave, y a los accesos masivos al valor constante de la máscara inicial L. A pesar de que el tiempo total parcial resulta más eficiente que el tiempo total de la versión multinúcleo, los tiempos de transferencia de datos entre la CPU y la GPU decrementan la eficiencia del algoritmo, al grado de considerar más óptimo el algoritmo multinúcleo con OMP que el algoritmo muchos núcleos con CUDA. En la figura 7.3 se muestra la gráfica resultante (en escala logarítmica) de los tiempos promedio registrados para el modo de operación OTR optimizado.

Además, en la tabla D.3 se muestran las aceleraciones máximas que se registraron en el cifrado de datos. En la versión multinúcleo se ganó una aceleración de 2,915 en la entrada de 512 MB con 4 hilos de ejecución, lo cual representa un mejor resultado que la aceleración de OTR original en su versión multinúcleo. En la versión muchos núcleos la mejor ejecución se dio en la entrada de 256 MB con una configuración de 2 × 1024, obteniendo una aceleración de 2,49. Este resultado permanece por debajo de las aceleraciones ganadas en CTR, pero se aproximan bastante a la aceleración máxima estimada para la arquitectura que se seleccionó. Si se considera solo la aceleración parcial de la ejecución, se obtiene una ganancia máxima de 3,349, superando así a los mejores resultados de CTR multinúcleo.

De esta forma, los resultados que brinda el modo OTR optimizado en su versión multinúcleo son más aproximados a los resultados del modo CTR, pero a diferencia de éste, el modo OTR optimizado proporciona tanto integridad como autenticación de datos en intervalos de tiempo cortos. Con esto se confirma que los modos de operación tratados en el trabajo

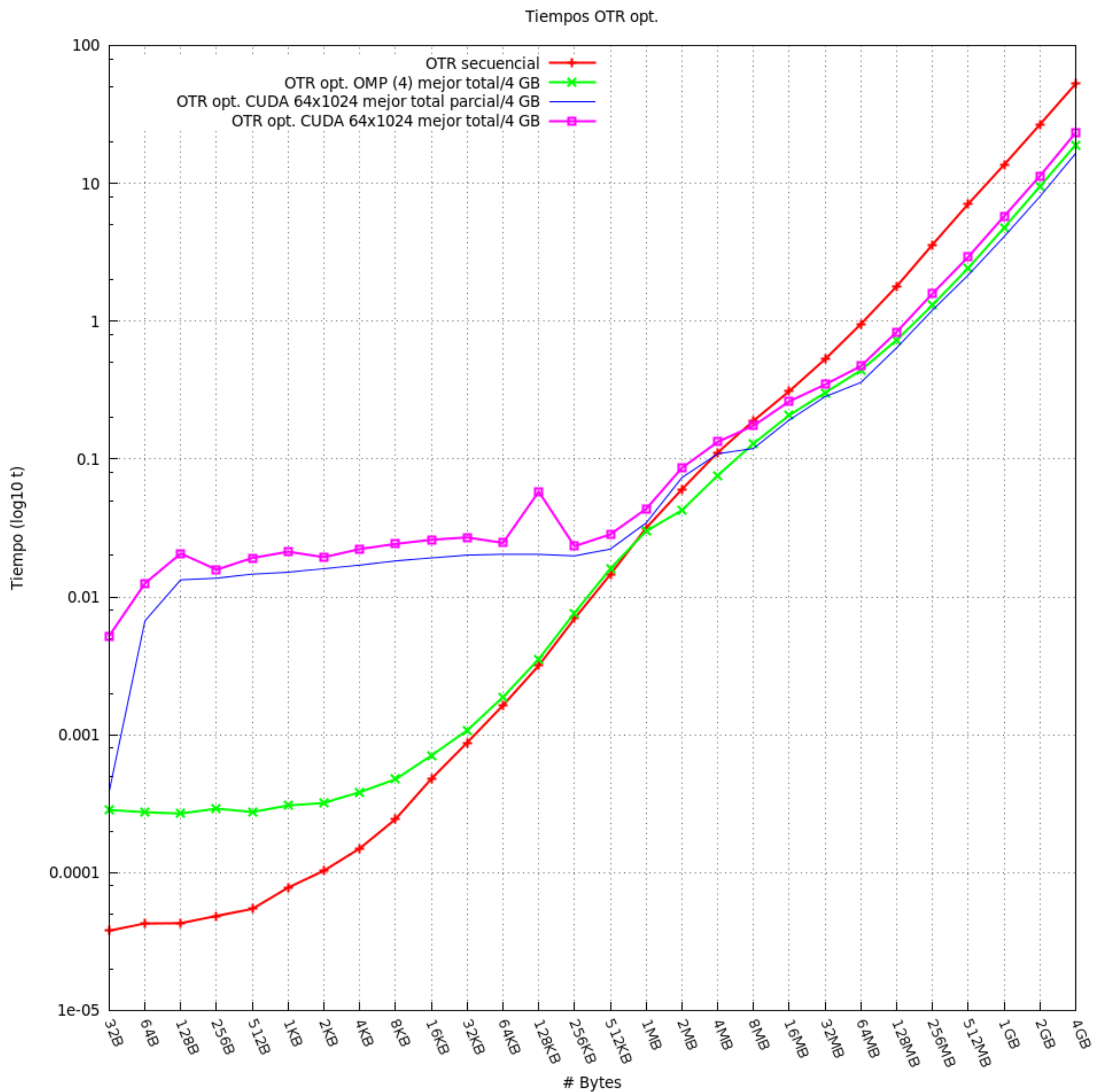


Figura 7.3: Gráfica de tiempos de ejecución para OTR optimizado (en escala logarítmica)

son factibles a paralelizar en cualquier sistema que los implemente y así obtener ganancias en los tiempos de ejecución para proveer integridad y autenticación de datos.

7.5. Rendimiento

Al tener los tiempos de ejecución totales de cada algoritmo, es posible calcular los rendimientos que se consiguieron con las optimizaciones hechas a los modos CTR y OTR. Conociendo los rendimientos es posible saber la cantidad de datos que se procesaron por segundo en las ejecuciones de los algoritmos. Estos rendimientos se calcularon como:

$$R = \frac{ent \times 8}{T} \tag{7.1}$$

donde R es el rendimiento medido en Gb/s, ent es el tamaño de la entrada (en GB) y T es el tiempo de ejecución del algoritmo en cuestión. Nótese que ent es multiplicado por 8; esto se realiza de esta forma con la intención de calcular el número de gigabits totales en la entrada.

En la tabla C.1 se muestran los rendimientos, tanto reales como parciales, que se consiguieron en el cifrado de datos utilizando el modo de operación CTR, y la figura 7.4 muestra la gráfica de rendimientos registrados.

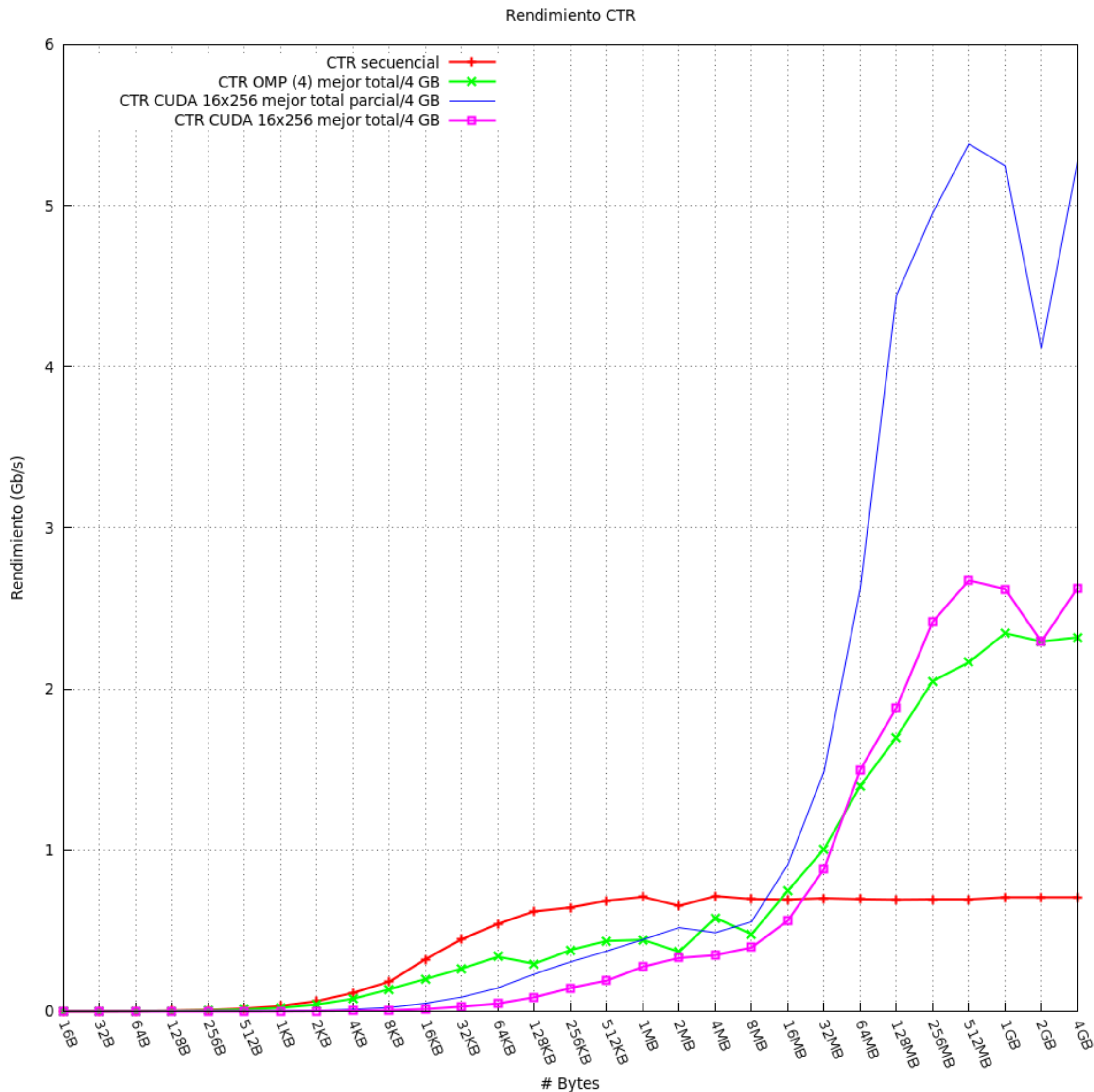


Figura 7.4: Gráfica de rendimientos para CTR

Los resultados indican que los rendimientos obtenidos superan al rendimiento secuencial.

En la versión multinúcleo, el rendimiento mayor que se logró fue en la entrada de 1 GB, siendo éste de 2,34 Gb/s. Este resultado se consiguió con 4 hilos de ejecución. En la versión muchos núcleos, el rendimiento máximo se obtuvo en la entrada de 512 MB con una configuración de 16 × 256, siendo éste de 2,67 Gb/s. Esta ejecución superó en 1,97 Gb/s al rendimiento secuencial y en 0,5 Gb/s al rendimiento multinúcleo. Si se considera solo el rendimiento parcial, la ejecución alcanza un rendimiento máximo 4,68 veces mayor que el secuencial.

Con respecto al modo de operación OTR, en la tabla C.2 se muestran los mejores rendimientos obtenidos. Se observa que los rendimientos de OTR son menores que los de CTR, por las operaciones secuenciales extra que se realizan en este modo de operación. En la versión multinúcleo, el rendimiento máximo que se logró fue de 1,41 Gb/s en la entrada de 4 GB con 4 hilos de ejecución. La implementación hecha con CUDA alcanzó un rendimiento máximo de 0,72 Gb/s en la entrada de 1 GB, lo cual resulta muy similar al rendimiento secuencial. En la figura 7.5 se muestra la gráfica resultante con los rendimientos calculados del modo de operación OTR.

Finalmente, en la tabla C.3 se muestran los rendimientos máximos que se registraron en el cifrado de datos del modo de operación OTR optimizado. En la versión multinúcleo se ganó un rendimiento de 1,688 Gb/s en la entrada de 2 GB con 4 hilos de ejecución, lo cual representa un mejor resultado que el rendimiento de OTR original en su versión multinúcleo. En la versión muchos núcleos, la mejor ejecución se dio en la entrada de 2 GB con una configuración de 2 × 1024, obteniendo un rendimiento de 1,42 Gb/s, el cual se asemeja bastante al rendimiento multinúcleo. Si se considera sólo el rendimiento parcial de la ejecución, se obtiene un rendimiento máximo de 2,00. En la figura 7.6 se muestra la gráfica resultante de los rendimientos del modo de operación OTR optimizado.

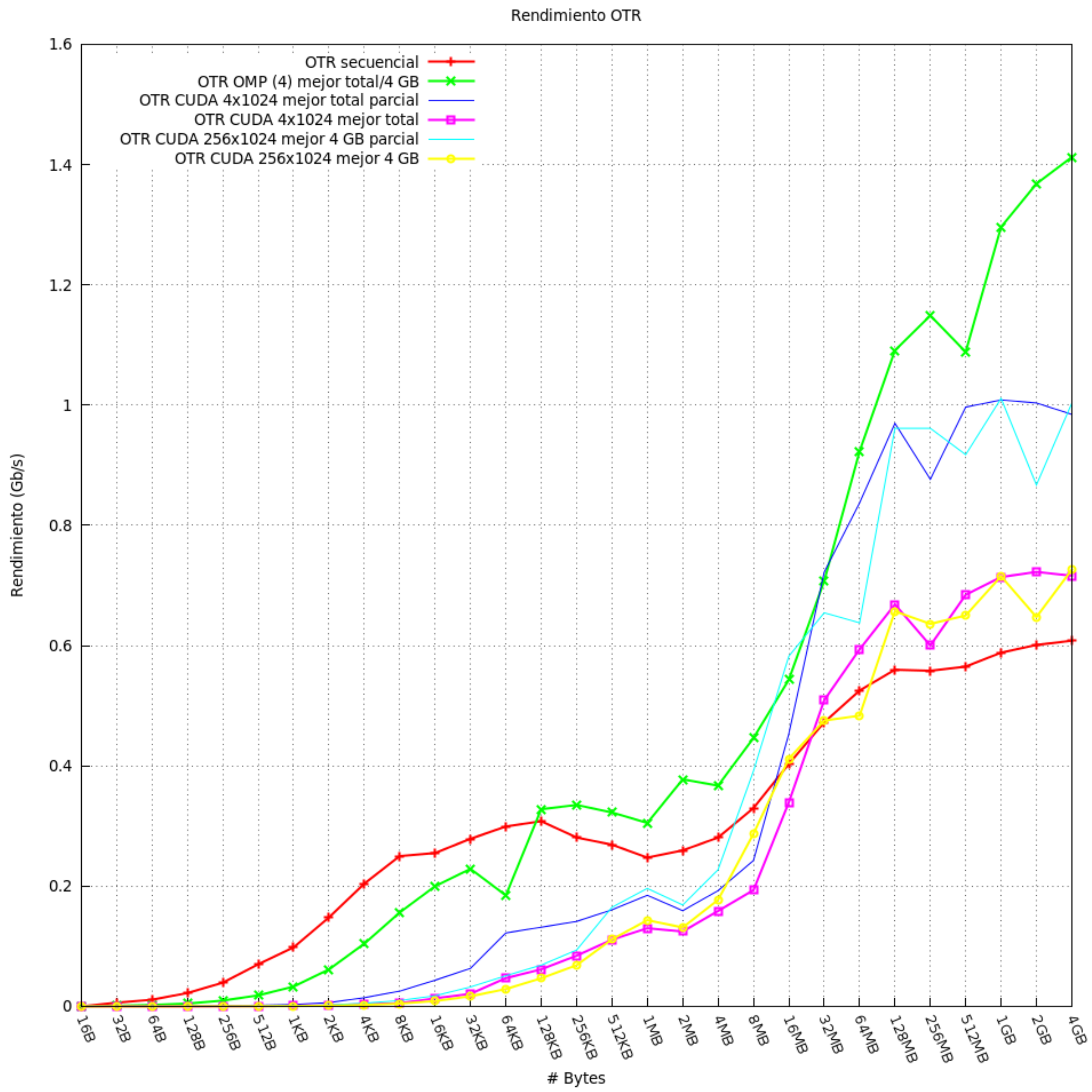


Figura 7.5: Gráfica de rendimientos para OTR

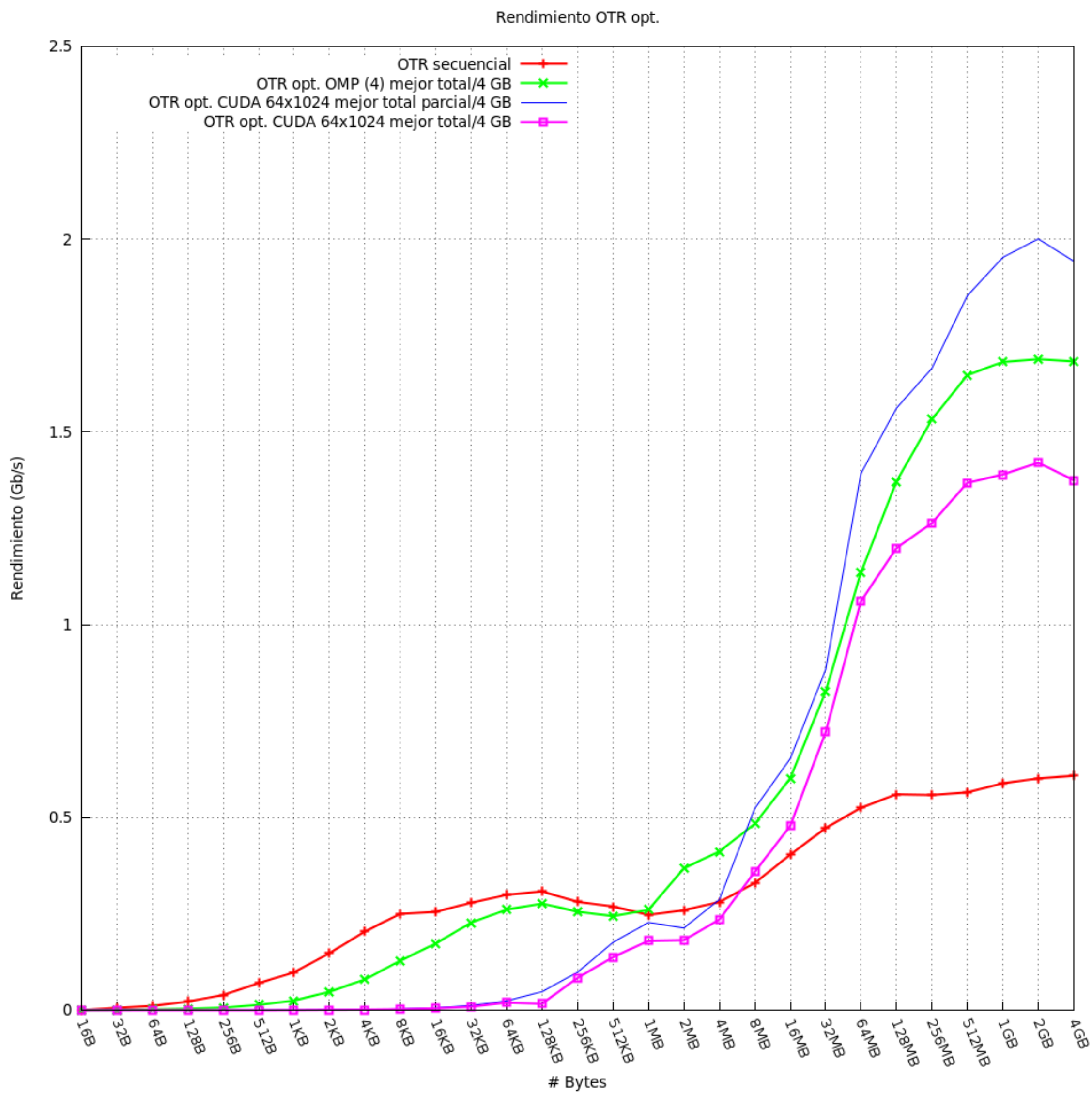


Figura 7.6: Gráfica de rendimientos para OTR optimizado

CAPÍTULO 8

Conclusiones

En esta tesis se generaron nuevos diseños paralelos optimizados de los algoritmos criptográficos AES-CTR y AES-OTR, con su respectiva implementación sobre una plataforma móvil. Estos diseños son capaces de invocar varias instancias de AES en paralelo, tanto en CPU como en GPU, para transformar grandes volúmenes de datos en una forma más eficiente que los algoritmos secuenciales clásicos y así, disminuyen los tiempos de ejecución del proceso criptográfico.

En la investigación que se realizó sobre el algoritmo AES se notó que las transformaciones de suma de la llave de ronda y sustitución de bytes se pueden paralelizar. En ambas transformaciones, el cálculo de un byte no depende de algún otro dentro del estado, por lo que se pueden crear diferentes hilos para procesar los bytes en paralelo. Sin embargo, dado que el cifrador por bloques acepta únicamente 16 bytes como entrada, se prefirió no realizar este procedimiento, debido a que la latencia que se genera en la creación de los hilos puede llegar a ser mayor que el ejecutar estas dos transformaciones secuencialmente. Por esta razón, se optó por utilizar la optimización cajas-T, con la cual se puede implementar un AES sumamente eficiente pre-calculando dos transformaciones.

Con respecto al modo de operación CTR, al implementar la versión secuencial se notó que utilizar las bibliotecas de OpenSSL puede ahorrarnos una gran cantidad de tiempo en la codificación, lo cual resulta bastante práctico, pero decrementa ligeramente el rendimiento del algoritmo, ya que en la función *AES_ctr128_encrypt* incluída en las bibliotecas de OpenSSL, se ejecuta la expansión de llave cada que ésta se manda llamar con un nuevo bloque a procesar. Dado que el objetivo de la tesis fue conseguir optimizaciones eficientes, en los algoritmos paralelos que se diseñaron se prefirió reemplazar esta función por códigos optimizados que realizan la expansión de llaves una sola vez antes de iniciar con el cifrado o descifrado de los bloques de entrada.

En el modo de operación OTR, se apreció que los cálculos de las máscaras 2^iL consumen una cantidad significativa de tiempo, al igual que la suma de bloques en claro pares Σ en la generación de la etiqueta TE. Si el número de bloques de entrada es pequeño, se tiene la impresión que estas operaciones no decrementan el rendimiento del algoritmo, pero si el número de bloques a procesar es muy elevado, se observa que al realizar estos cálculos se incrementa el tiempo total. Por ello, en las versiones paralelas de OTR, se optó por utilizar un arreglo $L[\frac{m}{2}]$ con los valores de todas las máscaras 2^iL para transformar los bloques de entrada impares, y en la transformación de los bloques pares, utilizar el mismo arreglo $L[\frac{m}{2}]$

para calcular las máscaras 2^i3L, tal como se describió en el capítulo 6. Además, se prefirió realizar la suma de bloques en claro pares Σ con un ciclo paralelo independiente del ciclo paralelo principal, esto para evitar sincronizaciones innecesarias de los hilos. En el cálculo de esta suma, cada hilo creado adiciona los bloques que le corresponden en una suma Σ_i propia, y al terminar, realiza una sola actualización a la suma total Σ.

También se propuso una optimización al modo OTR que elimina los cálculos secuenciales en las máscaras 2^iL y 2^i3L que se adicionan al bloque de entrada antes del cifrado AES. En esta propuesta las máscaras 2^iL y 2^i3L se reemplazan por las máscaras α_i y β_i, cuyo cálculo solo depende de un cifrado AES parcial con cuatro rondas y del número de partición en proceso adicionado al valor inicial de la máscara L. Con esta modificación, se provoca que el modo de operación OTR sea más factible a la paralelización, al igual que modo CTR y por lo tanto, las aceleraciones máximas estimadas sean mayores en arquitecturas paralelas.

Para comprobar que los algoritmos paralelos son más eficientes que los algoritmos secuenciales, primero se obtuvo un tiempo secuencial base para diversas entradas de datos, las cuales constan de tamaños que van desde 16 B hasta 4 GB. Después, sobre las mismas entradas, se ejecutaron los algoritmos paralelos, registrando los tiempos que tardaron en finalizar su ejecución. En cada ejecución de las distintas versiones paralelas, se utilizó un número diferente de hilos, con la intención de encontrar la configuración que produce la aceleración máxima en paralelo. Además, se registraron los tiempos que toma llevar a cabo una tarea en particular. Los tiempos que se midieron son: *lectura de datos a cifrar*, *escritura de datos cifrados*, *transferencia de datos a cifrar a la memoria global de la GPU*, *transferencia de datos cifrados desde la memoria global de la GPU*, *tiempos inherentemente secuenciales del algoritmo*, *tiempos paralelos del algoritmo* y *tiempo de generación de la etiqueta de autenticación*, este último exclusivamente en los algoritmos OTR; este tiempo se estimó con una cantidad excesiva de datos asociados: 1 KB. Al haber sometido al algoritmo a generar una etiqueta con esta cantidad se probó un caso extremo de generación, dando un tiempo de 23,06 microsegundos, lo cual representa un porcentaje menor al 0.0001 % con respecto al tiempo total de ejecución.

Los tiempos de transferencia entre CPU y GPU se midieron solamente en los algoritmos muchos núcleos implementados con CUDA, los cuales se ven seriamente afectados por la latencia que se genera en las transferencias de datos. También se ven afectados por la cantidad masiva de accesos a las variables compartidas entre todos los hilos que se generan, al grado de ser superados en varios casos por los algoritmos multinúcleo. Para reducir los tiempos de acceso a los valores compartidos se utilizó la memoria constante y la memoria compartida, ya que son las memorias que proporcionaron los mejores resultados en velocidades de acceso. A pesar de estas desventajas, los resultados que se obtuvieron muestran que a partir de ciertas cantidades de bloques, los tiempos de ejecución de los algoritmos paralelos muchos núcleos son menores que los tiempos de los algoritmos secuenciales y los algoritmos paralelos multinúcleo, y con ello se obtiene una ganancia en aceleración. En la tabla 8.1 se indica la cantidad de bytes a partir de los cuales se recomienda ejecutar los algoritmos paralelos para obtener una ganancia en aceleración.

Los tiempos y aceleraciones finales de todas las versiones paralelas de los distintos modos de operación se compararon contra los resultados de las versiones secuenciales clásicas de los mismos algoritmos. Los resultados finales indican que los algoritmos paralelos son capaces de realizar el cifrado o descifrado de datos en intervalos de tiempo menores que los algoritmos

	CTR		OTR		OTR opt.	
	Entrada	**# hilos**	**Entrada**	**# hilos**	**Entrada**	**# hilos**
OMP	16 MB	4	128 KB	4	1 MB	4
CUDA	64 MB	1024×64	16 MB	16×1024	16 MB	2×1024

Tabla 8.1: Cantidades mínimas recomendadas para obtener una ganancia en aceleración en el cifrado/descifrado de datos, tanto con algoritmos multinúcleo como algoritmos muchos núcleos en cada modo de operación

secuenciales, incrementando así la cantidad de datos que pueden ser procesados y persistirlos en algún medio de almacenamiento o transferirlos a través de un canal de comunicación en caso de ser requerido. De esta forma, el rendimiento de los algoritmos criptográficos se incrementó y por lo tanto, es más factible permitir a otras aplicaciones aprovechar mayormente tanto el uso de CPU como de GPU en la arquitectura que se seleccione.

Como comentario extra, se menciona que los mismos algoritmos que se desarrollaron en esta tesis se probaron en una arquitectura paralela diferente: un servidor con un microprocesador Intel I7 y una tarjeta gráfica NVIDIA Tesla C2070. La entrada de prueba que se utilizó para todos los algoritmos fue la de tamaño máximo: 4 GB. Las configuraciones de hilos que se escogieron para las pruebas son las que mostraron mejor eficiencia en la arquitectura móvil. En las tablas E.1, E.2, E.3, E.4, E.5 y E.6 se muestran los tiempos que se obtuvieron en estas pruebas. Como se puede apreciar en las tablas mencionadas, en esta arquitectura los resultados son superiores a los que se mostraron en el capítulo 7. En la tabla 8.2 se muestra el resumen de resultados obtenidos de las pruebas realizadas sobre la entrada de mayor tamaño (4 GB) en la tarjeta Jeton TK1 y en la tabla 8.3 se muestra el resumen de resultados obtenidos de las pruebas realizadas sobre la misma entrada en el servidor mencionado. Si se compara la aceleración máxima de AES-CTR CUDA en la tarjeta Jetson TK1 (3,463) contra la aceleración máxima en el servidor (21,105), el resultado en el servidor sobresale notablemente. Además, los resultados obtenidos en la optimización hecha a AES-OTR CUDA demuestran que el modo de operación es capaz de brindar un rendimiento similar a AES-CTR CUDA. Con esto se demuestra que si se cuenta con una mejor arquitectura, los algoritmos diseñados son capaces de brindar aceleraciones significativas comparados con los algoritmos secuenciales clásicos.

En la actualidad, los nuevos diseños de hardware se están enfocando a generar arquitecturas altamente paralelas. En la mayoría de los casos, estos diseños incluyen como dos de sus principales componentes un microprocesador multinúcleo y una tarjeta gráfica, ya sea en equipos móviles o de escritorio. Debido a que este tipo de arquitecturas se han vuelto muy populares entre desarrolladores e investigadores, estas implementaciones se pueden utilizar en todo sistema heterogéneo que incluya arquitecturas altamente paralelas para mejorar la seguridad y la eficiencia del software que se genere en ellas.

Un trabajo a futuro que se puede derivar de esta tesis es el diseño e implementación de algoritmos criptográficos capaces de proveer diversos servicios de seguridad sobre arquitecturas más poderosas. Como se mostró en los resultados, al planificar adecuadamente mejoras a los algoritmos criptográficos, es posible obtener optimizaciones eficientes de los mismos (como fue el caso de AES-CTR CUDA y AES-OTR opt. CUDA ejecutados en el servidor).

Algoritmo	Tiempo (s)	Rendimiento (Gb/s)	Aceleración
AES-CTR OpenSSL	45.274	0.706	1
AES-CTR OMP	13.788	2.32	3.283
AES-CTR CUDA	12.18	2.627	3.463
AES-OTR secuencial	52.627	0.608	1
AES-OTR OMP	22.658	1.412	2.322
AES-OTR CUDA	44.686	0.716	1.1
AES-OTR opt. OMP	19.016	1.682	2.767
AES-OTR opt. CUDA	23.279	1.374	2.201

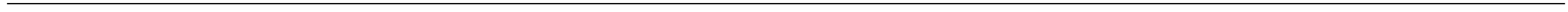
Tabla 8.2: Resumen de resultados obtenidos en cada algoritmo desarrollado sobre la entrada de 4 GB en la tarjeta Jetson TK1

Algoritmo	Tiempo (s)	Rendimiento (Gb/s)	Aceleración
AES-CTR OpenSSL	52.367	0.611	1
AES-CTR OMP	10.86	2.946	4.822
AES-CTR CUDA	2.481	12.896	21.105
AES-OTR secuencial	25.768	1.241	1
AES-OTR OMP	11.184	2.86	2.303
AES-OTR CUDA	15.225	2.101	1.692
AES-OTR opt. OMP	12.959	2.469	1.988
AES-OTR opt. CUDA	2.905	11.012	8.868

Tabla 8.3: Resumen de resultados obtenidos en cada algoritmo desarrollado sobre la entrada de 4 GB en el servidor con microprocesador Intel I7 y GPU NVIDIA Tesla C2070

También se puede derivar la creación de GPUs con instrucciones dedicadas a la criptografía, para así reducir la cantidad de ciclos de reloj que conlleva realizar una tranformación sobre

una entrada dada. Además, se puede plantear una arquitectura que comparta la memoria global de la GPU con la CPU, esto para eliminar por completo los tiempos de carga y descarga de datos. Si se llegara a contar con este tipo de arquitecturas, todos los algoritmos que se llegaran a ejecutar en ellas incrementarían considerablemente su rendimiento.



Apéndices

APÉNDICE A

Preliminares

Con el fin de brindar una mejor comprensión sobre la teoría de números y algebra que será utilizada a lo largo de la tesis, en esta sección se presenta una introducción básica basada en [?], [?] y [?].

A.1. Grupos

Definición 1. *Un grupo Abeliano $< G, + >$ está formado por un conjunto G junto con una operación definida sobre sus elementos, denotada como "+":*

$$+ : G \times G \to G : (a, b) \mapsto a + b \tag{A.1}$$

Además la operación "+" debe cumplir las siguientes condiciones:

$$\begin{aligned} \textit{cerradura:} &\quad \forall\ a, b \in G : a + b \in G \\ \textit{asociatividad:} &\quad \forall\ a, b, c \in G : (a + b) + c = a + (b + c) \\ \textit{conmutatividad:} &\quad \forall\ a, b \in G : a + b = b + a \\ \textit{elemento neutro:} &\quad \exists\ \boldsymbol{0} \in G, \forall\ a \in G : a + \boldsymbol{0} = a \\ \textit{elementos inversos:} &\quad \forall\ a \in G, \exists\ b \in G : a + b = \boldsymbol{0} \end{aligned}$$

El ejemplo más conocido de un grupo Abeliano es $< \mathbb{Z}, + >$, el conjunto de números enteros con la operación de adición (suma). La estructura $< \mathbb{Z}_n, + >$ es otro ejemplo; contiene los números enteros en el intervalo $[0, n - 1]$ y su operación es la adición módulo n.

A.2. Anillos

Definición 2. *Un anillo $< R, +, \cdot >$ está formado por un conjunto R junto con dos operaciones definidas sobre sus elementos, denotadas como "+" y "·". Las operaciones "+" y "·" deben cumplir las siguientes condiciones:*

I. *La estructura $< R, + >$ es un grupo Abeliano.*

II. *La operación "·" es cerrada y asociativa sobre R, además de que existe un elemento neutro para "·" en R.*

III. *Las operaciones "+" y "·" están relacionadas por la ley distributiva:*

$$\forall\ a, b, c \in R : (a + b) \cdot c = (a \cdot c) + (b \cdot c) \tag{A.2}$$

El elemento neutro de la operación "·" se denota como **1**. Un anillo $< R, +, \cdot >$ es llamado *anillo conmutativo* si la operación "·" es conmutativa.

El ejemplo más conocido de un anillo es $< \mathbb{Z}, +, \cdot >$, el conjunto de números enteros con las operaciones de adición y multiplicación; este ejemplo, a su vez, también es un anillo conmutativo.

A.3. Campos

Definición 3. *La estructura $< F, +, \cdot >$ es considerada un campo si se cumplen las siguientes dos condiciones:*

I. $< F, +, \cdot >$ *es un anillo conmutativo.*

II. *Para todos los elementos de F, existe un elemento inverso en F con respecto a la operación "·", con excepción del elemento neutro **0** de $< F, + >$.*

$$\forall\ a, b, c \in R : (a + b) \cdot c = (a \cdot c) + (b \cdot c) \tag{A.3}$$

Además, una estructura $< F, +, \cdot >$ es un campo si y solo si $< F, + >$ y $< F \setminus \{\mathbf{0}\}, \cdot >$ son grupos Abelianos y se aplica la ley distributiva. El elemento neutro de $< F \setminus \{\mathbf{0}\}, \cdot >$ es llamado el *elemento unidad* del campo.

El ejemplo más conocido de un campo es $< \mathbb{R}, +, \cdot >$, el conjunto de números reales con las operaciones de adición y multiplicación. Nótese que el número de elementos en este campo es infinito.

A.4. Campos finitos

Un *campo finito* es un campo que consta de un número finito de elementos. El número de elementos en el campo es llamado el *orden* del campo. Un campo de orden m existe si y solo si m es una potencia prima (i.e. $m = p^n$ para algún entero n y un entero p primo); p es llamado la *característica* del campo finito.

Los campos del mismo orden son considerados *isomórficos*; esto es, para cada potencia prima existe exactamente un campo finito, denotado como $\mathrm{GF}(p^n)$.

Los ejemplos de campos finitos más intuitivos son los campos de orden primo p. Los elementos del campo finito $\mathrm{GF}(p)$ pueden ser representados por los enteros $0, 1, 2, \ldots, p - 1$. Las dos operaciones del campo son: *adición entera módulo p* y *multiplicación entera módulo p*.

Para campos finitos de orden no primo, las operaciones *adición* y *multiplicación* no pueden ser representadas como adición y multiplicación de números enteros módulo un número; para poder hacerlo, representaciones más complejas deben ser definidas. Los campos finitos $\mathrm{GF}(p^n)$ con $n > 1$ pueden ser representados de varias formas. La representación de $\mathrm{GF}(p^n)$ por medio de polinomios sobre $\mathrm{GF}(p)$ es muy popular y será descrita en la siguiente sección.

A.5. Polinomios sobre un campo

Un polinomio sobre un campo F es una expresión de la forma:

$$b(x) = b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + \cdots + b_2x^2 + b_1x + b_0 \tag{A.4}$$

donde x es llamado el *indeterminante* del polinomio y $b_i \in F$ son los *coeficientes*. Estos polinomios nunca son evaluados. El *grado* de un polinomio es igual a l si $b_j = 0, \forall j > l$, y l es el número más pequeño con esta propiedad. El conjunto de polinomios sobre un campo F se denota como $F[x]$. El conjunto de polinomios sobre un campo F que tiene un grado menor que l se denota como $F[x]|_l$.

En la memoria de una computadora, los polinomios en $F[x]|_l$ pueden ser almacenados eficientemente guardando los coeficientes l como una cadena de caracteres (string). Por ejemplo, en el campo finito GF(2), con $l = 8$, un polinomio puede ser convenientemente almacenado como valores de 8 bits (un byte), que usualmente es abreviado utilizando notación hexadecimal:

$$b(x) \mapsto b_7b_6b_5b_4b_3b_2b_1b_0 \tag{A.5}$$

El polinomio en $\text{GF}(2)|_8 : x^6 + x^4 + x^2 + x + 1$ corresponde al string de bits 01010111, o 57 en notación hexadecimal.

A.6. Operaciones en polinomios

A.6.1. Adición

La suma de polinomios consiste en sumar los coeficientes con potencias de x iguales; la suma de los coeficientes ocurre en el campo F:

$$c(x) = a(x) + b(x) \Leftrightarrow c_i = a_i + b_i, 0 \leq i < n \tag{A.6}$$

El elemento neutro en la adición es el polinomio cuyos coeficientes son iguales a cero. El elemento inverso de un polinomio puede ser encontrado reemplazando cada coeficiente por su elemento inverso en F. El grado de $c(x)$ es a lo más el grado máximo de $a(x)$ y $b(x)$, por lo que la adición se considera cerrada; entonces, la estructura $< F[x]|_l, + >$ es un grupo abeliano. Por ejemplo, la suma de los polinomios denotados por 57 y 83 es el polinomio D4, dado que:

$$\begin{aligned}
&(x^6 + x^4 + x^2 + x + 1) \oplus (x^7 + x + 1) \\
&\qquad = x^7 + x^6 + x^4 + x^2 + (1 \oplus 1)x + (1 \oplus 1) \\
&\qquad = x^7 + x^6 + x^4 + x^2
\end{aligned}$$

En notación binaria tenemos: $01010111 \oplus 10000011 = 11010100$. Claramente, la adición puede ser implementada fácilmente con una operación bit a bit XOR.

A.6.2. Multiplicación

Recordando un poco, la multiplicación de polinomios es asociativa, conmutativa y distributiva con respecto a la suma de polinomios. Existe un elemento neutro: el polinomio de grado cero cuyo coeficiente x^0 es igual a uno. Para poder hacer la multiplicación cerrada sobre el campo $F[x]|_l$, se selecciona un polinomio $m(x)$ de grado l, llamado *polinomio de reducción*. Ahora bien, la multiplicación de dos polinomios $a(x)$ y $b(x)$ se define como el producto algebraico de los polinomios módulo el polinomio $m(x)$:

$$c(x) = a(x) \cdot b(x) \Leftrightarrow c_i \equiv a_i \times b_i \mod m(x) \tag{A.7}$$

Por lo tanto, la estructura $< F[x]|_l, +, \cdot >$ es un anillo conmutativo. Para elecciones especiales del polinomio de reducción $m(x)$, la estructura se convierte en un campo.

Definición 4. *Un polinomio $d(x)$ es irreducible sobre el campo GF(p) si y solo si no existen dos polinomios $a(x)$ y $b(x)$ con coeficientes en GF(p) tal que $d(x) = a(x) \times b(x)$, con $a(x)$ y $b(x)$ de grado mayor a cero.*

El elemento inverso para la multiplicación puede ser encontrado por medio del *algoritmo extendido de Euclides* (véase [**?**], p. 81). Sea $L(x)$ el polinomio al que se le quiere encontrar la inversa. El algoritmo extendido de Euclides puede ser utilizado para encontrar dos polinomios $b(x)$ y $c(x)$ tal que:

$$a(x) \times b(x) + m(x) \times c(x) = mcd(a(x), m(x)) \tag{A.8}$$

donde $mcd(a(x), m(x))$ denota el máximo común divisor de los polinomios $a(x)$ y $m(x)$, que es siempre igual a uno si y solo si $m(x)$ es irreducible. Aplicando reducción modular a (A.8) se obtiene:

$$a(x) \times b(x) \equiv 1 \mod m(x) \tag{A.9}$$

donde $b(x)$ es el elemento inverso de $a(x)$ por la definición de la multiplicación dada en (A.7).

Finalmente, sea F un campo GF(p); si se selecciona adecuadamente un polinomio irreducible, la estructura $< F[x]|_n, +, \cdot >$ es un campo con p^n elementos, denotado como GF(p^n). Nótese que, a diferencia de la adición, no existe una operación bit a bit capaz de realizar la multiplicación, pero ésta pueda ser implementada a nivel de bytes con corrimientos de bits a la izquierda y reduciendo el resultado módulo $m(x)$. La reducción se consigue restando el polinomio $m(x)$ (XOR $m(x)$). Esta operación se denota como $xtime()$. Las multiplicaciones por potencias mayores de x se pueden implementar repitiendo la aplicación de $xtime()$. Por ejemplo, en el campo GF(2^8), con las variables $A = \{57\}$, $B = \{13\}$ y el polinomio irreducible $m(x) = \{1b\}$, la multipliación $A \cdot B$ se realiza:

$$A \cdot B = 57 \cdot 13 = fe$$

debido a que:

$$\begin{aligned} 57 \cdot 02 &= xtime(57) = ae \\ 57 \cdot 04 &= xtime(ae) = 47 \\ 57 \cdot 08 &= xtime(47) = 8e \\ 57 \cdot 10 &= xtime(8e) = 07 \end{aligned}$$

y entonces:

$$57 \cdot 13 = 57 \cdot (01 \oplus 02 \oplus 10) = 57 \oplus ae \oplus 07 = fe$$

A.6.3. Doblado

El doblado de un polinomio $a(x)$ en un campo GF(p^n) se denota como $2a(x)$; consiste en duplicar los coeficientes del polinomio $a(x)$ sobre el campo GF(p^n) módulo un polinomio irreducible $m(x)$. Al escribir $2^i a(x)$ nos referimos a doblar i veces $a(x)$. Por ejemplo, podemos escribir $4a(x)$ para denotar $2^2a(x) = 2(2a(x))$; también podemos escribir $3a(x)$ para denotar $2a(x) \oplus a(x)$ y $7a(x)$ para denotar $2(2a(x)) \oplus 2a(x) \oplus a(x)$.

En el caso de esta tesis, el doblado de polinomios será utilizado sólamente en el campo GF(2^{128}). En este campo, el polinomio irreducible que se utiliza es $m(x) = x^{127}+x^7+x^2+x^1+1$ (87 en su representación hexadecimal o 10000111 en su representación binaria) y puede ser calculado con la siguiente ecuación:

$$2a(x) = \begin{cases} a(x) \ll 1, \text{si } msb_1(a(x)) = 0 \\ a(x) \ll 1 \oplus 0^{120}10000111, \text{si } msb_1(a(x)) = 1 \end{cases} \tag{A.10}$$

donde $a(x) \ll 1$ denota un corrimiento a la izquierda de 1 bit sobre a(x), $msb_1(a(x))$ representa el bit más significativo (más a la izquierda) de $a(x)$ y $0^{120}10000111$ se refiere a la cadena de bits conformada por 120 ceros concatenados con los bits 10000111, los cuales representan el polinomio irreducible $m(x)$ mencionado anteriormente.

Por ejemplo, teniendo el polinomio en GF(2^{128}) (representado en hexadecimal) $a(x)$ = 83000051000000000000000000000000, si se aplica la ecuación (A.10) sobre $a(x)$ se obtiene:

$$2a(x) = 060000a2000000000000000000000087$$

ya que el bit más significativo de 83 hexadecimal es igual a 1 (10000011 en binario); a su vez, aplicando nuevamente la ecuación (A.10) sobre $2a(x)$ se obtiene:

$$2(2a(x)) = 0c000144000000000000000000000010e$$

ya que el bit más significativo de 0c hexadecimal es igual a 0 (00001100 en binario).

A diferencia de la multiplicación, el doblado de un polinomio es relativamente fácil de implementar, ya que la instrucción que realiza el corrimiento de 1 bit sí existe en arquitecturas modernas.

APÉNDICE B

Tiempos de ejecución

En esta sección se presentan las tablas utilizadas para generar las gráficas del capítulo 7, correspondientes a los tiempos de ejecución promedio. Todos los tiempos mostrados en ellas representan la cantidad de segundos que tomó realizar la tarea descrita de acuerdo al tamaño de la entrada.

B.1. Lectura vs escritura

# Bytes	Lectura	Escritura
16 B	0.0018217759	0.0000270526
32 B	0.0006352170	0.0000521328
64 B	0.0004917034	0.0000451638
128 B	0.0003863204	0.0000439920
256 B	0.0003824610	0.0000454360
512 B	0.0003530035	0.0000479332
1 KB	0.0004626049	0.0000447694
2 KB	0.0004463181	0.0000485262
4 KB	0.0003944171	0.0001651927
8 KB	0.0004402961	0.0001848557
16 KB	0.0004686429	0.0002154093
32 KB	0.0006740378	0.0003102953
64 KB	0.0008435695	0.0005037768
128 KB	0.0019149497	0.0009525559
256 KB	0.0029452698	0.0017737539
512 KB	0.0055055710	0.0035807768
1 MB	0.0093601642	0.0077361808
2 MB	0.0192649761	0.0150997245
4 MB	0.0369845396	0.0242321803
8 MB	0.0733598122	0.0398087937
16 MB	0.1560176324	0.0673699667
32 MB	0.2786924580	0.1344738136
64 MB	0.5758245261	0.2592150429
128 MB	1.4180535832	0.7811191908
256 MB	2.9435678203	1.7606664385
512 MB	5.6690871528	3.7728671954
1 GB	11.0919633501	7.5590833728
2 GB	22.2656632831	15.1283202412
4 GB	44.8303224835	30.1455701372

Tabla B.1: Tiempos promedio de lectura y escritura de datos en disco, medidos en segundos

B.2. Subida vs bajada en GPU

	CTR		OTR		OTR optimizado	
# Bytes	**Subida**	**Bajada**	**Subida**	**Bajada**	**Subida**	**Bajada**
16 B	0.0048740741	0.0008502444	0.0000000000	0.0000000000	0.0000000000	0.0000000000
32 B	0.0045941400	0.0007825528	0.0037921778	0.0008467403	0.0033812437	0.0009011784
64 B	0.0045698872	0.0008091021	0.0044124041	0.0007862792	0.0036020759	0.0011791458
128 B	0.0043971165	0.0007998951	0.0047769519	0.0007577292	0.0038380034	0.0010230966
256 B	0.0044822339	0.0007856507	0.0045488710	0.0008079624	0.0041340942	0.0010787708
512 B	0.0043234483	0.0007951908	0.0045352003	0.0008167704	0.0040159266	0.0012551696
1 KB	0.0044012875	0.0008002975	0.0043774710	0.0007856097	0.0036938876	0.0010419840
2 KB	0.0043162326	0.0007856859	0.0042454354	0.0007845171	0.0046265761	0.0010589808
4 KB	0.0042257369	0.0007833271	0.0042751447	0.0007759186	0.0038156846	0.0011567800
8 KB	0.0050001202	0.0007837992	0.0055876246	0.0008183231	0.0045846611	0.0009518391
16 KB	0.0052466056	0.0008309501	0.0058978168	0.0008283916	0.0048212287	0.0010270609
32 KB	0.0053171158	0.0008857123	0.0059127915	0.0008561537	0.0056695073	0.0011488812
64 KB	0.0053523342	0.0008821626	0.0063339460	0.0009410556	0.0052875560	0.0009835047
128 KB	0.0053138624	0.0009569108	0.0058299223	0.0011632453	0.0059969059	0.0012883893
256 KB	0.0052257351	0.0012828998	0.0052390651	0.0016863897	0.0057586895	0.0018273697
512 KB	0.0055846041	0.0020305331	0.0065176308	0.0028743872	0.0056574775	0.0024191910
1 MB	0.0066341407	0.0033918011	0.0094452211	0.0058053395	0.0072763602	0.0042334708
2 MB	0.0080857986	0.0060005640	0.0155637599	0.0118174143	0.0089974197	0.0078011219
4 MB	0.0114340225	0.0096367995	0.0229077378	0.0195674969	0.0141397886	0.0119938558
8 MB	0.0237859520	0.0174370188	0.0353657618	0.0306976183	0.0213725157	0.0168834081
16 MB	0.0442032213	0.0279100969	0.0520139539	0.0475463755	0.0331031366	0.0292677196
32 MB	0.0625906347	0.0455823593	0.0874159319	0.0790032505	0.0386073775	0.0415170536
64 MB	0.0884791641	0.0734648217	0.1489403561	0.1415607102	0.0687643148	0.0719098789
128 MB	0.1529972492	0.1255712416	0.2613358545	0.2667892628	0.1215146609	0.1248981474
256 MB	0.2112758620	0.2156849479	0.5460105814	0.5384989428	0.2138307107	0.2287617105
512 MB	0.3872278024	0.3878525983	1.1117960312	0.7578722266	0.4345696698	0.4214573096
1 GB	0.7712736586	0.7767096951	2.2347334173	1.1918270983	0.8661906928	0.8438915982
2 GB	1.5432378652	1.5559567133	4.5170836418	2.0431068509	1.7118719654	1.6929495931
4 GB	3.0761745793	3.1042667239	9.0799870380	3.8032898445	3.4421962035	3.3780150166

Tabla B.2: Tiempos promedio de subida y bajada de datos en la GPU para cada modo de operación, medidos en segundos

B.3. CTR

# Bytes	OpenSSL	Tiempo secuencial	Tiempo paralelo	Tiempo total
16 B	0.0008226253	0.0000312995	0.0003297466	0.0003610461
32 B	0.0002198059	0.0000312160	0.0003094384	0.0003406544
64 B	0.0002185314	0.0000306497	0.0002864887	0.0003171384
128 B	0.0002192893	0.0000316911	0.0002972969	0.0003289880
256 B	0.0002346979	0.0000313416	0.0002842388	0.0003155804
512 B	0.0002210393	0.0000318249	0.0003095052	0.0003413301
1 KB	0.0002324480	0.0000315747	0.0003216466	0.0003532213
2 KB	0.0002426563	0.0000320747	0.0003301469	0.0003622216
4 KB	0.0002654308	0.0000326667	0.0003609046	0.0003935713
8 KB	0.0003324382	0.0000327746	0.0004127325	0.0004455071
16 KB	0.0003771548	0.0000317830	0.0005758111	0.0006075941
32 KB	0.0005458530	0.0000322247	0.0008886743	0.0009208990
64 KB	0.0008987831	0.0000318580	0.0014093859	0.0014412439
128 KB	0.0015752349	0.0000323413	0.0032766171	0.0033089584
256 KB	0.0030293793	0.0000332413	0.0051096987	0.0051429400
512 KB	0.0056897043	0.0000463414	0.0088999023	0.0089462437
1 MB	0.0110007040	0.0000389829	0.0175617151	0.0176006980
2 MB	0.0238554399	0.0000372161	0.0424277658	0.0424649819
4 MB	0.0437067263	0.0000480497	0.0537535520	0.0538016017
8 MB	0.0897348891	0.0000904990	0.1307114310	0.1308019300
16 MB	0.1803064860	0.0000387996	0.1670861555	0.1671249551
32 MB	0.3562536311	0.0000337163	0.2481991657	0.2482328820
64 MB	0.7183762135	0.0001078074	0.3573373782	0.3574451856
128 MB	1.4437016294	0.0000457245	0.5879030052	0.5879487297
256 MB	2.8776067877	0.0000378913	0.9760706398	0.9761085311
512 MB	5.7585724149	0.0000531326	1.8470782209	1.8471313535
1 GB	11.3126963559	0.0000930404	3.4093229109	3.4094159513
2 GB	22.6485703864	0.0000313915	6.9723014699	6.9723328614
4 GB	45.2740496518	0.0000295831	13.7889137535	13.7889433366

Tabla B.3: Tiempos promedio mínimos de cifrado del modo CTR en su versión multinúcleo (OMP) medidos en segundos, utilizando 4 hilos de ejecución

# Bytes	OpenSSL	Subida GPU	Bajada GPU	Tiempo secuencial	Tiempo paralelo	Tiempo total parcial	Tiempo total
16 B	0.0008226253	0.0052066155	0.0008270831	0.0000403999	0.0026601324	0.0027005323	0.0087342309
32 B	0.0002198059	0.0052274652	0.0008126415	0.0000595911	0.0025137076	0.0025732987	0.0086134054
64 B	0.0002185314	0.0046372905	0.0007319331	0.0000748910	0.0025612992	0.0026361902	0.0080054138
128 B	0.0002192893	0.0045533655	0.0007574746	0.0000598992	0.0024226911	0.0024825903	0.0077934304
256 B	0.0002346979	0.0054935320	0.0007952580	0.0000673327	0.0026156990	0.0026830317	0.0089718217
512 B	0.0002210393	0.0043489571	0.0007954164	0.0000763826	0.0025018661	0.0025782487	0.0077226222
1 KB	0.0002324480	0.0045597070	0.0015023995	0.0000727994	0.0025730244	0.0026458238	0.0087079303
2 KB	0.0002426563	0.0045803739	0.0007962581	0.0000700494	0.0025995577	0.0026696071	0.0080462391
4 KB	0.0002654308	0.0041337987	0.0007761914	0.0000842077	0.0025422408	0.0026264485	0.0075364386
8 KB	0.0003324382	0.0055094400	0.0011901914	0.0000644080	0.0024917245	0.0025561325	0.0092557639
16 KB	0.0003771548	0.0054838237	0.0007640665	0.0000721326	0.0024296741	0.0025018067	0.0087496969
32 KB	0.0005458530	0.0048025574	0.0007814581	0.0000677491	0.0027067827	0.0027745318	0.0083585473
64 KB	0.0008987831	0.0060002899	0.0008481749	0.0000604247	0.0033011575	0.0033615822	0.0102100470
128 KB	0.0015752349	0.0060693315	0.0009526747	0.0000637660	0.0041598404	0.0042236064	0.0112456126
256 KB	0.0030293793	0.0057077820	0.0014211079	0.0000661160	0.0063112399	0.0063773559	0.0135062458
512 KB	0.0056897043	0.0071300652	0.0028746243	0.0000583658	0.0104110723	0.0104694381	0.0204741276
1 MB	0.0110007040	0.0075962062	0.0031274743	0.0000591077	0.0175099208	0.0175690285	0.0282927090
2 MB	0.0238554399	0.0112784637	0.0057160819	0.0000830244	0.0299993086	0.0300823330	0.0470768786
4 MB	0.0437067263	0.0133959548	0.0120593386	0.0000515829	0.0640586830	0.0641102659	0.0895655593
8 MB	0.0897348891	0.0260441517	0.0195809116	0.0000514329	0.1123716536	0.1124230865	0.1580481498
16 MB	0.1803064860	0.0522308618	0.0326971589	0.0000465493	0.1372714592	0.1373180085	0.2222460292
32 MB	0.3562536311	0.0645787843	0.0503225787	0.0000453828	0.1678732475	0.1679186303	0.2828199933
64 MB	0.7183762135	0.0662915249	0.0766809058	0.0005010951	0.1900518715	0.1905529666	0.3335253973
128 MB	1.4437016294	0.1729932480	0.1323312983	0.0000587913	0.2250343689	0.2250931602	0.5304177065
256 MB	2.8776067877	0.2055221900	0.2182932422	0.0001213571	0.4035440826	0.4036654397	0.8274808719
512 MB	5.7585724149	0.3787962318	0.3736167669	0.0000495744	0.7431615051	0.7432110795	1.4956240782
1 GB	11.3126963559	0.7731887281	0.7549757958	0.0000411577	1.5247466675	1.5247878252	3.0529523491
2 GB	22.6485703864	1.5369884133	1.5478634238	0.0000443579	3.8900585510	3.8901029089	6.9749547460
4 GB	45.2740496518	3.0735962153	3.0382726193	0.0000391082	6.0684975342	6.0685366424	12.1804054770

Tabla B.4: Tiempos promedio mínimos de cifrado del modo CTR en su versión CUDA medidos en segundos, utilizando una configuración de 16 $\times$ 256

B.4. OTR

# Bytes	OTR secuencial	Tiempo secuencial	Tiempo paralelo	Tiempo total
32 B	0.0000378746	0.0000357828	0.0001712483	0.0002070311
64 B	0.0000427825	0.0000367825	0.0001666484	0.0002034309
128 B	0.0000429414	0.0000376909	0.0001623567	0.0002000476
256 B	0.0000484495	0.0000359410	0.0001599900	0.0001959310
512 B	0.0000544993	0.0000446323	0.0001640152	0.0002086475
1 KB	0.0000777992	0.0000381245	0.0001936898	0.0002318143
2 KB	0.0001031825	0.0000397997	0.0002083895	0.0002481892
4 KB	0.0001495235	0.0000481754	0.0002445143	0.0002926897
8 KB	0.0002441809	0.0000598410	0.0003312468	0.0003910878
16 KB	0.0004784786	0.0000813828	0.0005294279	0.0006108107
32 KB	0.0008764335	0.0001487812	0.0009187076	0.0010674888
64 KB	0.0016320008	0.0002291896	0.0024167265	0.0026459161
128 KB	0.0031716691	0.0003478045	0.0026310496	0.0029788541
256 KB	0.0069539156	0.0007110514	0.0051183004	0.0058293518
512 KB	0.0145327668	0.0013323540	0.0107801867	0.0121125407
1 MB	0.0315705931	0.0028148220	0.0227944533	0.0256092753
2 MB	0.0602359143	0.0069474493	0.0344421653	0.0413896146
4 MB	0.1113423754	0.0221734590	0.0629681877	0.0851416467
8 MB	0.1895687486	0.0346664877	0.1054034835	0.1400699712
16 MB	0.3100583185	0.0692002690	0.1609001603	0.2301004293
32 MB	0.5292848704	0.1216658586	0.2318018293	0.3534676879
64 MB	0.9521114258	0.2027258705	0.3388283637	0.5415542342
128 MB	1.7861992827	0.3342557995	0.5824899448	0.9167457443
256 MB	3.5834863647	0.5641171151	1.1769178147	1.7410349298
512 MB	7.0791480255	1.2462077914	2.4292335796	3.6754413710
1 GB	13.5993461357	1.7919829763	4.3805264155	6.1725093918
2 GB	26.6188003355	3.2722341908	8.4246980425	11.6969322333
4 GB	52.6277364418	6.2399369347	16.4186746974	22.6586116321

Tabla B.5: Tiempos promedio mínimos de cifrado del modo OTR en su versión multinúcleo (OMP) medidos en segundos, utilizando 4 hilos de ejecución

# Bytes	OTR secuencial	Subida GPU	Bajada GPU	Tiempo secuencial	Tiempo paralelo	Tiempo total parcial	Tiempo total
32 B	0.0000378746	0.0034983576	0.0008055001	0.0001412156	0.0013860361	0.0015272517	0.0058311094
64 B	0.0000427825	0.0053029402	0.0007866913	0.0000634158	0.0007199359	0.0007833517	0.0068729832
128 B	0.0000429414	0.0056109235	0.0007880666	0.0000657405	0.0011825177	0.0012482582	0.0076472483
256 B	0.0000484495	0.0050006153	0.0007932750	0.0000724078	0.0013450908	0.0014174986	0.0072113889
512 B	0.0000544993	0.0053408571	0.0007982164	0.0000881493	0.0017833619	0.0018715112	0.0080105847
1 KB	0.0000777992	0.0040933576	0.0008102748	0.0000666498	0.0023804066	0.0024470564	0.0073506888
2 KB	0.0001031825	0.0055107483	0.0007499832	0.0000802832	0.0023636194	0.0024439026	0.0087046341
4 KB	0.0001495235	0.0053815152	0.0008018582	0.0000794074	0.0020381489	0.0021175563	0.0083009297
8 KB	0.0002441809	0.0083475732	0.0008057749	0.0001015736	0.0023103908	0.0024119644	0.0115653125
16 KB	0.0004784786	0.0054567901	0.0008477831	0.0001096328	0.0027014283	0.0028110611	0.0091156343
32 KB	0.0008764335	0.0069735231	0.0008907829	0.0001609818	0.0037056926	0.0038666744	0.0117309804
64 KB	0.0016320008	0.0053991985	0.0010133415	0.0002343226	0.0037641645	0.0039984871	0.0104110271
128 KB	0.0031716691	0.0072441314	0.0012144415	0.0003268883	0.0070929168	0.0074198051	0.0158783780
256 KB	0.0069539156	0.0075988147	0.0017349163	0.0006172770	0.0132142935	0.0138315705	0.0231653015
512 KB	0.0145327668	0.0083571478	0.0025362077	0.0010303897	0.0232994952	0.0243298849	0.0352232404
1 MB	0.0315705931	0.0123614471	0.0054159070	0.0021541039	0.0401698893	0.0423239932	0.0601013473
2 MB	0.0602359143	0.0176856539	0.0096140477	0.0045115055	0.0936700835	0.0981815890	0.1254812906
4 MB	0.1113423754	0.0221803694	0.0125055052	0.0106182887	0.1517746034	0.1623928921	0.1970787667
8 MB	0.1895687486	0.0417795058	0.0239258191	0.0254130779	0.2322519557	0.2576650336	0.3233703585
16 MB	0.3100583185	0.0574189518	0.0352078581	0.0462129254	0.2297391524	0.2759520778	0.3685788877
32 MB	0.5292848704	0.0874171279	0.0574781187	0.0767624201	0.2694717941	0.3462342142	0.4911294608
64 MB	0.9521114258	0.1494960234	0.0954352357	0.1359563453	0.4611892223	0.5971455676	0.8420768267
128 MB	1.7861992827	0.2557105377	0.2088173375	0.2254808407	0.8058280653	1.0313089060	1.4958367812
256 MB	3.5834863647	0.5508431792	0.5004581779	0.4480658519	1.8335640904	2.2816299423	3.3329312994
512 MB	7.0791480255	1.1345381617	0.6967495918	0.6951592821	3.3185746712	4.0137339533	5.8450217068
1 GB	13.5993461357	2.2498905182	1.0225047290	1.2872863083	6.6453262798	7.9326125881	11.2050078353
2 GB	26.6188003355	4.5052197933	1.6964243412	2.4606374990	13.4836924384	15.9443299374	22.1459740719
4 GB	52.6277364418	9.0985546112	3.0834388256	4.8046531157	27.6993730815	32.5040261972	44.6860196340

Tabla B.6: Tiempos promedio mínimos de cifrado del modo OTR en su versión CUDA medidos en segundos, utilizando una configuración de 4×1024 para los tiempos totales menores registrados

# Bytes	OTR secuencial	Subida GPU	Bajada GPU	Tiempo secuencial	Tiempo paralelo	Tiempo total parcial	Tiempo total
32 B	0.0000378746	0.0054608568	0.0006934914	0.0001236155	0.0055629590	0.0056865745	0.0118409227
64 B	0.0000427825	0.0060673068	0.0007737498	0.0000720825	0.0057429718	0.0058150543	0.0126561109
128 B	0.0000429414	0.0059556984	0.0007781994	0.0001160734	0.0054296280	0.0055457014	0.0122795992
256 B	0.0000484495	0.0064540982	0.0008757750	0.0000842240	0.0057304418	0.0058146658	0.0131445390
512 B	0.0000544993	0.0077330315	0.0007908246	0.0000749656	0.0052604880	0.0053354536	0.0138593097
1 KB	0.0000777992	0.0062582232	0.0007536249	0.0000743905	0.0058138174	0.0058882079	0.0129000560
2 KB	0.0001031825	0.0061368904	0.0007978581	0.0000747075	0.0059921831	0.0060668906	0.0130016391
4 KB	0.0001495235	0.0059563982	0.0008047414	0.0000846076	0.0056305243	0.0057151319	0.0124762715
8 KB	0.0002441809	0.0059894986	0.0008118165	0.0001002486	0.0061050507	0.0062052993	0.0130066144
16 KB	0.0004784786	0.0060641232	0.0007452415	0.0001205737	0.0068164663	0.0069370400	0.0137464047
32 KB	0.0008764335	0.0061992484	0.0007753081	0.0001457155	0.0074233071	0.0075690226	0.0145435791
64 KB	0.0016320008	0.0063677486	0.0009838499	0.0002098983	0.0093813795	0.0095912778	0.0169428763
128 KB	0.0031716691	0.0054550153	0.0012002662	0.0003277971	0.0139154516	0.0142432487	0.0208985302
256 KB	0.0069539156	0.0057752237	0.0019503496	0.0005831533	0.0202678958	0.0208510491	0.0285766224
512 KB	0.0145327668	0.0087144396	0.0026167660	0.0011150980	0.0225878578	0.0237029558	0.0350341614
1 MB	0.0315705931	0.0099849141	0.0046909490	0.0022383114	0.0375862938	0.0398246052	0.0545004683
2 MB	0.0602359143	0.0174497958	0.0089728893	0.0049993019	0.0877016482	0.0927009501	0.1191236352
4 MB	0.1113423754	0.0247607023	0.0142741466	0.0138118243	0.1236906475	0.1375024718	0.1765373207
8 MB	0.1895687486	0.0368596904	0.0212433530	0.0249920409	0.1346874024	0.1596794433	0.2177824867
16 MB	0.3100583185	0.0545845110	0.0342652829	0.0445838249	0.1701069905	0.2146908154	0.3035406093
32 MB	0.5292848704	0.0872689955	0.0566765763	0.0819657617	0.2999913801	0.3819571418	0.5259027136
64 MB	0.9521114258	0.1491947293	0.1011631489	0.1414176669	0.6424907331	0.7839084000	1.0342662782
128 MB	1.7861992827	0.2653824344	0.2164008856	0.2227491593	0.8173718936	1.0401210529	1.5219043729
256 MB	3.5834863647	0.5495276630	0.5144001246	0.4473471421	1.6329385938	2.0802857359	3.1442135235
512 MB	7.0791480255	1.0970563769	0.6963150740	0.7104382344	3.6497051792	4.3601434136	6.1535148645
1 GB	13.5993461357	2.2151515484	1.0572726250	1.2906488168	6.6217418889	7.9123907057	11.1848148791
2 GB	26.6188003355	4.5202016353	1.7390795112	2.4757629439	15.9769623304	18.4527252743	24.7120064208
4 GB	52.6277364418	9.0552650452	3.1085625887	4.8547095483	27.0442744480	31.8989839963	44.0628116302

Tabla B.7: Tiempos promedio mínimos de cifrado del modo OTR en su versión CUDA medidos en segundos, utilizando una configuración de 256 × 1024 para la mejor ejecución sobre 4 GB de datos

B.5. OTR optimizado

# Bytes	OTR secuencial	Tiempo secuencial	Tiempo paralelo	Tiempo total
32 B	0.0000378746	0.0000325084	0.0002521311	0.0002846395
64 B	0.0000427825	0.0000311794	0.0002432059	0.0002743853
128 B	0.0000429414	0.0000331825	0.0002351226	0.0002683051
256 B	0.0000484495	0.0000322001	0.0002591808	0.0002913809
512 B	0.0000544993	0.0000315402	0.0002434143	0.0002749545
1 KB	0.0000777992	0.0000326764	0.0002745138	0.0003071902
2 KB	0.0001031825	0.0000331167	0.0002871969	0.0003203136
4 KB	0.0001495235	0.0000506621	0.0003325798	0.0003832419
8 KB	0.0002441809	0.0000321432	0.0004429956	0.0004751388
16 KB	0.0004784786	0.0000584926	0.0006478519	0.0007063445
32 KB	0.0008764335	0.0000328079	0.0010415229	0.0010743308
64 KB	0.0016320008	0.0000318464	0.0018392488	0.0018710952
128 KB	0.0031716691	0.0000337881	0.0035023408	0.0035361289
256 KB	0.0069539156	0.0000473878	0.0075973846	0.0076447724
512 KB	0.0145327668	0.0000382864	0.0159680112	0.0160062976
1 MB	0.0315705931	0.0000434005	0.0299557749	0.0299991754
2 MB	0.0602359143	0.0000439363	0.0424095615	0.0424534978
4 MB	0.1113423754	0.0000573145	0.0758819523	0.0759392668
8 MB	0.1895687486	0.0000477093	0.1293281985	0.1293759078
16 MB	0.3100583185	0.0000435466	0.2078346585	0.2078782051
32 MB	0.5292848704	0.0000399383	0.3024683197	0.3025082580
64 MB	0.9521114258	0.0000703509	0.4398444747	0.4399148256
128 MB	1.7861992827	0.0000461323	0.7291450220	0.7291911543
256 MB	3.5834863647	0.0000386695	1.3049516216	1.3049902911
512 MB	7.0791480255	0.0000430114	2.4281254522	2.4281684636
1 GB	13.5993461357	0.0000377576	4.7587227445	4.7587605021
2 GB	26.6188003355	0.0000335842	9.4758795880	9.4759131722
4 GB	52.6277364418	0.0000354333	19.0166539099	19.0166893432

Tabla B.8: Tiempos promedio mínimos de cifrado del modo OTR optimizado en su versión multinúcleo (OMP) medidos en segundos, utilizando 4 hilos de ejecución

# Bytes	OTR secuencial	Subida GPU	Bajada GPU	Tiempo secuencial	Tiempo paralelo	Tiempo total parcial	Tiempo total
32 B	0.0000378746	0.0041169579	0.0007069585	0.0000403997	0.0003467890	0.0003871887	0.0052111051
64 B	0.0000427825	0.0048497072	0.0009777495	0.0000521071	0.0066522928	0.0067043999	0.0125318566
128 B	0.0000429414	0.0063692063	0.0009600410	0.0000575580	0.0132388587	0.0132964167	0.0206256640
256 B	0.0000484495	0.0014462080	0.0006293335	0.0000678153	0.0135884888	0.0136563041	0.0157318456
512 B	0.0000544993	0.0036047075	0.0009285830	0.0000690066	0.0145491035	0.0146181101	0.0191514006
1 KB	0.0000777992	0.0051874984	0.0009601664	0.0000779575	0.0150260901	0.0151040476	0.0212517124
2 KB	0.0001031825	0.0025418749	0.0008862500	0.0000818492	0.0159069126	0.0159887618	0.0194168867
4 KB	0.0001495235	0.0041933323	0.0010250410	0.0001118907	0.0168834870	0.0169953777	0.0222137510
8 KB	0.0002441809	0.0049932499	0.0009777500	0.0001540402	0.0180875229	0.0182415631	0.0242125630
16 KB	0.0004784786	0.0058397481	0.0009433335	0.0001040162	0.0190566198	0.0191606360	0.0259437176
32 KB	0.0008764335	0.0058825826	0.0010198750	0.0000788646	0.0200027788	0.0200816434	0.0269841010
64 KB	0.0016320008	0.0035178733	0.0007529580	0.0001272313	0.0202291711	0.0203564024	0.0246272337
128 KB	0.0031716691	0.0366110752	0.0009477915	0.0000854001	0.0202578787	0.0203432788	0.0579021455
256 KB	0.0069539156	0.0022782070	0.0011912495	0.0000726584	0.0197723921	0.0198450505	0.0233145070
512 KB	0.0145327668	0.0045590820	0.0016482074	0.0000579988	0.0221404161	0.0221984149	0.0284057043
1 MB	0.0315705931	0.0059545818	0.0031321244	0.0001045240	0.0342437294	0.0343482534	0.0434349596
2 MB	0.0602359143	0.0078039139	0.0050271235	0.0000732323	0.0731912835	0.0732645158	0.0860955532
4 MB	0.1113423754	0.0133466222	0.0106184138	0.0000586578	0.1089362326	0.1089948904	0.1329599264
8 MB	0.1895687486	0.0354941571	0.0192804532	0.0000808910	0.1193439740	0.1194248650	0.1741994753
16 MB	0.3100583185	0.0468568216	0.0232044524	0.0000903158	0.1915256658	0.1916159816	0.2616772556
32 MB	0.5292848704	0.0351984911	0.0284007005	0.0000789327	0.2829256978	0.2830046305	0.3466038221
64 MB	0.9521114258	0.0650493149	0.0471563209	0.0000500829	0.3588751364	0.3589252193	0.4711308551
128 MB	1.7861992827	0.1040556021	0.0898268968	0.0000495160	0.6403314691	0.6403809851	0.8342634840
256 MB	3.5834863647	0.1948360353	0.1867179498	0.0000487569	1.2011270314	1.2011757883	1.5827297734
512 MB	7.0791480255	0.4204415232	0.3450973183	0.0000425662	2.1584532051	2.1584957713	2.9240346128
1 GB	13.5993461357	0.8894894421	0.7715860903	0.0000444989	4.0969233114	4.0969678103	5.7580433427
2 GB	26.6188003355	1.6217731237	1.6467088461	0.0000621832	7.9988815304	7.9989437136	11.2674256834
4 GB	52.6277364418	3.6308202743	3.1756175757	0.0000521500	16.4730578753	16.4731100253	23.2795478753

Tabla B.9: Tiempos promedio mínimos de cifrado del modo OTR optimizado en su versión CUDA medidos en segundos, utilizando una configuración de 64 $\times$ 1024

APÉNDICE C

Rendimiento

En esta sección se presentan las tablas utilizadas para generar las gráficas del capítulo 7, correspondientes a los rendimientos promedio. Todos los rendimientos mostrados en ellas representan la cantidad de gigabits por segundo que se procesaron dada una entrada.

C.1. CTR

# Bytes	OpenSSL	OMP	Parcial CUDA	CUDA
16 B	0.0005354705	0.0003301775	0.0000441429	0.0000136485
32 B	0.0010846778	0.0006998840	0.0000926510	0.0000276799
64 B	0.0021820075	0.0015035617	0.0001808812	0.0000595643
128 B	0.0043489323	0.0028988119	0.0003841449	0.0001223690
256 B	0.0081268245	0.0060439388	0.0007108931	0.0002125932
512 B	0.0172580046	0.0111759768	0.0014795692	0.0004939640
1 KB	0.0328219410	0.0215994747	0.0028835611	0.0008761433
2 KB	0.0628823116	0.0421255636	0.0057157434	0.0018963877
4 KB	0.1149737639	0.0775401512	0.0116193324	0.0040493368
8 KB	0.1835985042	0.1370015343	0.0238779313	0.0065942862
16 KB	0.3236610339	0.2009076660	0.0487928634	0.0139513761
32 KB	0.4472644192	0.2651111848	0.0879934499	0.0292084995
64 KB	0.5432692827	0.3387915467	0.1452534018	0.0478236045
128 KB	0.6199472218	0.2951268593	0.2312153187	0.0868394222
256 KB	0.6447277830	0.3797681871	0.3062593700	0.1446090223
512 KB	0.6865471023	0.4366357693	0.3731098042	0.1907895700
1 MB	0.7101818211	0.4438744418	0.4446745590	0.2761312110
2 MB	0.6549868737	0.3679502334	0.5194078531	0.3319039083
4 MB	0.7149929232	0.5808377262	0.4874414349	0.3489064351
8 MB	0.6964960967	0.4778216958	0.5559356352	0.3954491089
16 MB	0.6932640238	0.7479433573	0.9102957534	0.5624397450
32 MB	0.7017472334	1.0071187910	1.4888163365	0.8839544796
64 MB	0.6960141366	1.3988158748	2.6239423554	1.4991362099
128 MB	0.6926638993	1.7008285748	4.4426050046	1.8853065947
256 MB	0.6950219914	2.0489524846	4.9545980490	2.4169742986
512 MB	0.6946166014	2.1655200603	5.3820510893	2.6744688443
1 GB	0.7071700458	2.3464429434	5.2466316085	2.6204143024
2 GB	0.7064463552	2.2947843022	4.1130017315	2.2939216931
4 GB	0.7068066640	2.3206999419	5.2730999062	2.6271703401

Tabla C.1: Rendimientos máximos obtenidos en el modo CTR en sus versiones paralelas, utilizando 4 hilos de ejecución en la versión multinúcleo (OMP) y una configuración de 16×256 en la versión CUDA

C.2. OTR

# Bytes	OTR secuencial	OMP	Parcial CUDA	CUDA
32 B	0.0062949465	0.0011516076	0.0001561096	0.0000408873
64 B	0.0111456123	0.0023439761	0.0006087140	0.0000693785
128 B	0.0222087383	0.0047672370	0.0007640040	0.0001247082
256 B	0.0393677671	0.0097347976	0.0013455736	0.0002644912
512 B	0.0699953443	0.0182829761	0.0020382979	0.0004762071
1 KB	0.0980652054	0.0329116648	0.0031177845	0.0010379156
2 KB	0.1478815600	0.0614804716	0.0062436159	0.0017529501
4 KB	0.2040988749	0.1042659790	0.0144116962	0.0036764048
8 KB	0.2499587652	0.1560650991	0.0253051646	0.0052774325
16 KB	0.2551217808	0.1998496629	0.0434249944	0.0133913130
32 KB	0.2785614938	0.2287055611	0.0631396905	0.0208116131
64 KB	0.2991917957	0.1845414713	0.1221165000	0.0469003918
128 KB	0.3079017606	0.3278315981	0.1316156539	0.0615026610
256 KB	0.2808669406	0.3350501165	0.1412077537	0.0843125223
512 KB	0.2687891476	0.3224963364	0.1605535750	0.1108997910
1 MB	0.2474612997	0.3050652511	0.1845879703	0.1299887665
2 MB	0.2593967433	0.3775101593	0.1591438900	0.1245205554
4 MB	0.2806658282	0.3670354193	0.1924345308	0.1585660420
8 MB	0.3296956933	0.4462055604	0.2425629862	0.1932768368
16 MB	0.4031499642	0.5432410551	0.4529772017	0.3391404233
32 MB	0.4723354359	0.7072782281	0.7220545797	0.5090307545
64 MB	0.5251486186	0.9232685637	0.8373167735	0.5937700506
128 MB	0.5598479462	1.0908149901	0.9696415828	0.6685221359
256 MB	0.5581156998	1.1487420303	0.8765663366	0.6000723748
512 MB	0.5650397457	1.0883046677	0.9965782602	0.6843430531
1 GB	0.5882635768	1.2960693119	1.0084949834	0.7139664798
2 GB	0.6010789291	1.3678800288	1.0034915273	0.7224789457
4 GB	0.6080443919	1.4122665819	0.9844934226	0.7161076386

Tabla C.2: Rendimientos máximos obtenidos en el modo OTR en sus versiones paralelas, utilizando 4 hilos de ejecución en la versión multinúcleo (OMP) y una configuración de 16×1024 en la versión CUDA

C.3. OTR optimizado

# Bytes	OTR secuencial	OMP	Parcial CUDA	CUDA
32 B	0.0062949465	0.0008376159	0.0006157684	0.0000457520
64 B	0.0111456123	0.0017378378	0.0000711230	0.0000380500
128 B	0.0222087383	0.0035544398	0.0000717242	0.0000462373
256 B	0.0393677671	0.0065458945	0.0001396680	0.0001212413
512 B	0.0699953443	0.0138739219	0.0002609569	0.0001991863
1 KB	0.0980652054	0.0248360610	0.0005051225	0.0003590014
2 KB	0.1478815600	0.0476370315	0.0009543446	0.0007858515
4 KB	0.2040988749	0.0796300669	0.0017956399	0.0013738147
8 KB	0.2499587652	0.1284575291	0.0033459389	0.0025208053
16 KB	0.2551217808	0.1728197961	0.0063708904	0.0047051974
32 KB	0.2785614938	0.2272490233	0.0121574027	0.0090475731
64 KB	0.2991917957	0.2609601318	0.0239866181	0.0198268817
128 KB	0.3079017606	0.2761671103	0.0480041841	0.0168657395
256 KB	0.2808669406	0.2554850423	0.0984187468	0.0837729487
512 KB	0.2687891476	0.2440445691	0.1759697716	0.1375163931
1 MB	0.2474612997	0.2604238248	0.2274497020	0.1798666344
2 MB	0.2593967433	0.3680497676	0.2132683173	0.1814844021
4 MB	0.2806658282	0.4115130593	0.2867106879	0.2350332228
8 MB	0.3296956933	0.4830883977	0.5233416006	0.3587840887
16 MB	0.4031499642	0.6013136391	0.6523464220	0.4776876757
32 MB	0.4723354359	0.8264237203	0.8833777722	0.7212846024
64 MB	0.5251486186	1.1365836542	1.3930478359	1.0612762773
128 MB	0.5598479462	1.3713825162	1.5615704140	1.1986620764
256 MB	0.5581156998	1.5325784518	1.6650352259	1.2636395888
512 MB	0.5650397457	1.6473321600	1.8531423842	1.3679728627
1 GB	0.5882635768	1.6811100278	1.9526636211	1.3893608512
2 GB	0.6010789291	1.6884916218	2.0002641065	1.4200226786
4 GB	0.6080443919	1.6827324369	1.9425597201	1.3745971430

Tabla C.3: Rendimientos máximos obtenidos en el modo OTR optimizado en sus versiones paralelas, utilizando 4 hilos de ejecución en la versión multinúcleo (OMP) y una configuración de 2×1024 en la versión CUDA

APÉNDICE D

Aceleraciones

En esta sección se presentan las tablas correspondientes a las aceleraciones que se consiguieron con las optimizaciones hechas a los modos operación. Las aceleraciones mostradas representan la latencia S obtenida de un algoritmo pararelo respecto al algoritmo secuencial, de acuerdo al tamaño de la entrada. Estas aceleraciones se calcularon como:

$$S = \frac{T_{original}}{T_{mejorado}} \tag{D.1}$$

donde S es la aceleración real que se obtuvo, $T_{original}$ es el tiempo total del algoritmo secuencial y $T_{mejorado}$ es el tiempo total del algoritmo paralelo.

Además de las aceleraciones reales también se calcularon las aceleraciones parciales en los algoritmos muchos núcleos con CUDA. En estos cálculos no se toman en cuenta los tiempos de subida y bajada de datos en la GPU, con la intención de conocer la aceleración máxima que se obtiene en los procedimientos criptográficos. Las aceleraciones parciales se calcularon como:

$$S_{parcial} = \frac{T_{original}}{T_{parcial}} \tag{D.2}$$

donde $S_{parcial}$ es la aceleración parcial estimada y $T_{parcial}$ es el tiempo total parcial del algoritmo paralelo muchos núcleos.

D.1. CTR

# Bytes	OMP	Parcial CUDA	CUDA
16 B	0.6166118399	0.0060447661	0.0052811265
32 B	0.6452460323	0.0070268968	0.0061208503
64 B	0.6890726572	0.0069711582	0.0061247049
128 B	0.6665571389	0.0074898043	0.0064698499
256 B	0.7437023972	0.0087485376	0.0076815055
512 B	0.6475822085	0.0094039864	0.0082067064
1 KB	0.6580803593	0.0077028965	0.0066450010
2 KB	0.6699111814	0.0078427346	0.0068171745
4 KB	0.6744160461	0.0075783099	0.0067423215
8 KB	0.7462018002	0.0136039516	0.0117237612
16 KB	0.6207347965	0.0123358823	0.0105125988
32 KB	0.5927392689	0.0190077387	0.0162427209
64 KB	0.6236162387	0.0269748098	0.0232401047
128 KB	0.4760515877	0.0514207808	0.0440641907
256 KB	0.5890364850	0.0914372057	0.0774027921
512 KB	0.6359880740	0.1698264206	0.1406170248
1 MB	0.6250152125	0.3349035234	0.2700366279
2 MB	0.5617673394	0.5020086394	0.3806133561
4 MB	0.8123684968	0.4427066511	0.3544198118
8 MB	0.6860364300	0.7205480678	0.5460668630
16 MB	1.0788723078	0.7904595589	0.5877721658
32 MB	1.4351589049	1.2359798033	0.8648640233
64 MB	2.0097521031	5.2630991319	2.2851980626
128 MB	2.4554889848	5.7961823885	2.8617275354
256 MB	2.9480397886	5.6218400306	3.0897844609
512 MB	3.1175760208	7.9844938425	3.9217860305
1 GB	3.3180745669	7.3898005197	3.6786354928
2 GB	3.2483489869	7.0424876325	3.6066231928
4 GB	3.2833588874	6.5468809529	3.4638058955

Tabla D.1: Aceleraciones máximas obtenidas en el modo CTR en sus versiones paralelas, utilizando 4 hilos de ejecución en la versión multinúcleo (OMP) y una configuración de 1024×64 en la versión CUDA

D.2. OTR

# Bytes	OMP	Parcial CUDA	CUDA
32 B	0.1829415967	0.0166815903	0.0052657548
64 B	0.2103048259	0.0164798063	0.0049652430
128 B	0.2146559119	0.0180453884	0.0045931635
256 B	0.2472783786	0.0210191955	0.0056035099
512 B	0.2612027463	0.0219176607	0.0069752072
1 KB	0.3356100120	0.0317023437	0.0097578619
2 KB	0.4157412974	0.0414113537	0.0117761750
4 KB	0.5108601362	0.0604207956	0.0194901594
8 KB	0.6243633782	0.0858093420	0.0248291707
16 KB	0.7833500625	0.1458721473	0.0444177878
32 KB	0.8210236023	0.2086337804	0.0879138689
64 KB	0.6167999053	0.2647688374	0.1206996089
128 KB	1.0647279100	0.4385985988	0.2168233552
256 KB	1.1929140389	0.4876734687	0.3275601144
512 KB	1.1998115969	0.6448373411	0.4413378394
1 MB	1.2327796367	0.8038034005	0.5551021718
2 MB	1.4553388545	0.5379388591	0.4334565990
4 MB	1.3077310543	0.5778217301	0.4800507216
8 MB	1.3533860754	1.0579310458	0.7942011065
16 MB	1.3474912648	1.5003638555	1.0319712281
32 MB	1.4974066613	1.1472901463	0.8629862701
64 MB	1.7581090972	1.7357830494	1.1859597401
128 MB	1.9484129529	1.6042113770	1.1149235438
256 MB	2.0582507010	1.4796233350	1.0385272800
512 MB	1.9260674599	1.8784841790	1.2741026092
1 GB	2.2032118985	1.6557688324	1.1831958667
2 GB	2.2757078356	1.6671434605	1.1992100509
4 GB	2.3226372955	1.4778090393	1.1006279216

Tabla D.2: Aceleraciones máximas obtenidas en el modo OTR en sus versiones paralelas, utilizando 4 hilos de ejecución en la versión multinúcleo (OMP) y una configuración de 16×1024 en la versión CUDA

D.3. OTR optimizado

# Bytes	OMP	Parcial CUDA	CUDA
32 B	0.1330616446	0.0820112688	0.0074184102
64 B	0.1559212538	0.0456281157	0.0115012882
128 B	0.1600469018	0.0305875133	0.0057382958
256 B	0.1662754834	0.0311304905	0.0063159644
512 B	0.1982120678	0.0328708001	0.0111148757
1 KB	0.2532606834	0.0408106677	0.0127918653
2 KB	0.3221296255	0.0516522203	0.0205564462
4 KB	0.3901543646	0.0748565596	0.0218369929
8 KB	0.5139148813	0.1161259484	0.0394470117
16 KB	0.6774011831	0.1990831281	0.0516341922
32 KB	0.8157948185	0.2811034048	0.1211456903
64 KB	0.8722168706	0.3631972577	0.1405598493
128 KB	0.8969325468	0.4480387326	0.2725112562
256 KB	0.9096301677	0.6615179889	0.3781756666
512 KB	0.9079405596	0.9206021992	0.3838742331
1 MB	1.0523820298	0.9588767916	0.8137770432
2 MB	1.4188681127	0.8019063670	0.6518893207
4 MB	1.4662029289	0.8004457863	0.6853058267
8 MB	1.4652554082	1.0390025434	0.8645005058
16 MB	1.4915383667	1.6660804519	1.2847897017
32 MB	1.7496542868	2.5056487670	1.9440314428
64 MB	2.1643085670	2.9666858039	2.2500673578
128 MB	2.4495624668	3.1158713724	2.3124805299
256 MB	2.7459869925	3.3496640937	2.4903781921
512 MB	2.9154270520	3.4437732327	2.3922763200
1 GB	2.8577496450	3.3733067162	2.3886394220
2 GB	2.8091013343	3.2979589139	2.3341351329
4 GB	2.7674499747	3.0970004001	2.2011078915

Tabla D.3: Aceleraciones máximas obtenidas en el modo OTR optimizado en sus versiones paralelas, utilizando 4 hilos de ejecución en la versión multinúcleo (OMP) y una configuración de 2×1024 en la versión CUDA

APÉNDICE E

Resultados en servidor

En esta sección se muestran los resultados que se obtuvieron en las pruebas que se realizaron en un servidor con un microprocesador Intel I7 y una tarjeta gráfica NVIDIA Tesla C2070. Los resultados corresponden a los tiempos de ejecución promedio. Los tiempos mostrados en las tablas representan la cantidad de segundos que tomó realizar la tarea descrita para la entrada de 4 GB.

E.1. CTR

OpenSSL	Tiempo secuencial	Tiempo paralelo	Tiempo total
52.3678350000	0.0000130000	10.8601480001	10.8601610001

Tabla E.1: Tiempos promedio de cifrado del modo CTR en su versión multinúcleo (OMP) medidos en segundos, utilizando 4 hilos de ejecución en la arquitectura descrita

OpenSSL	Subida GPU	Bajada GPU	Tiempo secuencial	Tiempo paralelo	Tiempo total parcial	Tiempo total
52.3678350000	0.8758465648	0.9460543990	0.0000240000	0.6592879944	0.6593119940	2.4812129580

Tabla E.2: Tiempos promedio de cifrado del modo CTR en su versión CUDA medidos en segundos, utilizando una configuración de 1024 $\times$ 64 en la arquitectura descrita

E.2. OTR

OTR secuencial	Tiempo secuencial	Tiempo paralelo	Tiempo total
25.7685170001	4.4593859999	6.7255170001	11.1849030000

Tabla E.3: Tiempos promedio de cifrado del modo OTR en su versión multinúcleo (OMP) medidos en segundos, utilizando 4 hilos de ejecución en la arquitectura descrita

OTR secuencial	Subida GPU	Bajada GPU	Tiempo secuencial	Tiempo paralelo	Tiempo total parcial	Tiempo total
25.7685170001	3.9675476551	1.0745425224	2.5957509999	7.5877730177	10.1835240180	15.2256141950

Tabla E.4: Tiempos promedio de cifrado del modo OTR en su versión CUDA medidos en segundos, utilizando una configuración de 16 $\times$ 1024 en la arquitectura descrita

E.3. OTR optimizado

OTR secuencial	Tiempo secuencial	Tiempo paralelo	Tiempo total
25.7685170001	0.0000123999	12.9599440000	12.9599564000

Tabla E.5: Tiempos promedio de cifrado del modo OTR optimizado en su versión multinúcleo (OMP) medidos en segundos, utilizando 4 hilos de ejecución en la arquitectura descrita

OTR secuencial	Subida GPU	Bajada GPU	Tiempo secuencial	Tiempo paralelo	Tiempo total parcial	Tiempo total
25.7685170001	1.0983693600	1.1865602732	0.0000188000	0.6207427680	0.620761568	2.9056912010

Tabla E.6: Tiempos promedio de cifrado del modo OTR optimizado en su versión CUDA medidos en segundos, utilizando una configuración de 2 $\times$ 1024 en la arquitectura descrita