ESCUELA POLITÉCNICA
SUPERIOR DE CÓRDOBA
Universidad de Córdoba

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Optimización en chips de cómputo paralelo de algoritmos de ofuscación para dispositivos con recursos limitados.

Autor: D. Adolfo Fernández Gil
Directores: Dr. D. Joaquín Olivares Bueno
Dr. D. José Manuel Palomares Muñoz

noviembre, 2025

UNIVERSIDAD D CÓRDOBA

Agradecimientos

En primer lugar, deseo expresar mi agradecimiento a Joaquín Olivares Bueno y José Manuel Palomares Muñoz por su dedicación, su comprensión y el apoyo constante que me han brindado a lo largo de este proceso, más extenso y exigente de lo que imaginaba al inicio de esta aventura. Su orientación y sus consejos han sido fundamentales para poder culminar con éxito este Trabajo de Fin de Grado.

A mis amigos, gracias por todos los momentos compartidos y por recordarme, incluso en los días más complicados, la importancia de disfrutar de la vida.

A mi madre y a mi hermano, por acompañarme en cada etapa de este camino y por ofrecerme siempre su ayuda y su cariño incondicional para hacer mi día a día más llevadero.

Y, por último, a mi novia, verdadero pilar en mi vida, gracias por tu apoyo constante, por tu paciencia y por estar siempre a mi lado. Tu cariño y comprensión han sido esenciales para que pudiera avanzar y convertirme en la persona que soy hoy.

Resumen

Este trabajo presenta el diseño e implementación de un sistema digital de ofuscación y restauración de datos basado en módulos VHDL, orientado a la protección de información en sistemas embebidos. El sistema propuesto se compone de cuatro bloques funcionales —*Shuffling*, *Scrambling*, *Deshuffling* y *Descrambling*— organizados en dos parejas complementarias que ejecutan operaciones de permutación pseudoaleatoria y transformaciones no lineales reversibles. El proceso de barajado se basa en un mapa caótico controlado por parámetros de semilla y factor de realimentación, mientras que la etapa de mezcla utiliza operaciones XOR, sumas módulo 255 y sustituciones *S-box* para maximizar la difusión y la entropía de los datos.

Los módulos han sido implementados y verificados en **Xilinx ISE Design Suite 14.7**, utilizando simulaciones funcionales y estructurales para analizar la propagación de señales y la sincronización entre bloques. Los resultados experimentales demuestran la **reversibilidad total del sistema**, con una coincidencia exacta entre las matrices de entrada y salida tras aplicar las transformaciones inversas, así como una **propagación estable** de las señales de control en todo el flujo de procesamiento. El análisis de recursos muestra un consumo lógico moderado y una latencia reducida, lo que confirma la viabilidad del diseño para su implementación en dispositivos FPGA.

Estos resultados evidencian que el sistema propuesto constituye una solución ligera, determinista y completamente reversible para la ofuscación hardware de datos, combinando bajo coste computacional y elevada robustez frente a análisis predictivos.

Palabras clave: ofuscación de datos, shuffling, scrambling, mapa caótico.

Abstract

This paper presents the design and implementation of a VHDL-based digital system for data obfuscation and recovery, aimed at enhancing information protection in embedded environments. The proposed architecture consists of four functional modules —*Shuffling*, *Scrambling*, *Deshuffling*, and *Descrambling*— arranged in complementary pairs that perform pseudo-random permutation and reversible non-linear transformations. The shuffling stage relies on a chaotic map controlled by a seed and feedback factor, while the scrambling stage applies XOR operations, modular additions, and *S-box* substitutions to maximize diffusion and data entropy.

All modules were implemented and tested using **Xilinx ISE Design Suite 14.7**, combining functional and structural simulations to evaluate signal propagation and synchronization across the processing pipeline. Experimental results confirm the system's **complete reversibility**, with a perfect match between input and output matrices after applying the inverse transformations, as well as a **stable timing behavior** between control signals. The synthesis reports indicate moderate logic resource usage and low latency, confirming the design's suitability for medium-scale FPGA implementations.

Overall, the proposed architecture offers a lightweight, deterministic, and fully reversible hardware-oriented solution for data obfuscation, achieving high robustness and low computational cost within a compact and modular design.

Keywords: data obfuscation, scrambling, shuffling, chaotic map.

Índice general

Índice de figuras

Índice de tablas

Lista de Diagramas

Índice de códigos

Capítulo 1

Introducción

En los últimos años, el número de dispositivos conectados a Internet ha crecido de forma extraordinaria, dando lugar a lo que hoy se conoce como *Internet of Things* (IoT). Este conjunto de sistemas, formados por sensores, microcontroladores y nodos distribuidos, permite recopilar, procesar y transmitir información de manera continua desde el entorno físico. Según el informe más reciente de **IoT Analytics** [1], actualmente existen alrededor de **18,8 mil millones** de dispositivos IoT conectados en todo el mundo, y se espera que esa cifra se acerque a los **40 mil millones en 2030**. Esta expansión tan rápida ha generado nuevas necesidades en cuanto a **seguridad**, **consumo energético** y **eficiencia computacional**, especialmente en aquellos dispositivos con recursos muy limitados.

En el ámbito del IoT, la **seguridad de los datos** es un aspecto esencial. Los sistemas deben garantizar que la información transmitida mantenga su **confidencialidad**, su **integridad** y su **disponibilidad**. Para lograrlo, tradicionalmente se han utilizado algoritmos criptográficos simétricos o asimétricos, como AES o RSA. No obstante, este tipo de soluciones exigen una capacidad de cómputo y un consumo de energía que muchos dispositivos IoT no pueden asumir. Por ello, en los últimos años se han desarrollado métodos alternativos más ligeros que buscan proteger los datos sin sobrecargar el sistema. Entre ellos destacan las técnicas de **desordenamiento (*shuffling*) y mezcla (*scrambling*)**, que reorganizan los datos de forma pseudoaleatoria para hacerlos ilegibles a un atacante, sin necesidad de aplicar cifrado completo.

En esta línea de investigación, los trabajos de **Alcaraz, Palomares y Olivares** [2, 3] presentan un sistema de **seguridad ligera basado en el desordenamiento de bloques de datos**, combinando la verificación de integridad mediante códigos CRC16 con la permutación controlada de la información. Esta estrategia resulta

muy adecuada para entornos con recursos limitados, como las redes de sensores inalámbricas (WSN), ya que consigue un buen equilibrio entre seguridad y eficiencia energética. Sin embargo, llevar este tipo de algoritmos a hardware paralelo, como los **Field Programmable Gate Arrays (FPGAs)**, plantea un reto interesante: optimizar su implementación para aprovechar al máximo las capacidades del dispositivo sin aumentar su consumo o complejidad.

La **motivación principal** de este trabajo surge precisamente de ese reto. El objetivo es **mejorar el rendimiento de los algoritmos de ofuscación y desordenamiento de datos** cuando se implementan en **hardware reconfigurable**, aprovechando el paralelismo que ofrecen las arquitecturas FPGA. Estos dispositivos permiten ejecutar varias operaciones simultáneamente, reduciendo los tiempos de procesamiento y el consumo energético. No obstante, para que el diseño sea realmente eficiente, es necesario ajustar cuidadosamente los recursos lógicos, la estructura de control y los mecanismos de comunicación interna. En este sentido, la **optimización en chips de cómputo paralelo** se convierte en una herramienta clave para desarrollar sistemas IoT más seguros y sostenibles.

La **metodología** seguida combina el análisis teórico de las técnicas de seguridad ligera con su desarrollo e implementación práctica. En primer lugar, se realiza una revisión de los principales mecanismos de integridad y confidencialidad utilizados en redes IoT, destacando sus ventajas y limitaciones. A continuación, se diseña una arquitectura hardware que implementa el algoritmo de desordenamiento mediante una **máquina de estados finitos** capaz de gestionar las fases de inicialización, mezcla y salida de datos. Por último, se validan los resultados mediante simulaciones y pruebas sobre hardware, analizando su comportamiento y rendimiento.

El contenido del trabajo se organiza de la siguiente manera:

- En el **Capítulo 1** se presenta el contexto general del trabajo, junto con la motivación que ha impulsado su desarrollo y los fundamentos teóricos que lo enmarcan.
- El **Capítulo 2** recoge una revisión detallada del estado del arte, analizando las principales investigaciones y avances relacionados con la temática abordada.
- En el **Capítulo 3** se definen con precisión los problemas que se pretenden resolver y se formulan los objetivos generales y específicos del proyecto.
- El **Capítulo 4** expone la metodología empleada, describiendo el enfoque seguido, las herramientas utilizadas y el diseño general del estudio.

- En el **Capítulo 5** se detalla el proceso de desarrollo de los prototipos, así como la ejecución de las pruebas necesarias para su validación.
- El **Capítulo 6** presenta los resultados experimentales obtenidos y realiza un análisis exhaustivo del rendimiento del sistema y de las mejoras alcanzadas.
- Finalmente, el **Capítulo 7** reúne las conclusiones más relevantes del trabajo y propone diversas líneas de investigación y desarrollo futuro.

En resumen, este TFG se centra en la **optimización en hardware de algoritmos de seguridad ligera**, con el objetivo de ofrecer soluciones eficientes para sistemas IoT con restricciones de cómputo y energía. El desarrollo en FPGA permite aprovechar el paralelismo inherente del hardware para lograr un equilibrio entre rendimiento, consumo y seguridad, contribuyendo así a la evolución de dispositivos embebidos más rápidos, seguros y sostenibles.

Capítulo 2

Estado del Arte

Los dispositivos IoT continúan su crecimiento, lo que ha motivado mucha investigación hacia la seguridad de los datos, incluso en entornos distribuidos y con pocos recursos. A diferencia de los sistemas informáticos clásicos, los dispositivos IoT necesitan menos recursos de computación, memoria limitada y muchos menos recursos energéticos. Por lo tanto, los mecanismos de seguridad regulares, basados en algoritmos criptográficos complicados, no necesariamente funcionan para las condiciones dadas. En este capítulo se discuten la mayoría de los trabajos en el contexto de la seguridad ligera y la ofuscación de datos, incluyendo la implementación en hardware reconfigurable y dispositivos embebidos.

2.1. Seguridad IoT

La seguridad de los sistemas informáticos se basa en los 3 principios básicos de la seguridad de la información: **confidencialidad**, **integridad** y **disponibilidad**.

- *Confidencialidad* asegura la información enviada entre el usuario y el dispositivo.
- *Integridad* verifica que los datos no se corrompieron en el almacenamiento o en tránsito desde la transferencia.
- *Disponibilidad* garantiza que los servicios estén disponibles cuando se requiera algo.

Por lo tanto, los dispositivos IoT requieren soluciones diferentes que puedan considerarse soluciones de seguridad de alta calidad sin comprometer la funcionalidad y autonomía de los dispositivos. La primera línea de defensa en la protección de datos ha sido el **cifrado** (en forma simétrica (como AES, RC5 o TEA) - así como

en forma asimétrica (como RSA o ECC).

2.2. Métodos de seguridad tradicionales

Ambas técnicas son muy efectivas en confidencialidad, pero intensivas en computación, e inadecuadas para sensores de bajo consumo o microcontroladores. El cifrado con AES, por ejemplo, podría tomar miles de ciclos de reloj para leer y escribir un bloque de datos, y los algoritmos asimétricos como RSA o ECC pueden ser increíblemente costosos en tiempo y energía para realizar. Sin embargo, los **hashes criptográficos** (MD5, SHA-1, SHA-256) y los **códigos de autenticación de mensajes (MAC)** permiten validar la integridad de los datos generando resúmenes o firmas digitales. Aunque su implementación puede ser más ligera que el cifrado completo, todavía necesitan muchas operaciones aritméticas. En entornos IoT, donde la energía se mide en microjulios y los ciclos de CPU son un recurso siempre escaso, incluso ese nivel de computación puede parecer bastante grande. Finalmente, el **marcado digital**, otros métodos introducen marcas o códigos ocultos dentro del flujo de información para facilitar este cambio, y este método de detección es útil cuando se trata de aplicaciones de detección multimedia o ambiental. Sin embargo, implica redundancia y agrega peso adicional, reduciendo el número de paquetes enviados en la transmisión, aumentando así tanto el consumo de energía como el ancho de banda total disponible para la detección en tiempo real o el procesamiento de comunicaciones.

2.3. Limitaciones de las Soluciones criptográficas tradicionales

Los trabajos de Buhrow y Jongdeog [4, 5] muestran que el consumo de energía asociado con la implementación de cifrado completo y autenticación en dispositivos embebidos puede duplicar el de la transmisión no segura. Además, la gestión de claves en sistemas distribuidos es bastante complicada, aumentando las vulnerabilidades de cada nodo. Para crear consenso entre múltiples nodos, debe mantener diferentes claves privadas y compartidas en el sistema, lo que resulta en un mayor uso de memoria e introduce vulnerabilidades para ataques de compromiso de claves. Para estas limitaciones, se han introducido **mecanismos de seguridad ligeros** para proporcionar una buena cantidad de salvaguardas sin un efecto perjudicial en los recursos de hardware.

2.4. Seguridad ligera y ofuscación de datos

Las técnicas de **seguridad ligera** intentan minimizar la complejidad del procesamiento de información a través de operaciones más simples, como permutaciones, sumas modulares y desplazamientos lógicos, en lugar de cifrados complicados. No son el reemplazo de los métodos criptográficos estándar; más bien, solo son efectivos si la necesidad de la solución se basa en la inmediatez o "frescura"de la información. En estos sistemas, la recuperación de información no está completamente obstruida, sino que se vuelve inútil sin acceso en tiempo real. Algunos de los métodos más comunes son los **mecanismos de ofuscación de datos mediante barajado o mezcla controlada**. Estos métodos manipulan las posiciones de los datos en matrices o bloques de información de manera pseudoaleatoria: la información solo es conocida por el remitente y el receptor. Por lo tanto, el atacante que intercepte los paquetes no puede reconstruir el mensaje original.

2.5. Propuesta de Alcaraz, Palomares y Olivares

Alcaraz, Palomares y Olivares [2, 6] proponen como base de la **arquitectura de seguridad ligera** el barajado aleatorio de bloques de datos y el uso de códigos CRC16 para la verificación de integridad. En una matriz, se forma la información, donde los códigos de control calculados en el bloque anterior se utilizan para cada fila y columna de datos. Este enfoque crea una correlación entre mensajes consecutivos y complica la reconstrucción por fuerza bruta. Una de las ventajas clave de este enfoque es que no necesita cifrado completo, lo que a su vez minimiza significativamente el uso de energía y el tiempo de proceso. Los resultados experimentales obtenidos en los resultados de las plataformas TelosB y TinyOS demuestran que el tiempo de transmisión tiene un aumento de entre el 20 y el 30 por ciento en comparación con el 200 y el 300 por ciento para aquellos que utilizaron cifrado simétrico. Además, el enfoque es adecuado para casos de uso de datos de vida útil limitada, como tecnologías de monitoreo ambiental, automatización del hogar o agricultura inteligente.

2.6. Tendencias actuales y optimización

La tendencia principal en los estudios de seguridad ligera es hacia la **implementación en hardware reconfigurable** donde la mezcla, verificación y transmisión pueden ejecutarse en paralelo a través de la paralelización de tareas. Los **FPGAs** proporcionan la mejor configuración para tal solución con la facilidad de ajustar la arquitectura a los tamaños de bloque de datos mientras se mejora el algoritmo.

La mejora de estos algoritmos en hardware puede hacerse mediante. Los siguientes enfoques para optimizar para la heterogeneidad surgen:

- **De manera modular la separación de diseño de las etapas de mezcla y validación de procesos**,
- **La operación de máquinas de estados finitos** para controlar las transiciones de procesos.
- **La parametrización en VHDL** permite cambiar dinámicamente el número de filas y columnas, así como el número de semilla pseudoaleatoria.

No solo se puede optimizar dicho sistema para velocidades más rápidas, sino también para una mayor flexibilidad, ya que puede soportar varios escenarios y volúmenes de datos.

Capítulo 3

Formulación del Problema y Objetivos

El crecimiento de los sistemas inteligentes y la expansión del *Internet of Things* (IoT) han impulsado la aparición de millones de dispositivos conectados capaces de generar, procesar y transmitir información de forma continua. Estos dispositivos suelen operar con recursos limitados, tanto en capacidad de cómputo como en consumo energético, lo que plantea serias dificultades a la hora de aplicar técnicas de seguridad tradicionales. Los mecanismos criptográficos convencionales, aunque eficaces, exigen una carga computacional que supera las capacidades de la mayoría de los sistemas embebidos.

En este contexto surge la necesidad de diseñar **métodos alternativos de protección de la información**, que mantengan un equilibrio adecuado entre seguridad y eficiencia. El objetivo principal de este trabajo es analizar y optimizar un conjunto de técnicas de **ofuscación ligera**, concretamente los procesos de **shuffling** y **scrambling**, implementados en hardware reconfigurable. Estas técnicas permiten proteger los datos al desordenar su estructura interna y modificar su apariencia, de manera que resulten ininteligibles para un observador externo, pero sin necesidad de cifrarlos mediante algoritmos pesados.

3.1. Contexto y motivación técnica

En los sistemas IoT, la información suele transmitirse en paquetes sencillos y de tamaño reducido, que se envían de forma periódica desde nodos sensores a estaciones base o servidores. Aunque cada paquete contiene datos aparentemente triviales, su análisis conjunto permite extraer patrones de comportamiento y datos sensibles. Por

ello, la protección de la información en tránsito se convierte en un requisito básico incluso en redes de baja complejidad.

Los dispositivos utilizados en estos entornos, tales como microcontroladores, plataformas FPGA o nodos de sensado inalámbrico, deben operar con un consumo mínimo. Implementar algoritmos criptográficos complejos en este tipo de hardware puede duplicar el consumo energético y reducir drásticamente su vida útil. En consecuencia, es necesario emplear estrategias que proporcionen **seguridad suficiente con un coste computacional mínimo**.

Las técnicas de shuffling y scrambling responden a esta necesidad. El shuffling altera el orden en el que se disponen los datos dentro de una estructura determinada, mientras que el scrambling aplica un proceso de mezcla o transformación reversible que incrementa la aleatoriedad aparente de los valores. Al combinar ambas operaciones, se consigue un alto nivel de ofuscación sin recurrir a operaciones matemáticas intensivas.

3.2. Descripción general del sistema

El sistema desarrollado en este trabajo se compone de dos bloques principales:

1. **Bloque de shuffling y deshuffling:** Su función es reorganizar la disposición de los datos en una matriz o estructura interna siguiendo un patrón controlado por una semilla pseudoaleatoria. Esta semilla actúa como parámetro de sincronización entre el emisor y el receptor, de modo que ambos puedan aplicar el mismo proceso de reordenamiento e invertirlo posteriormente.

2. **Bloque de scrambling y descrambling:** Este módulo complementa al anterior aplicando una operación reversible sobre los valores de los datos. La transformación puede consistir en desplazamientos, rotaciones o intercambios simples, de manera que el resultado presente un alto grado de dispersión. El proceso inverso, descrambling, permite recuperar los valores originales utilizando la misma clave o parámetro de control.

Ambos bloques están diseñados para funcionar de forma secuencial o paralela, dependiendo de la configuración del hardware. Su implementación en FPGA permite ajustar el grado de paralelismo, el tamaño de los bloques de datos y el número de ciclos de procesamiento por operación, optimizando así el rendimiento global del sistema.

3.3. Arquitectura conceptual

La arquitectura del sistema se basa en una máquina de estados finitos que coordina las operaciones de mezcla, intercambio y control de flujo. Cada estado corresponde a una fase concreta del proceso, como la inicialización de los parámetros, la lectura de los datos, el reordenamiento y la escritura del resultado.

El diseño modular facilita la reutilización de componentes y permite escalar el sistema a diferentes tamaños de bloque sin alterar la estructura general. Cada módulo de procesamiento opera sobre un conjunto definido de datos, intercambiando información mediante señales internas. Esta arquitectura favorece la ejecución en paralelo de las operaciones de shuffling y scrambling, reduciendo la latencia y mejorando la eficiencia.

Uno de los objetivos principales de la arquitectura es minimizar el número de accesos a memoria, ya que las operaciones de lectura y escritura son las que más impacto tienen sobre el consumo energético. Por ello, las operaciones se realizan directamente sobre registros internos, reduciendo la necesidad de almacenamiento temporal y acortando el tiempo total de procesamiento.

3.4. Requisitos del sistema

Para garantizar el correcto funcionamiento del diseño en un entorno embebido y reconfigurable, el sistema debe cumplir una serie de requisitos técnicos:

- **Reversibilidad:** cada operación debe poder deshacerse de manera exacta. El proceso de deshuffling debe restaurar el orden original de los datos, y el descrambling debe recuperar sus valores iniciales sin pérdidas.
- **Determinismo:** dado un mismo conjunto de datos y parámetros, el resultado debe ser siempre idéntico. Esto permite reproducir el proceso y asegurar su fiabilidad.
- **Baja complejidad computacional:** las operaciones implementadas deben basarse en comparaciones, desplazamientos o sumas simples, evitando cálculos costosos.
- **Parametrización:** los tamaños de bloque, las semillas y las claves deben poder modificarse mediante genéricos en VHDL, lo que facilita la adaptación del diseño a distintos entornos.

- **Sincronización precisa:** los módulos deben trabajar coordinadamente para evitar errores de temporización o pérdida de datos durante el intercambio entre bloques.

Estos requisitos garantizan que el sistema sea adaptable, reproducible y eficiente desde el punto de vista del hardware.

3.5. Restricciones y limitaciones

El desarrollo en FPGA impone una serie de limitaciones físicas que deben considerarse durante la fase de diseño. Cada dispositivo dispone de un número limitado de recursos lógicos, bloques de memoria y líneas de interconexión. Por ello, la arquitectura debe optimizar el uso de estos recursos evitando redundancias innecesarias.

Asimismo, la frecuencia de reloj disponible condiciona la velocidad máxima de operación. Si el diseño incluye etapas secuenciales demasiado largas o dependencias entre operaciones, la latencia puede aumentar y comprometer el rendimiento. Por este motivo, el diseño debe equilibrar la profundidad del pipeline con el número de ciclos necesarios por operación.

Otro aspecto crítico es el consumo energético. En sistemas IoT alimentados por batería, la eficiencia energética es tan importante como la seguridad. Cualquier mejora en la rapidez o en la reducción del número de accesos a memoria repercute directamente en una menor demanda de energía y mayor autonomía del dispositivo.

Finalmente, las pruebas y simulaciones deben realizarse considerando que el sistema se integrará en entornos reales de transmisión, por lo que se prioriza la robustez frente a errores de sincronización y la estabilidad del diseño.

3.6. Evaluación del rendimiento y la complejidad

La evaluación del rendimiento del sistema se basa en el tiempo necesario para procesar un bloque de datos, el uso de recursos lógicos del FPGA y el grado de dispersión obtenido en los datos resultantes. Un buen diseño debe alcanzar un equilibrio entre estos tres aspectos.

En el caso del shuffling, la complejidad está directamente relacionada con la forma en que se generan las permutaciones y el número de intercambios requeridos. Cuantos menos accesos a memoria y operaciones de intercambio se realicen, mayor

será la eficiencia.

Por su parte, el módulo de scrambling debe mantener la simplicidad en las operaciones de mezcla, utilizando transformaciones que puedan implementarse con circuitos combinacionales o registros de desplazamiento. Las operaciones deben ser reversibles y generar un patrón de salida lo suficientemente irregular para evitar correlaciones evidentes entre la entrada y la salida.

El objetivo de la optimización es reducir al mínimo la cantidad de ciclos necesarios para completar un bloque, sin sacrificar la capacidad de recuperación ni la variabilidad de los resultados. Un diseño eficiente se caracteriza por mantener un uso estable de los recursos del dispositivo, una latencia predecible y un consumo energético reducido.

3.7. Alcance y enfoque del estudio

Este trabajo se centra en el estudio y optimización de las técnicas de shuffling y scrambling implementadas en hardware reconfigurable. No se abordan mecanismos criptográficos tradicionales, ni se incluyen módulos de verificación de integridad como CRC o funciones hash. El enfoque está dirigido exclusivamente al desarrollo de algoritmos ligeros de ofuscación reversibles, cuyo comportamiento pueda ajustarse mediante parámetros de diseño.

El proyecto se limita a analizar el comportamiento de los módulos en entornos controlados, evaluando su tiempo de ejecución, su consumo de recursos y la calidad del desordenamiento generado. La comparación entre distintas configuraciones permitirá determinar qué parámetros ofrecen un mejor compromiso entre seguridad aparente y eficiencia computacional.

El objetivo final es disponer de una arquitectura flexible, escalable y de bajo consumo que pueda integrarse en plataformas IoT o en sistemas de transmisión embebidos. Con ello se pretende contribuir al desarrollo de soluciones de seguridad ligera que aprovechen las ventajas del cómputo paralelo en hardware reconfigurable, manteniendo la simplicidad y la transparencia propias de los sistemas embebidos.

Capítulo 4

Metodología del trabajo

El desarrollo de un sistema hardware eficiente para la ofuscación de datos requiere una metodología estructurada que combine el análisis teórico con la implementación práctica y la validación experimental. En este proyecto, la metodología adoptada se ha diseñado para garantizar la correcta ejecución de cada fase del proceso, desde la concepción del algoritmo hasta la evaluación final del rendimiento del sistema implementado en FPGA.

El propósito de este capítulo es describir el enfoque metodológico seguido, detallando las herramientas empleadas, las etapas de diseño, las técnicas de optimización y los procedimientos de validación utilizados para verificar la funcionalidad del sistema.

4.1. Enfoque general

El enfoque metodológico se basa en un ciclo iterativo de **diseño, simulación y validación**, característico de los proyectos de ingeniería hardware. Este esquema permite refinar progresivamente el sistema, corrigiendo errores de diseño y mejorando su eficiencia antes de la implementación definitiva.

El proceso se estructura en tres niveles principales:

1. **Diseño conceptual:** definición de los módulos funcionales, estructura de datos y flujo de información entre los bloques de shuffling, deshuffling, scrambling y descrambling.
2. **Implementación en VHDL:** desarrollo del código fuente, descripción de las máquinas de estados y parametrización del sistema mediante genéricos.

3. **Validación experimental:** simulación del comportamiento, síntesis del diseño sobre la FPGA y análisis de los resultados obtenidos en términos de rendimiento y consumo.

Este enfoque metodológico permite integrar las consideraciones teóricas y las limitaciones prácticas del hardware en un mismo proceso de desarrollo, garantizando que el sistema final sea funcional, reproducible y eficiente.

4.2. Entorno de desarrollo y herramientas utilizadas

El proyecto se ha desarrollado utilizando el entorno **Xilinx Design Suite 14.7**, que proporciona las herramientas necesarias para la síntesis, simulación y programación de dispositivos FPGA.

El lenguaje de descripción de hardware utilizado es **VHDL**, debido a su capacidad para representar estructuras concurrentes, su portabilidad y su adecuación para diseños de complejidad media y alta. VHDL permite describir tanto el comportamiento lógico como la arquitectura estructural de cada módulo, facilitando la depuración y reutilización del código.

Las simulaciones funcionales se realizaron con el simulador integrado **ISim**, incluido en la suite de Xilinx. Este simulador permite observar las señales internas del diseño, verificar los tiempos de propagación y comprobar la secuencia de estados de las máquinas de control.

4.3. Etapas del proceso de diseño

El desarrollo del sistema se ha llevado a cabo siguiendo una metodología en fases claramente definidas:

1. **Especificación del sistema:** en esta primera etapa se definieron los objetivos funcionales y las restricciones del proyecto. Se establecieron los módulos que conformarían el sistema y las señales de entrada y salida necesarias para su interconexión.

2. **Diseño lógico y estructural:** se elaboraron las máquinas de estados finitos para cada módulo, definiendo los flujos de datos y las transiciones de estado

que regulan el proceso de ofuscación. Se diseñaron las entidades y arquitecturas de VHDL correspondientes, asegurando la modularidad del código.

3. **Simulación funcional:** una vez desarrollado el diseño lógico, se llevaron a cabo simulaciones exhaustivas para comprobar el comportamiento del sistema bajo diferentes configuraciones de semilla, tamaño de bloque y parámetros de control.

4. **Síntesis e implementación:** en esta etapa se tradujo el diseño a nivel lógico a una representación física compatible con el FPGA. Se generó el mapa de recursos, se ajustaron las restricciones de temporización y se realizaron las optimizaciones necesarias para reducir el uso de LUTs y flip-flops.

5. **Verificación post-síntesis:** se verificó que el diseño sintetizado mantuviera el comportamiento esperado y se evaluaron los tiempos de ejecución en condiciones reales de hardware.

6. **Evaluación de rendimiento:** finalmente, se analizaron los resultados obtenidos, comparando las métricas de eficiencia entre diferentes configuraciones y versiones del diseño.

Este proceso secuencial, aunque iterativo, garantiza un control completo del ciclo de vida del diseño, permitiendo detectar y corregir posibles deficiencias antes de la implementación definitiva.

4.4. Metodología de diseño hardware

El diseño del sistema se apoya en una metodología jerárquica que separa la descripción funcional del control estructural. Cada módulo —shuffling, deshuffling, scrambling y descrambling— se desarrolla de manera independiente, con interfaces bien definidas para su integración posterior.

El proceso de shuffling y deshuffling se implementa mediante una **máquina de estados finitos**, que controla las fases de inicialización, lectura, intercambio y escritura de datos. Por su parte, el módulo de scrambling utiliza **operaciones combinacionales y registros de desplazamiento**, lo que permite un tratamiento rápido de los datos sin requerir procesos secuenciales complejos.

Esta separación de responsabilidades simplifica la depuración y permite introducir mejoras en cada bloque sin afectar a los demás. Además, la modularidad del diseño posibilita su reutilización en otros sistemas o su adaptación a diferentes tamaños de matriz o parámetros de configuración.

4.5. Estrategia de validación y pruebas

Una parte fundamental de la metodología consiste en la validación del diseño. El proceso de verificación se llevó a cabo en dos fases complementarias:

1. **Validación funcional por simulación:** en esta etapa se verificó el correcto comportamiento lógico de los módulos. Se utilizaron bancos de pruebas (testbenchs) diseñados en VHDL que generaban patrones de entrada y comprobaban automáticamente las salidas esperadas.

2. **Validación temporal:** se evaluaron los tiempos de propagación y las restricciones de sincronización, comprobando que las señales cumplieran los márgenes de temporización definidos por el dispositivo.

Durante las pruebas, se recopilaron métricas de consumo de recursos (LUTs, FFs, BRAMs) y de rendimiento (frecuencia máxima, tiempo por bloque y eficiencia energética). Estos datos permitieron comparar distintas versiones del diseño y seleccionar la configuración más adecuada.

4.6. Criterios de evaluación

La evaluación del sistema se centró en los siguientes criterios:

- **Tiempo de ejecución:** mide la latencia total del sistema, es decir, el número de ciclos necesarios para procesar un bloque completo.

- **Uso de recursos:** analiza la cantidad de recursos lógicos utilizados, incluyendo LUTs, flip-flops, registros y bloques de memoria.

- **Consumo energético:** estima la energía empleada por operación, considerando la actividad interna y la frecuencia de reloj.

- **Eficiencia y escalabilidad:** evalúa cómo se comporta el sistema al modificar el tamaño del bloque o el grado de paralelismo.

- **Reversibilidad y estabilidad:** comprueba que el sistema recupere los datos originales sin errores y mantenga un comportamiento determinista en diferentes ejecuciones.

Estos parámetros proporcionan una visión global del rendimiento del diseño y permiten valorar su idoneidad para aplicaciones IoT o sistemas embebidos de bajo consumo.

4.7. Optimización y refinamiento del diseño

Durante el proceso de desarrollo se aplicaron distintas estrategias de optimización destinadas a mejorar el rendimiento sin incrementar el área utilizada. Entre las principales se destacan:

- **Reducción de lógica combinacional:** se simplificaron expresiones y se reutilizaron señales internas para minimizar la cantidad de puertas lógicas necesarias.
- **Equilibrio del pipeline:** se ajustaron los niveles de paralelismo para mejorar la frecuencia máxima alcanzable sin introducir latencias innecesarias.
- **Minimización del acceso a memoria:** se priorizó el uso de registros internos frente a la memoria externa, disminuyendo el tiempo de acceso y el consumo.
- **Parametrización flexible:** se utilizaron genéricos para controlar las dimensiones del sistema, permitiendo adaptar el diseño a diferentes aplicaciones sin modificar la estructura básica.

Capítulo 5

Desarrollo y Experimentación

5.1. Diseño de los prototipos

5.1.1. Entorno de desarrollo y herramientas

Para la implementación, validación y documentación de los módulos descritos, se ha empleado un conjunto de herramientas de software y hardware que han permitido cubrir todas las fases del desarrollo: desde la descripción en VHDL y simulación, hasta el control de versiones y la elaboración de diagramas.

5.1.1.1. Entorno de desarrollo principal

El código VHDL ha sido desarrollado íntegramente utilizando el editor **Visual Studio Code**, configurado con extensiones específicas para el soporte del lenguaje VHDL y la integración con sistemas de compilación externos. Este entorno ha permitido mantener un flujo de trabajo ágil, con resaltado de sintaxis, autocompletado y herramientas de depuración integradas, garantizando la trazabilidad y la legibilidad del código fuente.

5.1.1.2. Herramienta de síntesis y simulación

Para la síntesis, simulación y verificación de los módulos hardware se ha utilizado **Xilinx ISE Design Suite 14.7**, una plataforma completa que permite la implementación y simulación de diseños digitales descritos en VHDL. Mediante esta herramienta se ha verificado el comportamiento funcional de los módulos, comprobando la coherencia de las máquinas de estados, la correcta propagación de señales y la integridad de las operaciones aritméticas y lógicas.

5.1.1.3. Control de versiones

Durante todo el desarrollo se ha empleado **GitHub** como sistema de control de versiones distribuido. Esta herramienta ha permitido mantener un registro completo de las modificaciones, facilitar la gestión de versiones intermedias y garantizar la integridad del proyecto a lo largo del tiempo. El repositorio ha servido también como medio de respaldo y sincronización entre los distintos entornos de trabajo utilizados.

5.1.1.4. Diseño de diagramas y documentación técnica

Los diagramas de bloques, de estados y de flujo del sistema se han diseñado con la herramienta **Microsoft Visio**, que ha permitido representar de manera clara y estructurada las relaciones entre los módulos y el flujo de señales. Estos esquemas se han exportado a imagen (`.png`) para su integración directa en el documento final en LaTeX.

5.1.1.5. Hardware utilizado

El desarrollo se ha llevado a cabo empleando dos ordenadores portátiles con sistemas operativos compatibles con las herramientas mencionadas. El uso de dos equipos ha permitido disponer de entornos de prueba diferenciados: uno destinado a la edición y documentación del código, y otro enfocado a la síntesis y simulación de los módulos. Esta división de tareas ha contribuido a mantener la estabilidad del entorno principal de trabajo y a optimizar los tiempos de compilación.

5.1.2. Diseño del módulo shuffling

El módulo *Shuffling* constituye la primera etapa del sistema de transformación de datos. Su función principal es reordenar los elementos de la matriz de entrada siguiendo un patrón pseudoaleatorio determinado por una secuencia generada a partir de un mapa caótico. Esta operación de barajado inicial incrementa la dispersión de la información y prepara los datos para la etapa posterior de *scrambling* que actua sobre los mismos principios de aleatoriedad controlada.

El módulo está diseñado de manera parametrizable, permitiendo ajustar el número de filas y columnas de la matriz mediante los parámetros genéricos `filas` y `columnas`. De este modo, el sistema puede adaptarse fácilmente a distintos tamaños de datos sin alterar la arquitectura base.

5.1.2.1. Descripción funcional

El módulo recibe como entrada una matriz bidimensional `matrix_in` y produce como salida una matriz `matrix_out` que contiene los mismos elementos reorganizados. Las señales `clk` y `reset` controlan el funcionamiento síncrono del sistema, garantizando que la lógica de estado avance correctamente en cada flanco de reloj.

El proceso de barajado se desarrolla a través de una máquina de estados finita (FSM), en la que cada estado representa una fase del algoritmo. A grandes rasgos, el funcionamiento puede dividirse en las siguientes etapas:

1. **Inicialización:** se copian los elementos de la matriz de entrada a un vector unidimensional, que será el espacio sobre el que se aplicarán las transformaciones.

2. **Generación pseudoaleatoria:** se inicializa una variable `x_var` con la semilla definida por el parámetro `semilla`. A partir de ella, se genera una secuencia pseudoaleatoria utilizando un mapa caótico logístico de la forma

$$x_{n+1} = r \cdot x_n \cdot (1 - x_n),$$

   adaptado a valores enteros. Los resultados se almacenan en el array `pseudorandom_var`.

3. **Selección y ordenación de índices:** se extraen tres valores aleatorios (`a1`, `a2`, `a3`) y se ordenan de menor a mayor, de modo que definan los límites que delimitan los subbloques de la matriz.

4. **Reorganización de subbloques:** en función de otro valor pseudoaleatorio `n`, se determina el orden en el que los subbloques resultantes (`P1`, `P2`, `P3`, `P4`) se concatenan para formar el vector final `P`. Esta etapa introduce la mayor parte de la variabilidad del algoritmo.

5. **Lectura diagonal:** en función del valor psudoaleatorio `m`, se determina el orden de lectura diagonal dentro de las cuatro posibilidades de lectura:

   - **m = 0:** Diagonales de izquierda a derecha, de la esquina superior hacia la inferior.
   - **m = 1:** Diagonales de derecha a izquierda, de la esquina superior hacia la inferior.
   - **m = 2:** Diagonales de derecha a izquierda, comenzando desde la esquina inferior derecha hacia la superior.
   - **m = 3:** Diagonales de izquierda a derecha, desde la esquina inferior hacia la superior.

6. **Reconstrucción matricial:** finalmente, el vector reorganizado se distribuye nuevamente en una matriz bidimensional, que constituye la salida del módulo.

5.1.2.2. Diagrama de estados

El comportamiento descrito se implementa mediante una máquina de estados finita compuesta por diez estados: `State0` a `State9`. El estado `State0` corresponde a la fase de inicialización, mientras que los estados intermedios gestionan la generación del mapa caótico, la ordenación y la recomposición de los datos. El estado `State9` marca la finalización del proceso y la transferencia de la matriz barajada a la salida.

Cada transición se realiza de forma secuencial controlada por el reloj del sistema. El diagrama de estados correspondiente se muestra en la figura adjunta, donde se ilustran las relaciones entre los distintos estados y el flujo general del algoritmo.

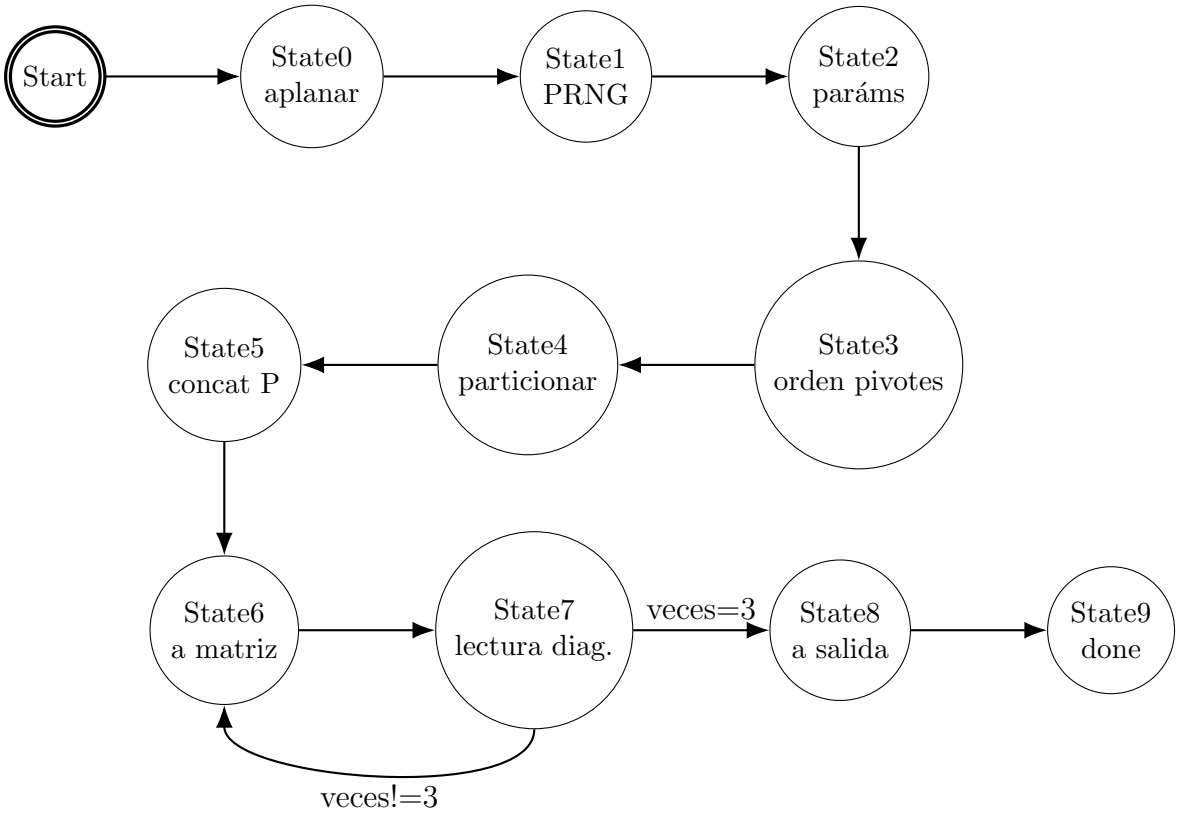

Diagrama 1: Diagrama de estados del módulo `shuffling`

5.1.2.3. Diagrama de bloques

El módulo puede representarse mediante un bloque funcional que recibe las señales `clk` y `reset_sh`, junto con la matriz de entrada `matrix_in`, y produce la salida `shuffling_out` y `done_sh`.

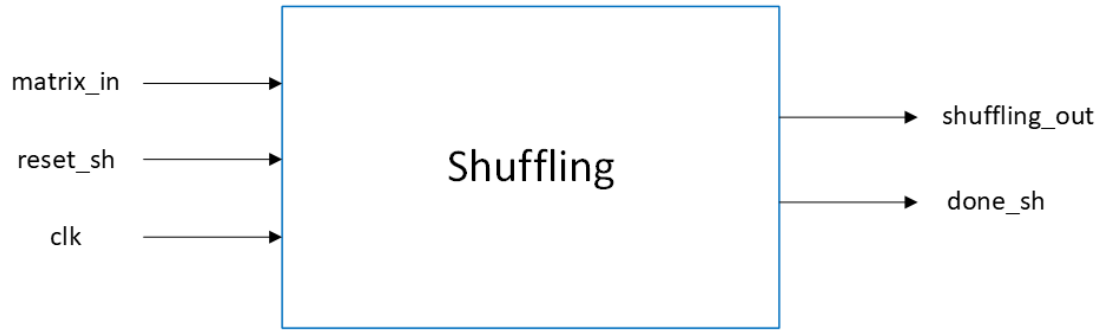

Figura 1: Diagrama de bloques del módulo `Shuffling`

5.1.3. Diseño del módulo scrambling

El módulo *Scrambling* constituye la segunda fase del sistema y se encarga de aplicar una transformación no lineal sobre la matriz obtenida tras el proceso de barajado. Su finalidad es incrementar la dispersión de la información y romper cualquier tipo de correlación entre los elementos originales, introduciendo una componente pseudoaleatoria y no lineal inspirada en principios criptográficos.

5.1.3.1. Descripción funcional

El proceso de funcionamiento puede dividirse en seis etapas principales:

1. **Inicialización:** se construye un *estado interno* a partir de las claves de entrada (`K1`, `K2`, `K3`, `K4`) y de las constantes auxiliares (`C1`, `C2`, `S1`, `S2`, `S3`). Paralelamente, la matriz de entrada se convierte en un vector unidimensional sobre el que se aplicarán las operaciones de mezcla y sustitución.

2. **Generación pseudoaleatoria:** se genera una secuencia de valores pseudoaleatorios a partir de un mapa caótico definido por los parámetros `semilla` y `r`. Esta secuencia sirve para modificar dinámicamente el estado interno y para introducir una componente de imprevisibilidad en las operaciones posteriores.

3. **Transformaciones aritméticas y lógicas:** sobre cada fragmento del vector de datos se aplican combinaciones entre el estado interno y los valores pseudoaleatorios mediante operaciones de tipo XOR y sumas módulo 255. Estas operaciones garantizan la difusión de la información, de modo que pequeñas variaciones en las claves o en los datos de entrada produzcan grandes diferencias en la salida.

4. **Actualización del estado interno:** tras cada iteración de mezcla, el estado interno se actualiza combinando sus distintas secciones con los valores generados por el mapa caótico. Este mecanismo incrementa la dependencia temporal entre iteraciones y contribuye a la complejidad global del proceso.

5. **Sustitución no lineal (*S-box*):** una vez completadas las operaciones de mezcla, cada byte del vector resultante se somete a una sustitución basada en una tabla *S-box*, que introduce una no linealidad adicional. Este paso se inspira en técnicas de cifrado por bloques y garantiza una distribución uniforme de los valores posibles.

6. **Reconstrucción matricial:** finalmente, el vector transformado se reorganiza en una matriz bidimensional de dimensiones idénticas a la original. Este resultado constituye la salida del módulo y se entrega junto con la señal de control `done_scr`, que indica la finalización del proceso.

5.1.3.2. Diagrama de estados

El comportamiento interno del módulo está gobernado por una máquina de estados finita (FSM) que coordina las etapas descritas anteriormente. El flujo general de operación se muestra en la siguiente figura:

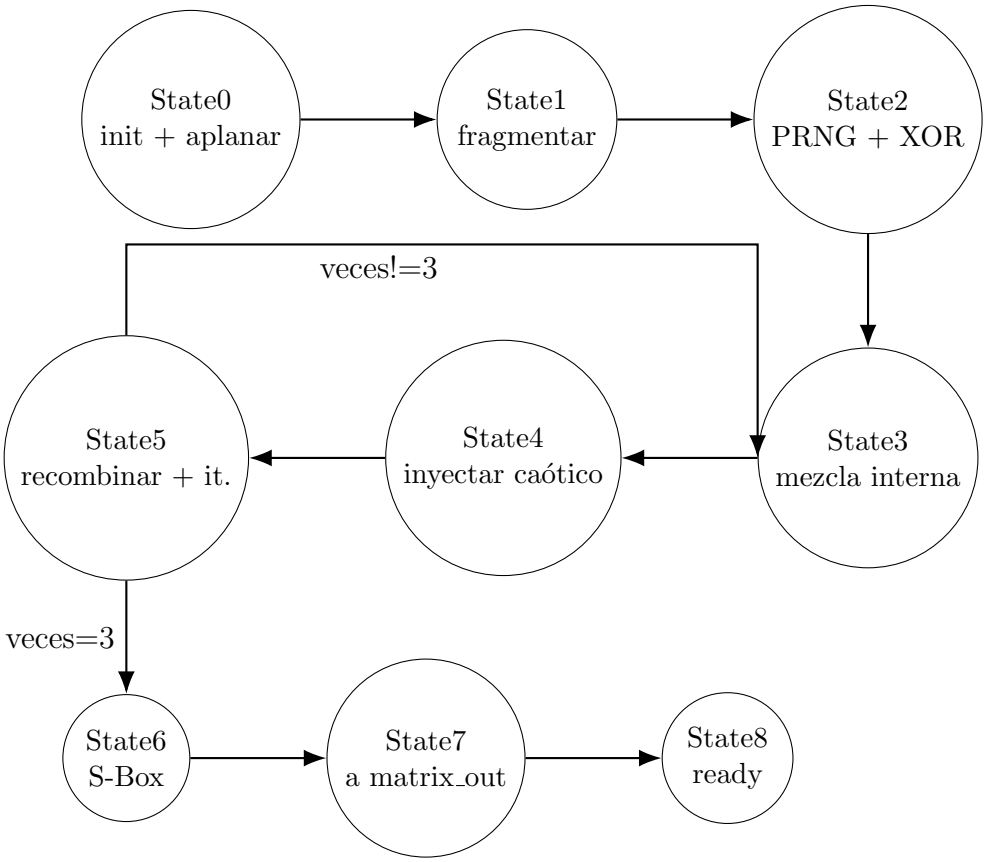

Diagrama 2: Diagrama de estados del módulo `scrambling`

5.1.3.3. Diagrama de bloques

El módulo puede representarse mediante un bloque funcional que recibe las señales `clk` y `reset_scr`, junto con la matriz de entrada `scrambling_in`, y produce la salida `scrambling_out` y `done_scr`. Además, recibe las señales `K1, K2, K3, K4, C1, C2, S1, S2, S3` encargadas de la construcción del estado interno.

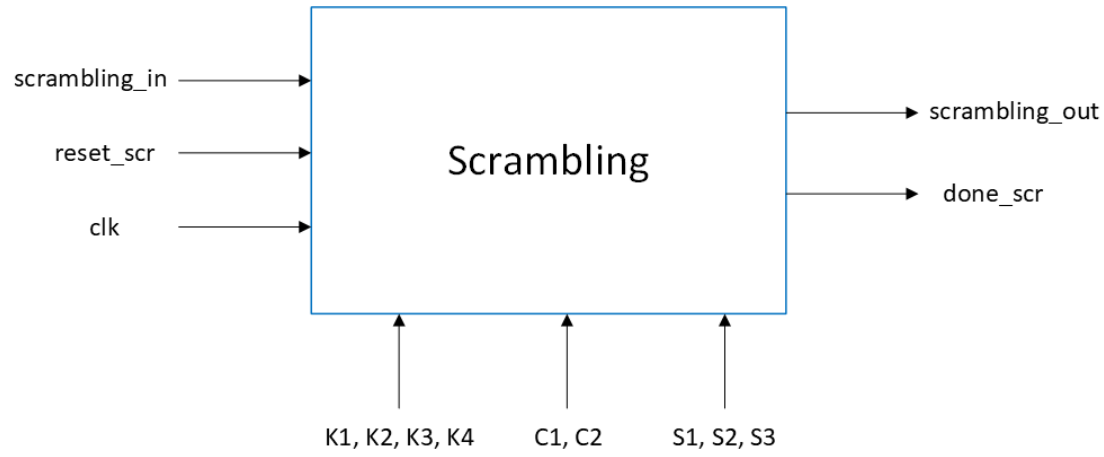

Figura 2: Diagrama de bloques del módulo `Scrambling`

5.1.4. Diseño del módulo deshuffling

El módulo *Deshuffling* tiene como propósito invertir la transformación realizada previamente por el módulo *Shuffling*. Su función principal consiste en restaurar el orden original de los datos a partir de la matriz barajada, aplicando el proceso inverso de reorganización y lectura diagonal de forma controlada y determinista.

5.1.4.1. Descripción funcional

El proceso general de funcionamiento puede dividirse en seis etapas principales:

1. **Inicialización:** se recibe la matriz de entrada `shuffling_in`, correspondiente al resultado del módulo *Shuffling*, y se copian sus elementos en un vector unidimensional. Este vector servirá como base para aplicar las operaciones de desordenamiento inverso.

2. **Generación pseudoaleatoria:** al igual que en el módulo *Shuffling*, se inicializa una variable con la semilla definida por el parámetro `semilla`, y se utiliza un mapa caótico logístico adaptado a valores enteros:

$$x_{n+1} = r \cdot x_n \cdot (1 - x_n),$$

   para generar una secuencia pseudoaleatoria. Estos valores permiten reproducir exactamente las mismas condiciones utilizadas durante el proceso de barajado, garantizando la reversibilidad del sistema.

3. **Selección y ordenación de índices:** se extraen tres valores pseudoaleatorios (`a1`, `a2`, `a3`) que definen los límites de los subbloques de datos. Dichos valores se ordenan de menor a mayor para reconstruir la misma segmentación empleada en el módulo de barajado.

4. **Recomposición de subbloques:** según el valor pseudoaleatorio `n`, se determinan las posiciones en las que deben ubicarse los distintos subbloques (`P1`, `P2`, `P3`, `P4`) para invertir el orden impuesto originalmente. Esta fase es clave para garantizar que la concatenación de los subbloques restituya el vector original antes de la lectura diagonal.

5. **Lectura diagonal inversa:** utilizando el parámetro pseudoaleatorio `m`, se aplica el recorrido diagonal correspondiente a la inversa de la operación del módulo *Shuffling*. Las posibles trayectorias se definen como:

   - **m = 0:** lectura diagonal inversa de izquierda a derecha, desde la esquina inferior hacia la superior.
   - **m = 1:** lectura diagonal inversa de derecha a izquierda, desde la esquina inferior hacia la superior.
   - **m = 2:** lectura diagonal inversa de derecha a izquierda, desde la esquina superior hacia la inferior.
   - **m = 3:** lectura diagonal inversa de izquierda a derecha, desde la esquina superior hacia la inferior.

   Esta operación asegura que los elementos vuelvan a su posición original siguiendo el patrón inverso exacto del barajado.

6. **Reconstrucción matricial:** una vez completadas las operaciones de recomposición y lectura inversa, el vector resultante se reorganiza nuevamente en una matriz bidimensional. Esta matriz, almacenada en `matrix_out`, representa la versión desbarajada o restaurada de la original.

5.1.4.2. Diagrama de estados

El control de las operaciones descritas se realiza mediante una máquina de estados finita (FSM) encargada de coordinar las fases de lectura, generación pseudoaleatoria, reorganización y reconstrucción. El flujo general del proceso se muestra en la figura siguiente:

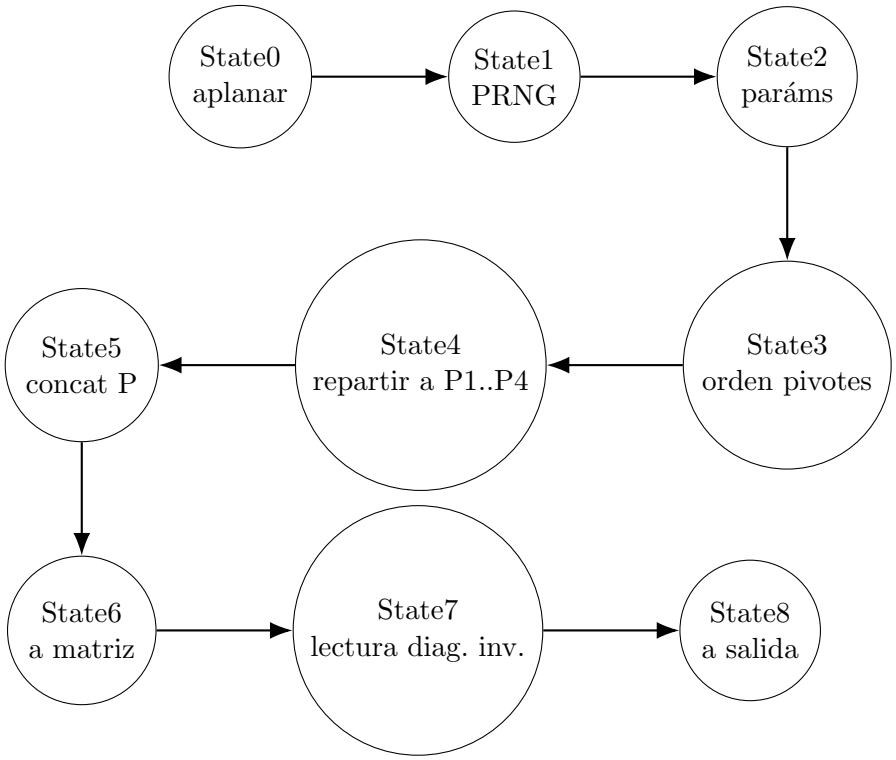

Diagrama 3: Diagrama de estados del módulo `deshuffling`

5.1.4.3. Diagrama de bloques

El módulo puede representarse mediante un bloque funcional que recibe las señales `clk` y `reset`, junto con la matriz de entrada `sehuffling_in`, y produce la salida `matrix_out` y `done_desh`.

Figura 3: Diagrama de bloques del módulo `Deshuffling`

5.1.5. Diseño del módulo descrambling

El módulo *Descrambling* constituye la etapa final del sistema y tiene como objetivo revertir la transformación no lineal aplicada por el módulo *Scrambling*. Su función es restaurar los valores originales de la matriz mediante la aplicación secuencial de las operaciones inversas, garantizando la correcta reconstrucción de la información inicial.

5.1.5.1. Descripción funcional

El proceso de funcionamiento puede dividirse en seis fases principales:

1. **Inicialización:** se configuran las señales de entrada y se genera un estado interno a partir de las claves (`K1`, `K2`, `K3`, `K4`) y de las constantes (`C1`, `C2`, `S1`, `S2`, `S3`), replicando la estructura usada en el módulo de *Scrambling*. Además, se linealiza la matriz de entrada `matrix_in` en un vector unidimensional para facilitar el procesamiento.

2. **Sustitución inversa:** cada elemento del vector de entrada se transforma utilizando la tabla inversa *Inv-S-box*, que aplica la sustitución contraria a la efectuada en el módulo anterior. Esta operación revierte la no linealidad introducida por el *Scrambling*.

3. **División en fragmentos:** el vector resultante se divide en tres fragmentos, reproduciendo la misma segmentación aplicada en la fase de cifrado. Cada fragmento se procesará de manera independiente, manteniendo la coherencia con la estructura original.

4. **Combinación inversa:** se aplican operaciones XOR entre los fragmentos y el estado interno, de manera inversa a las efectuadas durante el proceso de mezcla. Con ello, se cancelan las operaciones aritméticas y lógicas del módulo *Scrambling*, recuperando los valores originales de cada bloque.

5. **Reconstrucción matricial:** los fragmentos restaurados se concatenan nuevamente y se reorganizan en una matriz bidimensional de las mismas dimensiones que la original. Este proceso reconstruye la disposición inicial de los datos.

6. **Finalización del proceso:** una vez completada la reconstrucción, el módulo activa la señal `ready`, indicando que la matriz `matrix_out` está disponible y que la operación de descrifrado ha concluido correctamente.

5.1.5.2. Diagrama de estados

El control del proceso se lleva a cabo mediante una máquina de estados finita (FSM) encargada de coordinar las etapas de sustitución, combinación y reconstrucción. El flujo general del proceso se muestra en la figura siguiente:

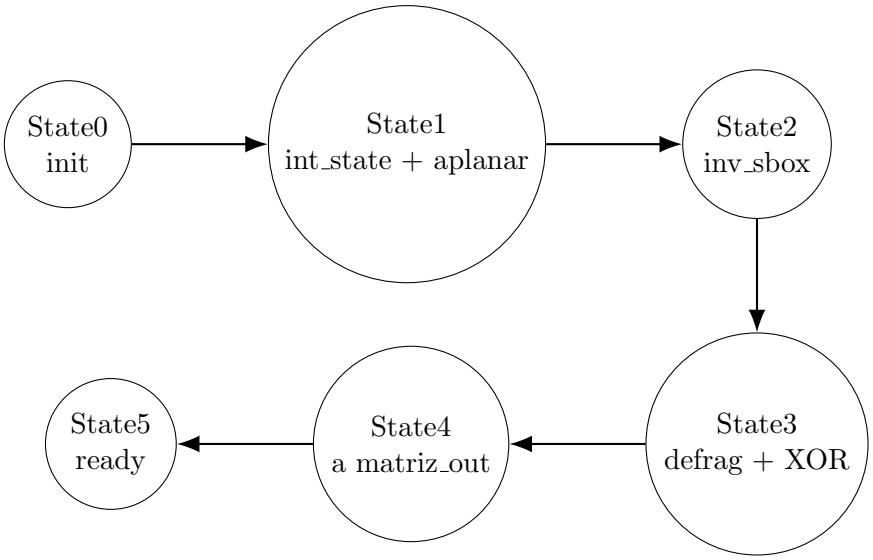

Diagrama 4: Diagrama de estados del módulo `descrambling`

5.1.5.3. Diagrama de bloques

El módulo puede representarse mediante un bloque funcional que recibe las señales `clk` y `reset_descr`, junto con la matriz de entrada `descarmbling_in`, y produce la salida `descrambling_out` y `done_descr`. Además, recibe las señales `K1, K2, K3, K4, C1, C2, S1, S2, S3` encargadas de la construcción del estado interno.

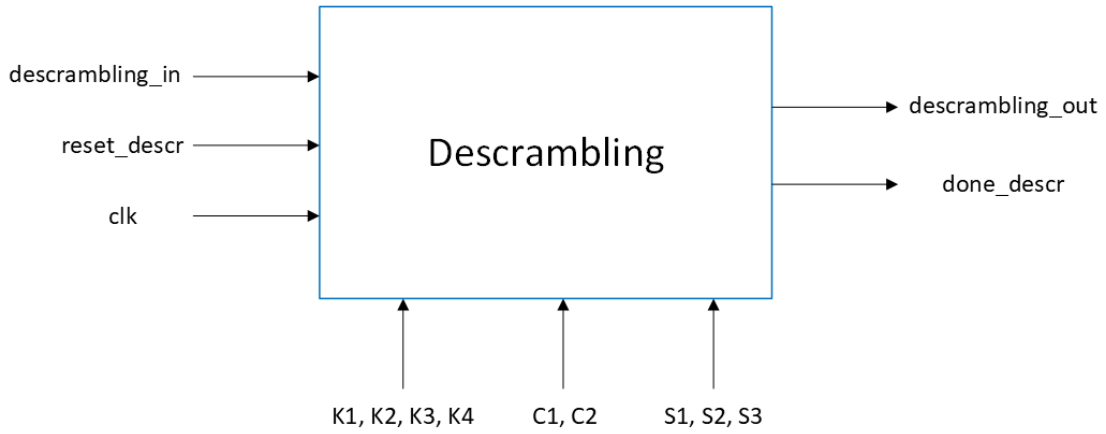

Figura 4: Diagrama de bloques del módulo `Descrambling`

5.2. Implementación de los prototipos

5.2.1. Implementación de los tipos personalizados

El paquete `tipo.vhd` define los tipos de datos y las funciones básicas utilizadas de forma recurrente en los módulos `shuffling`, `scrambling`, `deshuffling` y `descrambling`. Su objetivo principal es ofrecer una estructura tipada y modular que facilite la manipulación de matrices de bytes, coordenadas y operaciones aritmético-lógicas elementales, garantizando la consistencia entre los diferentes componentes del sistema.

En primer lugar, se declara el subtipo `valor`, definido como un entero en el rango de 0 a 255. Este subtipo representa la unidad básica de información con la que opera el sistema, equivalente a un byte, y permite expresar valores de intensidad o datos codificados en un rango compacto.

A partir de este subtipo se definen varias estructuras de datos fundamentales. El tipo `matrix` representa una matriz bidimensional de elementos de tipo `valor`, y constituye la base para la representación de imágenes o bloques de datos estructurados que serán posteriormente mezclados por el módulo `shuffling`. Por otro lado, el tipo `byte_array` define un vector unidimensional de elementos `valor`, empleado en operaciones intermedias donde se requiere manipular secuencias lineales de bytes.

El tipo `sbox_array` se define como un arreglo de 256 posiciones, cada una de ellas compuesta por un vector de 8 bits (`std_logic_vector(0 to 7)`). Este tipo

está pensado para implementar estructuras de sustitución (S-Boxes) o tablas de correspondencia utilizadas en operaciones no lineales o en transformaciones pseudoaleatorias basadas en mapas caóticos.

También se incluye la definición del tipo `coordinate`, un registro con dos campos: `row` y `col`, ambos definidos como enteros entre 0 y 9. Este tipo permite representar una coordenada dentro de una matriz de dimensiones 10x10, y se utiliza para identificar posiciones específicas durante los procesos de lectura y mezcla de los datos. A partir de este registro se construye el tipo `coordinate_array`, que agrupa 100 elementos de tipo `coordinate`, abarcando todas las posibles posiciones dentro de una matriz 10x10. Esta estructura resulta especialmente útil para almacenar y manipular trayectorias de acceso o secuencias de lectura ordenadas.

Además de los tipos de datos, el paquete `tipo` implementa tres funciones sobrecargadas para operar entre vectores de bytes: `xor_byte_array`, `sum_byte_array` y `subtract_byte_array`. Todas ellas reciben dos argumentos de tipo `byte_array` y devuelven un nuevo arreglo del mismo tipo tras aplicar la operación correspondiente elemento a elemento.

La función `xor_byte_array` realiza la operación lógica XOR entre cada par de elementos de los arreglos de entrada. Para ello convierte cada entero de 8 bits en un `std_logic_vector`, aplica la operación bit a bit y convierte el resultado de nuevo a entero. Esta función es fundamental en los procesos de mezcla pseudoaleatoria, donde la combinación no lineal de datos asegura una difusión uniforme de la información.

La función `sum_byte_array` implementa una suma modular de los elementos de ambos vectores, tomando el resultado módulo 255. De esta forma, se evitan desbordamientos en los cálculos y se garantiza que el resultado permanezca dentro del rango permitido por el subtipo `valor`. Se añade una condición de saturación para limitar el valor máximo a 255, lo que asegura la estabilidad numérica del sistema.

Por su parte, la función `subtract_byte_array` realiza la resta modular de los elementos de los dos vectores, aplicando nuevamente módulo 255 y corrigiendo los resultados negativos a cero. Esta operación permite implementar mecanismos de reversibilidad en los procesos de desmezcla (`deshuffling`), donde se requiere invertir transformaciones aritméticas aplicadas en la fase de mezcla.

En conjunto, el paquete `tipo.vhd` proporciona una base sólida para la manipulación de datos en el sistema, permitiendo trabajar con estructuras complejas de manera tipada, segura y fácilmente reutilizable. Su diseño modular favorece la cla-

ridad del código y la compatibilidad entre los diferentes módulos del proyecto.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  package tipo is
6   subtype valor is integer range 0 to 255;
7   type matrix is array(natural range <>, natural range <>) of
       valor;
8   type byte_array is array(natural range <>) of valor;
9   type sbox_array is array(0 to 255) of std_logic_vector(0 to
       7);
10  type coordinate is record
11   row : integer range 0 to 9;
12   col : integer range 0 to 9;
13  end record;
14  type coordinate_array is array (0 to 99) of coordinate;
15
16  function xor_byte_array(a, b : byte_array) return byte_array
       ;
17  function sum_byte_array(a, b : byte_array) return byte_array
       ;
18  function subtract_byte_array(a, b : byte_array) return
       byte_array;
19
20 end package tipo;
21
22 package body tipo is
23
24  function xor_byte_array(a, b : byte_array) return byte_array
       is
25
26   variable result : byte_array(a'range);
27   variable temp_a, temp_b, temp_res : std_logic_vector(7
       downto 0);
28
29  begin
30
31   if a'length /= b'length then
32    report "Error: Arrays de diferentes  t a m a os ." severity
        failure;
```

```
33   end if;
34   for i in a'range loop
35    temp_a := std_logic_vector(to_unsigned(a(i), 8));
36    temp_b := std_logic_vector(to_unsigned(b(i), 8));
37    temp_res := temp_a xor temp_b;
38    result(i) := to_integer(unsigned(temp_res));
39   end loop;
40   return result;
41
42  end function;
43
44  function sum_byte_array(a, b : byte_array) return byte_array
        is
45
46   variable result : byte_array(a'range);
47   variable temp_a, temp_b, temp_sum : integer;
48
49  begin
50
51   if a'length /= b'length then
52    report "Error: Arrays de diferentes  t a m a os ." severity
        failure;
53   end if;
54
55
56   for i in a'range loop
57
58    temp_a := a(i);
59    temp_b := b(i);
60    temp_sum := (temp_a + temp_b) mod 255;
61    if temp_sum > 255 then
62     temp_sum := 255;
63    end if;
64    result(i) := temp_sum;
65   end loop;
66
67   return result;
68  end function;
69
70  function subtract_byte_array(a, b : byte_array) return
       byte_array is
71
72   variable result : byte_array(a'range);
```

```
73    variable temp_a, temp_b, temp_diff : integer;
74
75   begin
76
77    if a'length /= b'length then
78     report "Error: Arrays de diferentes  t a m a os ." severity
          failure;
79    end if;
80
81    for i in a'range loop
82     temp_a := a(i);
83     temp_b := b(i);
84     temp_diff := (temp_a - temp_b) mod 255;
85     if temp_diff < 0 then
86      temp_diff := 0;
87     end if;
88     result(i) := temp_diff;
89    end loop;
90    return result;
91
92   end function;
93
94  end package body tipo;
```

Código 1: Tipos personalizados

5.2.2. Implementación del módulo shuffling

El módulo `shuffling` implementa la transformación principal de mezcla de datos, encargada de reordenar los elementos de una matriz de entrada siguiendo una secuencia pseudoaleatoria controlada por parámetros internos. Su propósito es dispersar la información contenida en la matriz de manera que se pierda la correspondencia directa entre posiciones originales y finales, manteniendo al mismo tiempo la posibilidad de inversión del proceso mediante el módulo `deshuffling`.

El diseño está estructurado en torno a una máquina de estados finita (FSM) que coordina de forma secuencial las distintas fases del proceso de mezcla. El módulo es completamente parametrizable mediante los genéricos `filas`, `columnas`, `semilla` y `r`. Los dos primeros definen las dimensiones de la matriz de entrada y salida, mientras que los dos últimos controlan la generación de números pseudoaleatorios utilizados durante la mezcla. La entidad del módulo incluye las señales de reloj (`clk`), reinicio (`reset`), matriz de entrada (`matrix_in`), matriz de salida (`matrix_out`) y la señal

de control (`done`), que indica la finalización del proceso.

En la arquitectura `Behavorial` se definen los estados de la máquina y un conjunto de señales y variables auxiliares que permiten manipular la información a lo largo de las distintas etapas. Se declaran los estados `Start`, `State0`, ..., `State9`, que corresponden a las fases secuenciales del algoritmo. Las señales `estado_actual` y `estado_siguiente` controlan la transición entre estados, mientras que diversas variables (`a`, `P`, `P1`, `P2`, `P3`, `P4`, `my_array`, entre otras) almacenan temporalmente los datos en formato lineal o matricial durante la ejecución del proceso.

El módulo está compuesto por dos procesos principales. El primero, de naturaleza secuencial, actualiza el estado actual en cada flanco de subida del reloj o reinicia el sistema cuando la señal `reset` se activa. El segundo proceso, de tipo combinacional, implementa la lógica de transición entre estados y la descripción completa del algoritmo de mezcla.

La operación comienza en el estado `Start`, que inicializa la máquina y da paso a `State0`. En esta fase inicial se aplanan los datos de la matriz de entrada, almacenando sus valores de manera lineal en el vector `a`. Esta conversión permite operar sobre los datos de forma unidimensional, simplificando las operaciones de particionado y permutación posteriores.

En `State1`, el módulo genera una secuencia pseudoaleatoria a partir de un mapa caótico discreto, empleando las variables `r` y `semilla` para calcular una sucesión de enteros que posteriormente se reducen módulo 256, conformando así el vector `pseudorandom_var`. A continuación, en `State2` se extraen varios valores de esta secuencia para determinar los parámetros que guiarán la mezcla: tres índices de partición (`a1`, `a2`, `a3`) y dos variables de control (`n` y `m`), que decidirán el tipo de recombinación a aplicar.

En `State3` se realiza una ordenación de los tres índices de partición, obteniendo las posiciones ordenadas `sorted_a1`, `sorted_a2` y `sorted_a3`. Este paso permite segmentar el vector de datos `a` en cuatro regiones diferenciadas, delimitadas por los pivotes seleccionados.

En `State4`, según el valor del parámetro `n`, se reorganizan las cuatro regiones generadas, almacenándolas en los vectores intermedios `P1`, `P2`, `P3` y `P4`. Cada valor posible de `n` define un patrón distinto de mezcla, con el objetivo de incrementar la variabilidad de la transformación. Los bloques resultantes se concatenan en el estado `State5` para formar el vector `P`, que contiene nuevamente todos los datos

reorganizados.

El estado `State6` reconstruye una matriz bidimensional a partir del vector lineal `P`, almacenando el resultado en la variable `Mtr`. Esta matriz intermedia se somete a un proceso de lectura diagonal en `State7`, donde se define la secuencia `read_sequence`. Esta secuencia contiene coordenadas que describen un recorrido específico de lectura a través de la matriz, configurado según el valor del parámetro `m`. Se contemplan cuatro posibles patrones de lectura diagonal, cada uno con una dirección o simetría distinta. A partir de esta secuencia se genera el vector `my_array`, que contiene los elementos de la matriz reordenados de acuerdo con el recorrido definido.

Tras cada iteración de lectura diagonal, los datos se copian nuevamente en `P` y el proceso se repite un número determinado de veces, controlado por la variable `veces`. Cuando el número de iteraciones alcanza el valor prefijado, el sistema avanza hacia el estado `State8`.

En `State8`, el vector resultante `my_array` se vuelca en la matriz de salida `matrix_out`, reconstituyendo el formato bidimensional original pero con los elementos ya mezclados. Finalmente, en el estado `State9` se activa la señal `done`, indicando que el proceso de mezcla ha concluido y que la salida está disponible para ser utilizada por el siguiente módulo del sistema.

El diseño completo del módulo `shuffling` garantiza que la mezcla de datos sea determinista para una configuración dada de parámetros, de modo que, conociendo la misma semilla y el mismo conjunto de constantes, el proceso de mezcla pueda reproducirse e invertirse de manera exacta. Su implementación modular y su estructura basada en una máquina de estados permiten mantener una clara separación entre las fases de inicialización, generación pseudoaleatoria, particionado, recombinación, lectura y finalización, asegurando así un comportamiento predecible y verificable en simulación.

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4 use ieee.math_real.all;
5 use work.tipo.all;
6
7
8 entity shuffling is
9 generic(
```

```
 filas : integer := 10;
 columnas : integer := 10;
 semilla : integer := 500000;
 r : integer := 3800000
);
port(
 clk : in std_logic;
 reset : in std_logic;
 matrix_in : in matrix(0 to filas-1, 0 to columnas-1);
 matrix_out : out matrix(0 to filas-1, 0 to columnas-1);
 done : out std_logic
);
end entity shuffling;

architecture Behavorial of shuffling is
 type Estados is (Start, State0, State1, State2, State3,
    State4, State5, State6, State7, State8, State9);
 signal estado_actual, estado_siguiente : Estados := State0;
 signal row, col : integer :=0;
begin
process(clk, reset)
begin
 if clk'event and clk='1' then
  if reset='1' then
   estado_actual<=Start;
  else
   estado_actual<=estado_siguiente;
  end if;
 end if;
end process;

process(estado_actual)
 variable x_var : integer := semilla;
 variable pseudorandom_var : byte_array(0 to 255);
 variable a1, a2, a3, n, m, prueba : integer :=0;
 variable P, P1, P2, P3, P4, aux1, aux2, aux3, aux4, a,
    my_array : byte_array(0 to filas*columnas-1) := (others
    =>0);
 variable index, index1, i : integer :=0;
 variable sorted_a1, sorted_a2, sorted_a3 : integer :=0;
 variable Mtr : matrix(0 to filas-1, 0 to columnas-1);
 variable read_sequence : coordinate_array;
 variable veces : integer :=0;
```

```
50 begin
51  case estado_actual is
52  when Start=>
53   estado_siguiente<=State0;
54  when State0 =>
55   index:=0;
56   veces:=0;
57   done<='0';
58   for i in 0 to filas-1 loop
59    for j in 0 to columnas-1 loop
60     a(index) := matrix_in(i, j);
61     index := index + 1;
62    end loop;
63   end loop;
64   estado_siguiente<=State1;
65  when State1 =>
66   for k in 0 to 255 loop
67    x_var := (r * x_var * (1000000 - x_var))/1000000;
68    pseudorandom_var(k):= x_var mod 256;
69   end loop;
70   estado_siguiente<=State2;
71  when State2=>
72   a1 := pseudorandom_var(8) mod filas*columnas;
73   a2 := pseudorandom_var(9) mod filas*columnas;
74   a3 := pseudorandom_var(10) mod filas*columnas;
75   n := pseudorandom_var(90) mod 4;
76   m := pseudorandom_var(91) mod 4;
77   estado_siguiente<=State3;
78  when State3 =>
79   if a1 <= a2 and a1 <= a3 then
80    sorted_a1 := a1;
81    if a2 <= a3 then
82     sorted_a2 := a2;
83     sorted_a3 := a3;
84    else
85     sorted_a2 := a3;
86     sorted_a3 := a2;
87    end if;
88   elsif a2 <= a1 and a2<= a3 then
89    sorted_a1 := a2;
90     if a1 <= a3 then
91      sorted_a2 := a1;
92      sorted_a3 := a3;
```

```
93      else
94       sorted_a2 := a3;
95       sorted_a3 := a1;
96      end if;
97     else
98      sorted_a1 := a3;
99      if a1 <= a2 then
100      sorted_a2 := a1;
101      sorted_a3 := a2;
102     else
103      sorted_a2 := a2;
104      sorted_a3 := a1;
105     end if;
106    end if;
107    estado_siguiente<=State4;
108  when State4 =>
109   case n is
110   when 0 =>
111    index:=0;
112    for i in 0 to filas*columnas-1 loop
113     if i >= sorted_a3 and i<=filas*columnas-1 then
114      P1(index):=a(i);
115      index:=index+1;
116         end if;
117    end loop;
118    index:=0;
119    for i in 0 to filas*columnas-1 loop
120         if i< sorted_a1 then
121          P2(index):=a(i);
122          index:=index+1;
123         end if;
124    end loop;
125    index:=0;
126    for i in 0 to filas*columnas-1 loop
127         if i>=sorted_a1 and i<sorted_a2 then
128          P3(index):=a(i);
129          index:=index+1;
130         end if;
131    end loop;
132    index:=0;
133    for i in 0 to filas*columnas-1 loop
134         if  i>=sorted_a2 and i<sorted_a3  then
135          P4(index):=a(i);
```

```
136           index:=index+1;
137          end if;
138     end loop;
139    when 1 =>
140     index:=0;
141     for i in 0 to filas*columnas-1 loop
142          if i >= sorted_a1 and i<sorted_a2 then
143           P1(index):=a(i);
144           index:=index+1;
145          end if;
146     end loop;
147     index:=0;
148     for i in 0 to filas*columnas-1 loop
149          if i< sorted_a1 then
150           P2(index):=a(i);
151           index:=index+1;
152          end if;
153     end loop;
154     index:=0;
155     for i in 0 to filas*columnas-1 loop
156          if i>=sorted_a3 and i<=filas*columnas-1 then
157       P3(index):=a(i);
158           index:=index+1;
159          end if;
160     end loop;
161     index:=0;
162     for i in 0 to filas*columnas-1 loop
163          if i>=sorted_a2 and i<sorted_a3 then
164           P4(index):=a(i);
165           index:=index+1;
166          end if;
167     end loop;
168    when 2 =>
169     index:=0;
170     for i in 0 to filas*columnas-1 loop
171          if i >= sorted_a2 and i<sorted_a3 then
172           P1(index):=a(i);
173           index:=index+1;
174          end if;
175     end loop;
176     index:=0;
177     for i in 0 to filas*columnas-1 loop
178          if i< sorted_a1 then
```

```
179           P2(index):=a(i);
180           index:=index+1;
181          end if;
182     end loop;
183     index:=0;
184     for i in 0 to filas*columnas-1 loop
185          if i>=sorted_a3 and i<=filas*columnas-1 then
186           P3(index):=a(i);
187           index:=index+1;
188          end if;
189     end loop;
190     index:=0;
191     for i in 0 to filas*columnas-1 loop
192          if i>=sorted_a1 and i<sorted_a2 then
193           P4(index):=a(i);
194           index:=index+1;
195          end if;
196     end loop;
197    when 3 =>
198     index:=0;
199     for i in 0 to filas*columnas-1 loop
200          if i >= sorted_a3 and i<=filas*columnas-1 then
201           P1(index):=a(i);
202           index:=index+1;
203          end if;
204     end loop;
205     index:=0;
206     for i in 0 to filas*columnas-1 loop
207          if i >= sorted_a2 and i<sorted_a3 then

208           P2(index):=a(i);
209           index:=index+1;
210          end if;
211     end loop;
212     index:=0;
213     for i in 0 to filas*columnas-1 loop
214          if i>=sorted_a1 and i<sorted_a2 then
215           P3(index):=a(i);
216           index:=index+1;
217          end if;
218     end loop;
219     index:=0;
220     for i in 0 to filas*columnas-1 loop
```

```
221          if i<sorted_a1 then
222           P4(index):=a(i);
223           index:=index+1;
224          end if;
225    end loop;
226   when others =>
227   end case;
228   estado_siguiente<=State5;
229  when State5 =>
230   i:=0;
231   index:=0;
232   while P1(index) /=0 and index <99 loop
233    P(i) := P1(index);
234    index:=index+1;
235    i:=i+1;
236   end loop;
237   index:=0;
238   while P2(index) /=0 and index <99 loop
239    P(i) := P2(index);
240    i:=i+1;
241    index:=index+1;
242   end loop;
243   index:=0;
244   while P3(index) /=0 and index <99 loop
245    P(i) := P3(index);
246    i:=i+1;
247    index:=index+1;
248   end loop;
249   index:=0;
250   while P4(index) /=0 and index <99 loop
251    P(i) := P4(index);
252    i:=i+1;
253    index:=index+1;
254   end loop;
255   estado_siguiente<=State6;
256  when State6 =>
257   index:=0;
258   for i in 0 to filas-1 loop
259    for j in 0 to columnas-1 loop
260     Mtr(i,j):=P(index);
261          index:=index+1;
262    end loop;
263   end loop;
```

```
264   estado_siguiente<=State7;
265
266  when State7 =>
267
268   case m is
269   when 0 =>
270    read_sequence:=((0,9),
271                    (0,8),(1,9),
272                    (0,7),(1,8),(2,9),
273                    (0,6),(1,7),(2,8),(3,9),
274                    (0,5),(1,6),(2,7),(3,8),(4,9),
275                    (0,4),(1,5),(2,6),(3,7),(4,8),(5,9),
276                    (0,3),(1,4),(2,5),(3,6),(4,7),(5,8),(6,9),
277                    (0,2),(1,3),(2,4),(3,5),(4,6),(5,7),(6,8)
                          ,(7,9),
278                    (0,1),(1,2),(2,3),(3,4),(4,5),(5,6),(6,7)
                          ,(7,8),(8,9),
279                    (0,0),(1,1),(2,2),(3,3),(4,4),(5,5),(6,6)
                          ,(7,7),(8,8),(9,9),
280                    (1,0),(2,1),(3,2),(4,3),(5,4),(6,5),(7,6)
                          ,(8,7),(9,8),
281                    (2,0),(3,1),(4,2),(5,3),(6,4),(7,5),(8,6)
                          ,(9,7),
282                    (3,0),(4,1),(5,2),(6,3),(7,4),(8,5),(9,6),
283                    (4,0),(5,1),(6,2),(7,3),(8,4),(9,5),
284                    (5,0),(6,1),(7,2),(8,3),(9,4),
285                    (6,0),(7,1),(8,2),(9,3),
286                    (7,0),(8,1),(9,2),
287                    (8,0),(9,1),
288                    (9,0));
289
290   when 1 =>
291    read_sequence:=((9,0),
292                    (9,1),(8,0),
293                    (9,2),(8,1),(7,0),
294                    (9,3),(8,2),(7,1),(6,0),
295                    (9,4),(8,3),(7,2),(6,1),(5,0),
296                    (9,5),(8,4),(7,3),(6,2),(5,1),(4,0),
297                    (9,6),(8,5),(7,4),(6,3),(5,2),(4,1),(3,0),
298                    (9,7),(8,6),(7,5),(6,4),(5,3),(4,2),(3,1)
                          ,(2,0),
299                    (9,8),(8,7),(7,6),(6,5),(5,4),(4,3),(3,2)
                          ,(2,1),(1,0),
```

```
300                     (9,9),(8,8),(7,7),(6,6),(5,5),(4,4),(3,3)
                            ,(2,2),(1,1),(0,0),
301                     (8,9),(7,8),(6,7),(5,6),(4,5),(3,4),(2,3)
                            ,(1,2),(0,1),
302                     (7,9),(6,8),(5,7),(4,6),(3,5),(2,4),(1,3)
                            ,(0,2),
303                     (6,9),(5,8),(4,7),(3,6),(2,5),(1,4),(0,3),
304                     (5,9),(4,8),(3,7),(2,6),(1,5),(0,4),
305                     (4,9),(3,8),(2,7),(1,6),(0,5),
306                     (3,9),(2,8),(1,7),(0,6),
307                     (2,9),(1,8),(0,7),
308                     (1,9),(0,8),
309                     (0,9));
310
311   when 2 =>
312    read_sequence:=((9,9),
313                     (8,9),(9,8),
314                     (7,9),(8,8),(9,7),
315                     (6,9),(7,8),(8,7),(9,6),
316                     (5,9),(6,8),(7,7),(8,6),(9,5),
317                     (4,9),(5,8),(6,7),(7,6),(8,5),(9,4),
318                     (3,9),(4,8),(5,7),(6,6),(7,5),(8,4),(9,3),
319                     (2,9),(3,8),(4,7),(5,6),(6,5),(7,4),(8,3)
                            ,(9,2),
320                     (1,9),(2,8),(3,7),(4,6),(5,5),(6,4),(7,3)
                            ,(8,2),(9,1),
321                     (0,9),(1,8),(2,7),(3,6),(4,5),(5,4),(6,3)
                            ,(7,2),(8,1),(9,0),
322                     (0,8),(1,7),(2,6),(3,5),(4,4),(5,3),(6,2)
                            ,(7,1),(8,0),
323                     (0,7),(1,6),(2,5),(3,4),(4,3),(5,2),(6,1)
                            ,(7,0),
324                     (0,6),(1,5),(2,4),(3,3),(4,2),(5,1),(6,0),
325                     (0,5),(1,4),(2,3),(3,2),(4,1),(5,0),
326                     (0,4),(1,3),(2,2),(3,1),(4,0),
327                     (0,3),(1,2),(2,1),(3,0),
328                     (0,2),(1,1),(2,0),
329                     (0,1),(1,0),
330                     (0,0));
331
332   when 3 =>
333    read_sequence:=((0,0),
334                     (1,0),(0,1),
```

```
335                     (2,0),(1,1),(0,2),
336                     (3,0),(2,1),(1,2),(0,3),
337                     (4,0),(3,1),(2,2),(1,3),(0,4),
338                     (5,0),(4,1),(3,2),(2,3),(1,4),(0,5),
339                     (6,0),(5,1),(4,2),(3,3),(2,4),(1,5),(0,6),
340                     (7,0),(6,1),(5,2),(4,3),(3,4),(2,5),(1,6)
                           ,(0,7),
341                     (8,0),(7,1),(6,2),(5,3),(4,4),(3,5),(2,6)
                           ,(1,7),(0,8),
342                     (9,0),(8,1),(7,2),(6,3),(5,4),(4,5),(3,6)
                           ,(2,7),(1,8),(0,9),
343                     (9,1),(8,2),(7,3),(6,4),(5,5),(4,6),(3,7)
                           ,(2,8),(1,9),
344                     (9,2),(8,3),(7,4),(6,5),(5,6),(4,7),(3,8)
                           ,(2,9),
345                     (9,3),(8,4),(7,5),(6,6),(5,7),(4,8),(3,9),
346                     (9,4),(8,5),(7,6),(6,7),(5,8),(4,9),
347                     (9,5),(8,6),(7,7),(6,8),(5,9),
348                     (9,6),(8,7),(7,8),(6,9),
349                     (9,7),(8,8),(7,9),
350                     (9,8),(8,9),
351                     (9,9));
352
353   when others =>
354
355   end case;
356
357   for t in 0 to filas*columnas-1 loop
358    my_array(t) := Mtr(read_sequence(t).row, read_sequence(t).
          col);
359   end loop;
360   for w in 0 to filas*columnas-1 loop
361    P(w):=my_array(w);
362   end loop;
363   veces:=veces+1;
364   if veces = 3 then
365    estado_siguiente<=State8;
366   else
367    estado_siguiente<=State6;
368   end if;
369  when State8 =>
370   index:=0;
371   for i in 0 to filas-1 loop
```

```
372     for j in 0 to columnas-1 loop
373      matrix_out(i,j)<=my_array(index);
374          index:=index+1;
375     end loop;
376    end loop;
377    estado_siguiente<=State9;
378   when State9 =>
379    done<='1';
380   end case;
381 end process;
382 end architecture Behavorial;
```

Código 2: Módulo shuffling

5.2.3. Implementación del módulo scrambling

El módulo `scrambling` implementa una etapa de dispersión adicional sobre los datos previamente procesados, cuyo propósito es aumentar la complejidad estructural de la matriz y reforzar la resistencia del sistema frente a correlaciones o patrones residuales. Esta operación actúa como una capa de cifrado interno, en la que los valores de entrada son modificados mediante una combinación de transformaciones aritméticas, operaciones lógicas y sustituciones no lineales.

El módulo está definido de forma genérica, admitiendo la parametrización de las dimensiones de la matriz de entrada mediante los genéricos `filas` y `columnas`, y de la generación pseudoaleatoria mediante los parámetros `semilla` y `r`. En la entidad se definen múltiples puertos de entrada: las claves `K1`, `K2`, `K3` y `K4`, los coeficientes `C1` y `C2`, y las secuencias de estado `S1`, `S2` y `S3`, todos ellos representados como vectores de tipo `byte_array(0 to 3)`. Estos valores constituyen el conjunto de parámetros de control que gobiernan el proceso de mezcla interna. Además, se definen la matriz de entrada (`matrix_in`), la matriz de salida (`matrix_out`), las señales de reloj y reinicio (`clk`, `reset`), y la señal de control `scrambled_ready`, que indica la finalización del proceso.

En la arquitectura `Behavorial` se define una máquina de estados finita con nueve estados (`State0` a `State8`) que controla la secuencia de operaciones. Asimismo, se declara una constante `sbox` de tipo `sbox_array`, que contiene una tabla de sustitución fija utilizada para realizar transformaciones no lineales sobre los datos, similar a las empleadas en algoritmos de cifrado como AES. La presencia de la `sbox` introduce una relación no lineal entre los datos de entrada y salida, mejorando la difusión y evitando que pequeños cambios en la entrada produzcan resultados predecibles.

El proceso comienza en el estado `State0`, donde se inicializan las variables internas y se carga el vector de estado `internal_state` con las claves y secuencias de entrada. Este vector de 36 elementos se forma concatenando las claves (`K1`, `K2`, `K3`, `K4`), los coeficientes (`C1`, `C2`) y los vectores de estado (`S1`, `S2`, `S3`), de modo que cada grupo de cuatro bytes ocupa una región consecutiva del espacio interno. Durante esta fase, también se aplanan los valores de la matriz de entrada en el vector `a`, permitiendo tratar los datos en una estructura unidimensional.

En el estado `State1`, el vector `a` se divide en tres fragmentos de 36, 36 y 28 elementos respectivamente (`fragmento`, `fragmento1` y `fragmento2`), completando el último fragmento con ceros hasta alcanzar la longitud de 36 posiciones. Esta división permite realizar las operaciones posteriores de manera paralela sobre bloques de datos de tamaño uniforme.

A continuación, en el estado `State2`, se genera una secuencia pseudoaleatoria a partir del mapa caótico controlado por los parámetros `r` y `semilla`. La variable `x_var` se actualiza iterativamente en un bucle de 256 pasos, produciendo una secuencia de enteros que se almacenan en `pseudorandom_var`. A partir de cuatro posiciones específicas de esta secuencia se extraen los valores `a1`, `a2`, `a3` y `a4`, que se utilizan para inicializar el vector `chaotic`. Estos valores actúan como semillas dinámicas que modifican el comportamiento interno del proceso, garantizando variabilidad en cada ejecución.

En esta misma fase se aplican operaciones lógicas entre el estado interno y los fragmentos de datos. Se calculan los vectores `resultado`, `resultado1` y `resultado2` mediante la función `xor_byte_array`, combinando cada fragmento con el vector `internal_state`. Esta operación introduce una dependencia directa entre los datos de entrada y el estado del sistema, dificultando la reversión directa de la transformación sin conocer los parámetros utilizados.

El estado `State3` ejecuta una serie de combinaciones aritméticas entre grupos de cuatro bytes dentro del vector `internal_state`. Estas combinaciones utilizan operaciones de suma y XOR mediante las funciones `sum_byte_array` y `xor_byte_array`. Los grupos se procesan de manera independiente en bloques de tres, actualizando sus valores en cascada de forma que cada bloque influye parcialmente en los demás. Este procedimiento incrementa la complejidad interna y refuerza la interdependencia entre las distintas secciones del estado.

En `State4`, el vector `internal_state` se modifica nuevamente mediante la suma

de los valores pseudoaleatorios contenidos en el vector `chaotic`. En este punto, tres subgrupos del estado (los comprendidos entre las posiciones 0–3, 16–19 y 32–35) son actualizados sumando los valores de `chaotic(0 to 3)`, con el fin de inyectar perturbaciones adicionales en el sistema y evitar la aparición de patrones repetitivos.

Durante el estado `State5`, los fragmentos procesados se recombinan en el vector `a`, reemplazando los valores originales con los contenidos en `resultado`, `resultado1` y `resultado2`. El proceso se repite un número fijo de veces controlado por la variable `veces`, que actúa como contador de iteraciones internas. Una vez alcanzado el número máximo de repeticiones, el sistema avanza al estado `State6`.

En este punto se aplica la tabla de sustitución `sbox` a cada elemento del vector `a`, generando el vector `scrambled`. Esta operación introduce una transformación no lineal que produce una dispersión significativa de los valores originales. El resultado se almacena en el vector `scrambled`, que constituye la versión final transformada de los datos.

En el estado `State7`, los valores de `scrambled` se vuelcan secuencialmente sobre la matriz de salida `matrix_out`, reconstruyendo su estructura bidimensional original. Finalmente, en el estado `State8`, se activa la señal `scrambled_ready`, indicando que el proceso de mezcla ha finalizado correctamente y que los datos están listos para ser utilizados en etapas posteriores del sistema.

En conjunto, el módulo `scrambling` representa una capa adicional de transformación no lineal y pseudoaleatoria aplicada a los datos. Su diseño combina operaciones aritméticas modulares, lógica XOR, permutaciones parciales y sustituciones mediante `sbox`, logrando una alta difusión y un comportamiento determinista condicionado por las claves y parámetros de entrada. La estructura basada en una máquina de estados garantiza la reproducibilidad y control temporal del proceso, manteniendo la coherencia con el resto de módulos del sistema de procesamiento.

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4 use work.tipo.all;
5
6 entity scrambling is
7 generic(
8  filas : integer := 10;
9  columnas : integer := 10;
```

```
 semilla : integer := 500000;
 r : integer := 3800000
);
port(
 clk : in std_logic;
 reset : in std_logic;
 K1, K2, K3, K4 : in byte_array(0 to 3);
 C1, C2 : in byte_array(0 to 3);
 S1, S2, S3 : in byte_array(0 to 3);
 matrix_in : in matrix(0 to filas-1, 0 to columnas-1);
 matrix_out : out matrix(0 to filas-1, 0 to columnas-1);
 scrambled_ready : out std_logic
);
end entity scrambling;

architecture Behavorial of scrambling is
 type Estados is (State0, State1, State2, State3, State4,
    State5, State6, State7, State8);
 signal estado_actual, estado_siguiente : Estados;
 constant sbox: sbox_array := (
 x"63", x"7C", x"77", x"7B", x"F2", x"6B", x"6F", x"C5",
 x"30", x"01", x"67", x"2B", x"FE", x"D7", x"AB", x"76",
 x"CA", x"82", x"C9", x"7D", x"FA", x"59", x"47", x"F0",
 x"AD", x"D4", x"A2", x"AF", x"9C", x"A4", x"72", x"C0",
 x"B7", x"FD", x"93", x"26", x"36", x"3F", x"F7", x"CC",
 x"34", x"A5", x"E5", x"F1", x"71", x"D8", x"31", x"15",
 x"04", x"C7", x"23", x"C3", x"18", x"96", x"05", x"9A",
 x"07", x"12", x"80", x"E2", x"EB", x"27", x"B2", x"75",
 x"09", x"83", x"2C", x"1A", x"1B", x"6E", x"5A", x"A0",
 x"52", x"3B", x"D6", x"B3", x"29", x"E3", x"2F", x"84",
 x"53", x"D1", x"00", x"ED", x"20", x"FC", x"B1", x"5B",
 x"6A", x"CB", x"BE", x"39", x"4A", x"4C", x"58", x"CF",
 x"D0", x"EF", x"AA", x"FB", x"43", x"4D", x"33", x"85",
 x"45", x"F9", x"02", x"7F", x"50", x"3C", x"9F", x"A8",
 x"51", x"A3", x"40", x"8F", x"92", x"9D", x"38", x"F5",
 x"BC", x"B6", x"DA", x"21", x"10", x"FF", x"F3", x"D2",
 x"CD", x"0C", x"13", x"EC", x"5F", x"97", x"44", x"17",
 x"C4", x"A7", x"7E", x"3D", x"64", x"5D", x"19", x"73",
 x"60", x"81", x"4F", x"DC", x"22", x"2A", x"90", x"88",
 x"46", x"EE", x"B8", x"14", x"DE", x"5E", x"0B", x"DB",
 x"E0", x"32", x"3A", x"0A", x"49", x"06", x"24", x"5C",
 x"C2", x"D3", x"AC", x"62", x"91", x"95", x"E4", x"79",
 x"E7", x"C8", x"37", x"6D", x"8D", x"D5", x"4E", x"A9",
```

```
 x"6C", x"56", x"F4", x"EA", x"65", x"7A", x"AE", x"08",
 x"BA", x"78", x"25", x"2E", x"1C", x"A6", x"B4", x"C6",
 x"E8", x"DD", x"74", x"1F", x"4B", x"BD", x"8B", x"8A",
 x"70", x"3E", x"B5", x"66", x"48", x"03", x"F6", x"0E",
 x"61", x"35", x"57", x"B9", x"86", x"C1", x"1D", x"9E",
 x"E1", x"F8", x"98", x"11", x"69", x"D9", x"8E", x"94",
 x"9B", x"1E", x"87", x"E9", x"CE", x"55", x"28", x"DF",
 x"8C", x"A1", x"89", x"0D", x"BF", x"E6", x"42", x"68",
 x"41", x"99", x"2D", x"0F", x"B0", x"54", x"BB", x"16"
 );
begin
 process(clk, reset)
 begin
  if clk'event and clk='1' then
   if reset='1' then
    estado_actual <= State0;
   else
    estado_actual <= estado_siguiente;
   end if;
  end if;
 end process;

 process(estado_actual)
  variable x_var : integer := semilla;
  variable pseudorandom_var : byte_array(0 to 255);
  variable fragmento, fragmento1, fragmento2 : byte_array(0
     to 35);
  variable internal_state : byte_array(0 to 35);
  variable resultado, resultado1, resultado2 : byte_array(0
     to 35);
  variable a : byte_array(0 to filas*columnas-1);
  variable scrambled : byte_array(0 to filas*columnas-1);
  variable index : integer := 0;
  variable chaotic : byte_array(0 to filas*columnas-1);
  variable a1, a2, a3, a4 : integer;
  variable aux0, aux1, aux2 : byte_array(0 to 3);
  variable veces : integer;
 begin

  case estado_actual is
  when State0 =>

   scrambled_ready <= '0';
```

```
93    veces := 0;
94    index:=0;
95    internal_state(0 to 3) := K1(0 to 3);
96    internal_state(4 to 7) := S1(0 to 3);
97    internal_state(8 to 11) := S2(0 to 3);
98    internal_state(12 to 15) := S3(0 to 3);
99    internal_state(16 to 19) := K2(0 to 3);
100   internal_state(20 to 23) := K4(0 to 3);
101   internal_state(24 to 27) := C1(0 to 3);
102   internal_state(28 to 31) := C2(0 to 3);
103   internal_state(32 to 35) := K3(0 to 3);
104
105   for i in 0 to filas-1 loop
106    for j in 0 to columnas-1 loop
107     a(index) := matrix_in(i, j);
108     index := index + 1;
109    end loop;
110
111   end loop;
112
113   estado_siguiente <= State1;
114
115  when State1 =>
116
117   fragmento(0 to 35) := a(0 to 35);
118   fragmento1(0 to 35) := a(36 to 71);
119   fragmento2(0 to 27) := a(72 to 99);
120   fragmento2(28 to 35) := (others => 0);
121
122   estado_siguiente <= State2;
123
124  when State2 =>
125
126   for k in 0 to 255 loop
127    x_var := (r * x_var * (1000000 - x_var))/1000000;
128    pseudorandom_var(k):= x_var mod 256;
129   end loop;
130
131   a1 := pseudorandom_var(92);
132   a2 := pseudorandom_var(228);
133   a3 := pseudorandom_var(154);
134   a4 := pseudorandom_var(183);
135
```

```
136     chaotic(0) := a1 mod 255;
137     chaotic(1) := a2 mod 255;
138     chaotic(2) := a3 mod 255;
139     chaotic(3) := a4 mod 255;
140
141     resultado := xor_byte_array(internal_state,fragmento);
142     resultado1 := xor_byte_array(internal_state,fragmento1);
143     resultado2 := xor_byte_array(internal_state,fragmento2);
144
145     estado_siguiente <= State3;
146
147   when State3 =>
148
149     aux0 := internal_state(0 to 3);
150     aux1 := internal_state(4 to 7);
151     aux2 := internal_state(8 to 11);
152
153     aux2 := xor_byte_array(aux2,(sum_byte_array(aux0,aux1)));
154     aux1 := xor_byte_array(aux1,(sum_byte_array(aux2,aux0)));
155     aux0 := xor_byte_array(aux0,(sum_byte_array(aux1,aux2)));
156
157     internal_state(0 to 3) := aux0;
158     internal_state(4 to 7) := aux1;
159     internal_state(8 to 11) := aux2;
160
161
162     aux0 := internal_state(12 to 15);
163     aux1 := internal_state(16 to 19);
164     aux2 := internal_state(20 to 23);
165
166     aux2 := xor_byte_array(aux2,(sum_byte_array(aux1,aux0)));
167     aux1 := xor_byte_array(aux1,(sum_byte_array(aux2,aux0)));
168     aux0 := xor_byte_array(aux0,(sum_byte_array(aux1,aux2)));
169
170     internal_state(12 to 15) := aux0;
171     internal_state(16 to 19) := aux1;
172     internal_state(20 to 23) := aux2;
173
174
175     aux0 := internal_state(24 to 27);
176     aux1 := internal_state(28 to 31);
177     aux2 := internal_state(32 to 35);
178
```

```
179     aux2 := xor_byte_array(aux2,(sum_byte_array(aux1,aux0)));
180     aux1 := xor_byte_array(aux1,(sum_byte_array(aux2,aux0)));
181     aux0 := xor_byte_array(aux0,(sum_byte_array(aux1,aux2)));
182
183     internal_state(24 to 27) := aux0;
184     internal_state(28 to 31) := aux1;
185     internal_state(32 to 35) := aux2;
186
187     estado_siguiente <= State4;
188
189   when State4 =>
190
191     aux0 := internal_state(0 to 3);
192     aux1 := internal_state(16 to 19) ;
193     aux2 := internal_state(32 to 35);
194
195     aux0 := sum_byte_array(aux0,chaotic(0 to 3));
196     aux1 := sum_byte_array(aux1,chaotic(0 to 3));
197     aux2 := sum_byte_array(aux2,chaotic(0 to 3));
198
199     internal_state(0 to 3) := aux0;
200     internal_state(16 to 19) := aux1;
201     internal_state(32 to 35) := aux2;
202
203     estado_siguiente <= State5;
204
205   when State5 =>
206
207     a(0 to 35) := resultado(0 to 35);
208     a(36 to 71) := resultado1(0 to 35);
209     a(72 to 99) := resultado2(0 to 27);
210     veces := veces + 1;
211
212     if veces = 3 then
213       estado_siguiente <= State6;
214     else
215       estado_siguiente <= State3;
216     end if;
217
218   when State6 =>
219
220     for i in 0 to 99 loop
221       scrambled(i) := to_integer(unsigned(sbox(a(i))));
```

```
222     end loop;
223
224     estado_siguiente <= State7;
225
226   when State7 =>
227
228     index := 0;
229     for f in 0 to filas-1 loop
230       for c in 0 to columnas-1 loop
231         matrix_out(f, c) <= scrambled(index);
232         index := index + 1;
233       end loop;
234     end loop;
235
236     estado_siguiente <= State8;
237
238   when State8 =>
239     scrambled_ready <= '1';
240   end case;
241 end process;
242 end Behavorial;
```

Código 3: Módulo scrambling

5.2.4. Implementación del módulo deshuffling

El módulo de desmezcla implementa la transformación inversa de lectura y recombinación sobre una matriz de bytes, con el objetivo de revertir la reordenación aplicada en la fase de mezcla. Trabaja sobre una matriz de entrada y produce una matriz de salida de las mismas dimensiones, controlado por una máquina de estados finita (FSM) que organiza el proceso en etapas bien delimitadas. El diseño es parametrizable mediante los genéricos `filas`, `columnas`, `semilla` y `r`, lo que permite ajustar el tamaño de los datos y fijar la dinámica de la secuencia pseudoaleatoria utilizada durante la desmezcla.

La interfaz del módulo incluye las señales de reloj y reinicio (`clk`, `reset`), la matriz de entrada (`matrix_in`) y la matriz de salida (`matrix_out`). La arquitectura `Behavorial` define una FSM con estados `State0` a `State8`, y un conjunto de señales y variables auxiliares para manipular datos en formato lineal y matricial durante el proceso. Entre estas variables destacan los vectores intermedios `a`, `P`, `P1`, `P2`, `P3`, `P4` y `my_array`, así como la matriz temporal `Mtr` y la secuencia de coordenadas

`read_sequence` que gobierna los recorridos de lectura.

El flujo de operación se inicia en `State0`, donde se realiza el aplanado de la matriz de entrada: los elementos de `matrix_in` se recorren por filas y se almacenan secuencialmente en el vector lineal `a`. Esta representación unidimensional facilita el posterior particionado y la recombinación por bloques.

En `State1` se genera una secuencia pseudoaleatoria a partir de un mapa discreto controlado por `r` y `semilla`. La variable `x_var` se itera en un bucle y de cada estado se deriva un byte que se almacena en `pseudorandom_var`. A continuación, en `State2` se extraen de esa secuencia tres índices de partición (`a1`, `a2`, `a3`) y dos selectores (`n` y `m`). Los índices determinan los cortes sobre el vector `a`, mientras que los selectores escogen la variante de recombinación por bloques y el patrón de recorrido diagonal.

En `State3` se ordenan los índices de partición para obtener `sorted_a1`, `sorted_a2` y `sorted_a3`. De este modo, el vector `a` queda conceptualmente dividido en cuatro segmentos contiguos delimitados por dichos pivotes, lo que prepara el particionado explícito de la etapa siguiente.

El estado `State4` ejecuta el particionado y la reordenación por bloques. Según el valor del selector `n`, los cuatro segmentos resultantes se extraen del vector `a` y se asignan a los vectores intermedios `P1`, `P2`, `P3` y `P4` siguiendo un patrón específico. Cada variante define un orden de extracción distinto en función de la posición relativa de los cortes, con el fin de reconstruir un orden lineal que revierta la mezcla previa a nivel de bloques.

Una vez preparados los bloques, `State5` concatena secuencialmente `P1`, `P2`, `P3` y `P4` en el vector `P`, que vuelve a contener todos los elementos tras la recombinación. Posteriormente, `State6` reconstituye la estructura bidimensional volcando `P` por filas en la matriz temporal `Mtr`.

La etapa `State7` aplica la lectura mediante una secuencia de coordenadas que recorre la matriz `Mtr` siguiendo patrones diagonales predefinidos. Existen cuatro variantes de recorrido, seleccionadas por el parámetro `m`, que describen ordenaciones diferentes de pares `(fila, columna)`. Para cada posición de la secuencia, el elemento de `Mtr` correspondiente se copia a `my_array`, produciendo así una linealización final acorde con el patrón seleccionado. A continuación, el contenido de `my_array` se replica en `P` para uniformizar la representación lineal antes del volcado de salida.

En `State8` se reconstruye la matriz de salida `matrix_out` recorriendo `my_array` por orden e insertando sus valores por filas. Con ello se completa la transformación de desmezcla, entregando en la salida la matriz resultante tras la recombinación por bloques y el recorrido diagonal inverso.

En conjunto, el módulo de desmezcla aplica de manera determinista una secuencia de operaciones inversa a la etapa de mezcla: aplanado de entrada, generación y selección de parámetros pseudoaleatorios, ordenación de pivotes, particionado y recombinación por bloques, reensamblado matricial y lectura diagonal controlada. El resultado es una matriz de salida coherente con la inversión del proceso de mezcla, bajo la misma configuración de parámetros y semilla que gobierna la trayectoria pseudoaleatoria.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use ieee.math_real.all;
5  use work.tipo.all;
6  entity deshuffling is
7  generic(
8   filas : integer := 10;
9   columnas : integer := 10;
10  semilla : integer := 500000;
11  r : integer := 3800000
12 );
13 port(
14  clk : in std_logic;
15  reset : in std_logic;
16  matrix_in : in matrix(0 to filas-1, 0 to columnas-1);
17  matrix_out : out matrix(0 to filas-1, 0 to columnas-1);
18  done        : out std_logic
19 );
20 end entity deshuffling;
21
22 architecture Behavorial of deshuffling is
23  type Estados is (State0, State1, State2, State3, State4,
       State5, State6, State7, State8);
24  signal estado_actual, estado_siguiente : Estados := State0;
25  signal row, col : integer :=0;
26 begin
27 process(clk, reset)
28 begin
```

```
29  if clk'event and clk='1' then
30   if reset='1' then
31    estado_actual<=State0;
32   else
33    estado_actual<=estado_siguiente;
34   end if;
35  end if;
36 end process;
37 process(estado_actual)
38  variable x_var : integer := semilla;
39  variable pseudorandom_var : byte_array(0 to 255);
40  variable limit1, limit2, limit3, limit4 : integer;
41  variable t1, t2, t3, t4, taux1, taux2, taux3, taux4 :
       integer := 0;
42  constant max_limit1, max_limit2, max_limit3, max_limit4 :
       integer := 99;
43  variable a1, a2, a3, n, m : integer :=0;
44  variable P, P1, P2, P3, P4, aux1, aux2, aux3, aux4, a,
       my_array : byte_array(0 to filas*columnas-1) := (others
       =>0);
45  variable index, index1, i : integer :=0;
46  variable sorted_a1, sorted_a2, sorted_a3 : integer :=0;
47  variable Mtr : matrix(0 to filas-1, 0 to columnas-1);
48  variable read_sequence : coordinate_array;
49 begin
50  case estado_actual is
51  when State0 =>
52   index:=0;
53   for i in 0 to filas-1 loop
54    for j in 0 to columnas-1 loop
55     a(index) := matrix_in(i, j);
56     index := index + 1;
57    end loop;
58   end loop;
59   estado_siguiente<=State1;
60  when State1 =>
61   for k in 0 to 255 loop
62    x_var := (r * x_var * (1000000 - x_var))/1000000;
63    pseudorandom_var(k):= x_var mod 256;
64   end loop;
65   estado_siguiente<=State2;
66  when State2=>
67   a1 := pseudorandom_var(12) mod filas*columnas;
```

```
68    a2 := pseudorandom_var(103) mod filas*columnas;
69    a3 := pseudorandom_var(66) mod filas*columnas;
70    n := pseudorandom_var(174) mod 4;
71    m := pseudorandom_var(239) mod 4;
72    estado_siguiente<=State3;
73   when State3 =>
74    if a1 <= a2 and a1 <= a3 then
75     sorted_a1 := a1;
76     if a2 <= a3 then
77      sorted_a2 := a2;
78      sorted_a3 := a3;
79     else
80      sorted_a2 := a3;
81      sorted_a3 := a2;
82     end if;
83    elsif a2 <= a1 and a2<= a3 then
84     sorted_a1 := a2;
85     if a1 <= a3 then
86      sorted_a2 := a1;
87      sorted_a3 := a3;
88     else
89      sorted_a2 := a3;
90      sorted_a3 := a1;
91     end if;
92    else
93     sorted_a1 := a3;
94     if a1 <= a2 then
95      sorted_a2 := a1;
96      sorted_a3 := a2;
97     else
98      sorted_a2 := a2;
99      sorted_a3 := a1;
100    end if;
101   end if;
102   estado_siguiente<=State4;
103  when State4 =>
104   case n is
105   when 0 =>
106    index:=0;
107    for i in 0 to filas*columnas-1 loop
108         if i >= sorted_a3 and i<=filas*columnas-1 then
109          P1(index):=a(i);
110          index:=index+1;
```

```
111         end if;
112    end loop;
113    index:=0;
114    for i in 0 to filas*columnas-1 loop
115         if i< sorted_a1 then
116          P2(index):=a(i);
117          index:=index+1;
118         end if;
119    end loop;
120    index:=0;
121    for i in 0 to filas*columnas-1 loop
122         if i>=sorted_a1 and i<sorted_a2 then
123      P3(index):=a(i);
124          index:=index+1;
125         end if;
126    end loop;
127    index:=0;
128    for i in 0 to filas*columnas-1 loop
129         if  i>=sorted_a2 and i<sorted_a3  then
130          P4(index):=a(i);
131          index:=index+1;
132         end if;
133    end loop;
134
135   when 1 =>
136    index:=0;
137    for i in 0 to filas*columnas-1 loop
138         if i >= sorted_a1 and i<sorted_a2 then
139          P1(index):=a(i);
140          index:=index+1;
141         end if;
142    end loop;
143    index:=0;
144    for i in 0 to filas*columnas-1 loop
145         if i< sorted_a1 then
146          P2(index):=a(i);
147          index:=index+1;
148         end if;
149    end loop;
150    index:=0;
151    for i in 0 to filas*columnas-1 loop
152     if i>=sorted_a3 and i<=filas*columnas-1 then
153      P3(index):=a(i);
```

```
154       index:=index+1;
155      end if;
156     end loop;
157     index:=0;
158     for i in 0 to filas*columnas-1 loop
159      if i>=sorted_a2 and i<sorted_a3 then
160       P4(index):=a(i);
161       index:=index+1;
162      end if;
163     end loop;
164    when 2 =>
165     index:=0;
166     for i in 0 to filas*columnas-1 loop
167      if i >= sorted_a2 and i<sorted_a3 then
168       P1(index):=a(i);
169       index:=index+1;
170      end if;
171     end loop;
172     index:=0;
173     for i in 0 to filas*columnas-1 loop
174      if i< sorted_a1 then
175       P2(index):=a(i);
176       index:=index+1;
177      end if;
178     end loop;
179     index:=0;
180     for i in 0 to filas*columnas-1 loop
181      if i>=sorted_a3 and i<=filas*columnas-1 then
182       P3(index):=a(i);
183       index:=index+1;
184      end if;
185     end loop;
186     index:=0;
187     for i in 0 to filas*columnas-1 loop
188      if i>=sorted_a1 and i<sorted_a2 then
189       P4(index):=a(i);
190       index:=index+1;
191      end if;
192     end loop;
193    when 3 =>
194     index:=0;
195     for i in 0 to filas*columnas-1 loop
196      if i >= sorted_a3 and i<=filas*columnas-1 then
```

```
197       P1(index):=a(i);
198       index:=index+1;
199      end if;
200     end loop;
201     index:=0;
202     for i in 0 to filas*columnas-1 loop
203      if i >= sorted_a2 and i<sorted_a3 then
204       P2(index):=a(i);
205       index:=index+1;
206      end if;
207     end loop;
208     index:=0;
209     for i in 0 to filas*columnas-1 loop
210      if i>=sorted_a1 and i<sorted_a2 then
211       P3(index):=a(i);
212       index:=index+1;
213      end if;
214     end loop;
215     index:=0;
216     for i in 0 to filas*columnas-1 loop
217      if i<sorted_a1 then
218       P4(index):=a(i);
219       index:=index+1;
220      end if;
221     end loop;
222    when others =>
223    end case;
224    estado_siguiente<=State5;
225   when State5 =>
226    i:=0;
227    index:=0;
228    while P1(index) /=0 and index <99 loop
229     P(i) := P1(index);
230     index:=index+1;
231     i:=i+1;
232    end loop;
233    index:=0;
234    while P2(index) /=0 and index <99 loop
235     P(i) := P2(index);
236     i:=i+1;
237     index:=index+1;
238    end loop;
239    index:=0;
```

```
240   while P3(index) /=0 and index <99 loop
241    P(i) := P3(index);
242    i:=i+1;
243    index:=index+1;
244   end loop;
245   index:=0;
246   while P4(index) /=0 and index <99 loop
247    P(i) := P4(index);
248    i:=i+1;
249    index:=index+1;
250   end loop;
251   estado_siguiente<=State6;
252  when State6 =>
253   index:=0;
254   for i in 0 to filas-1 loop
255    for j in 0 to columnas-1 loop
256     Mtr(i,j):=P(index);
257     index:=index+1;
258    end loop;
259   end loop;
260   estado_siguiente<=State7;
261  when State7 =>
262   case m is
263   when 0 =>
264    read_sequence:=((7,9), (3,0), (0,1), (4,1), (1,1), (5,0),
        (0,6), (5,6), (2,4), (4,0),
265    (8,5), (5,7), (0,4), (1,0), (2,5), (4,6), (3,2), (2,1),
        (1,6), (0,8),
266    (3,4), (7,3), (2,3), (2,9), (8,7), (1,3), (0,3), (1,8),
        (3,6), (5,5),
267    (6,2), (9,0), (4,8), (0,2), (0,0), (6,0), (7,2), (1,5),
        (0,5), (4,5),
268    (3,1), (9,2), (8,0), (1,7), (2,2), (5,3), (3,3), (3,8),
        (2,8), (4,7),
269    (5,2), (7,1), (6,1), (6,6), (6,4), (7,7), (8,2), (1,9),
        (0,7), (6,8),
270    (4,5), (9,4), (8,4), (2,7), (3,9), (9,9), (9,7), (5,1),
        (0,9), (3,7),
271    (4,4), (3,6), (8,1), (9,6), (8,5), (1,2), (7,0), (7,6),
        (2,6), (6,5),
272    (9,1), (8,3), (7,8), (6,7), (5,3), (7,4), (8,9), (9,5),
        (4,2), (1,4),
273    (5,9), (7,5), (4,3), (9,3), (4,9), (8,8), (5,8), (9,8),
```

```
      (6,9), (2,0)) ;
274
275   when 1 =>
276    read_sequence:=((2,0), (6,9), (9,8), (5,8), (8,8), (4,9),
          (9,3), (4,3), (7,5), (5,9),
277    (1,4), (4,2), (9,5), (8,9), (7,4), (5,3), (6,7), (7,8),
          (8,3), (9,1),
278    (6,5), (2,6), (7,6), (7,0), (1,2), (8,6), (9,6), (8,1),
          (6,3), (4,4),
279    (3,7), (0,9), (5,1), (9,7), (9,9), (3,9), (2,7), (8,4),
          (9,4), (5,4),
280    (6,8), (0,7), (1,9), (8,2), (7,7), (6,4), (6,6), (6,1),
          (7,1), (5,2),
281    (4,7), (2,8), (3,8), (3,3), (3,5), (2,2), (1,7), (8,0),
          (9,2), (3,1),
282    (4,5), (0,5), (1,5), (7,2), (6,0), (0,0), (0,2), (4,8),
          (9,0), (6,2),
283    (5,5), (3,6), (1,8), (0,3), (1,3), (8,7), (2,9), (2,3),
          (7,3), (3,4),
284    (0,8), (1,6), (2,1), (3,2), (4,6), (2,5), (1,0), (0,4),
          (5,7), (8,5),
285    (4,0), (2,4), (5,6), (0,6), (5,0), (1,1), (4,1), (0,1),
          (3,0), (7,9));
286
287   when 2 =>
288    read_sequence:=((9,9), (7,9), (9,6), (5,0), (3,6), (9,0),
          (6,9), (8,3), (8,9), (1,5),
289    (9,4), (8,0), (5,9), (8,6), (5,1), (7,6), (8,2), (8,8),
          (2,1), (2,2),
290    (4,5), (5,8), (5,5), (6,0), (7,5), (7,4), (8,1), (9,3),
          (2,9), (3,8),
291    (4,7), (8,5), (6,8), (6,6), (5,6), (6,5), (8,7), (2,8),
          (3,7), (3,5),
292    (9,7), (6,7), (4,6), (9,5), (9,8), (7,3), (9,2), (2,7),
          (0,8), (2,4),
293    (5,7), (9,1), (7,2), (0,7), (2,6), (0,1), (0,4), (3,5),
          (2,3), (0,2),
294    (6,4), (6,2), (7,1), (2,1), (3,4), (4,3), (3,3), (1,3),
          (1,4), (5,2),
295    (6,1), (7,0), (0,6), (1,8), (2,5), (2,4), (3,9), (4,4),
          (4,1), (5,4),
296    (7,7), (7,8), (1,1), (1,7), (2,3), (4,8), (1,3), (4,0),
          (1,9), (0,5),
```

```
297     (8,4), (1,0), (1,6), (3,0), (0,9), (6,3), (4,9), (0,3),
           (2,0), (0,0));
298
299    when 3 =>
300     read_sequence:=((0,0), (2,0), (0,3), (4,9), (6,3), (0,9),
           (3,0), (1,6), (1,0), (8,4),
301     (0,5), (1,9), (4,0), (1,3), (4,8), (2,3), (2,7), (1,1),
           (7,8), (7,7),
302     (5,4), (4,1), (4,4), (3,9), (2,4), (2,5), (1,8), (0,6),
           (7,0), (6,1),
303     (5,2), (1,5), (3,1), (3,3), (4,3), (3,4), (1,2), (7,1),
           (6,2), (6,4),
304     (0,2), (3,2), (5,3), (0,4), (0,1), (2,6), (0,7), (7,2),
           (9,1), (5,7),
305     (4,2), (0,8), (2,7), (9,2), (7,3), (9,8), (9,5), (4,6),
           (6,7), (9,7),
306     (3,5), (3,7), (2,8), (8,7), (6,5), (5,6), (6,6), (6,8),
           (8,5), (4,7),
307     (3,8), (2,9), (9,3), (8,1), (7,4), (7,5), (6,0), (5,5),
           (5,8), (4,5),
308     (2,2), (2,1), (8,8), (8,2), (7,6), (5,1), (8,6), (5,9),
           (8,0), (9,4),
309     (1,5), (8,9), (8,3), (6,9), (9,0), (3,6), (5,0), (9,6),
           (7,9), (9,9));
310
311    when others =>
312
313    end case;
314
315    for t in 0 to filas*columnas-1 loop
316     my_array(t) := Mtr(read_sequence(t).row, read_sequence(t).
           col);
317    end loop;
318    for w in 0 to filas*columnas-1 loop
319     P(w):=my_array(w);
320    end loop;
321    estado_siguiente<=State8;
322   when State8 =>
323    index:=0;
324    for i in 0 to filas-1 loop
325     for j in 0 to columnas-1 loop
326      matrix_out(i,j)<=my_array(index);
327      index:=index+1;
```

```
328     end loop;
329   end loop;
330   estado_siguiente<=State0;
331  end case;
332 end process;
333 end architecture behavorial;
```

Código 4: Módulo deshuffling

5.2.5. Implementación del módulo descrambling

El módulo `descrambling` implementa la etapa final del proceso de reconstrucción de datos, actuando como inverso funcional del módulo `scrambling`. Su objetivo principal es revertir la dispersión aplicada sobre la matriz de entrada, restaurando los valores originales mediante la aplicación de la tabla de sustitución inversa (`inv_sbox`) y las operaciones lógicas complementarias. De esta forma, el sistema recupera la coherencia estructural del conjunto de datos tras el proceso de mezcla no lineal.

El módulo se declara como entidad parametrizable mediante los genéricos `filas`, `columnas`, `semilla` y `r`, que definen las dimensiones de la matriz y los parámetros del mapa caótico. La interfaz de puertos incluye las claves `K1`, `K2`, `K3`, `K4`, los coeficientes `C1` y `C2`, las secuencias de estado `S1`, `S2`, `S3`, la matriz de entrada `matrix_in`, la matriz de salida `matrix_out` y la señal `ready`, que indica la finalización del proceso de descrifrado lógico. Todos los datos se manipulan en formato de arrays de bytes definidos en el paquete `tipo`.

La arquitectura `Behavioral` utiliza una máquina de estados finita con seis estados (`State0` a `State5`) que organizan secuencialmente las fases del proceso. Además, se define una constante `inv_sbox` de tipo `sbox_array`, que contiene la tabla de sustitución inversa correspondiente a la empleada en el módulo `scrambling`. Esta estructura fija permite aplicar la transformación no lineal inversa sobre cada elemento del conjunto de datos, restaurando los valores originales antes de la difusión.

En el estado `State0` se produce la inicialización del sistema y la transición hacia el estado operativo `State1`. En esta fase, el vector de estado interno `int_state` se reconstruye combinando las claves y secuencias de entrada. Los valores se asignan de forma ordenada para reproducir la disposición utilizada durante la mezcla: las claves `K1`, `K2`, `K3` y `K4`, los coeficientes `C1`, `C2`, y las secuencias `S1`, `S2` y `S3`. Paralelamente, la matriz de entrada `matrix_in` se aplana en un vector lineal `a`, que contendrá los 100 elementos a procesar.

El estado `State2` aplica la sustitución inversa sobre todos los elementos del vector `a`, utilizando la tabla `inv_sbox`. Cada valor almacenado se traduce a su correspondiente elemento inverso, lo que constituye el primer paso para revertir la transformación no lineal aplicada previamente. Este procedimiento garantiza que la operación de dispersión basada en `sbox` pueda deshacerse de forma exacta, asegurando la reversibilidad del proceso completo.

En el estado `State3`, el vector `a` se divide nuevamente en tres fragmentos: `fragmento`, `fragmento1` y `fragmento2`. Los dos primeros contienen 36 elementos, y el último 28, completado con ceros hasta un total de 36 posiciones. Sobre estos bloques se aplican las operaciones de restauración mediante la función `xor_byte_array`, que invierte la combinación lógica realizada durante la fase de mezcla. En concreto, cada fragmento se combina con el vector de estado interno `int_state`, obteniéndose los vectores `orig0`, `orig1` y `orig2`. Estos resultados sustituyen a los fragmentos originales dentro del vector `a`, completando la fase de reconstrucción.

Posteriormente, en el estado `State4`, se reconstituye la matriz bidimensional original. Los valores almacenados en el vector `a` se recorren secuencialmente y se vuelcan por filas sobre `matrix_out`. Este paso final restituye la estructura matricial que había sido linealizada durante el proceso de dispersión.

Finalmente, en el estado `State5`, se activa la señal `ready`, indicando que el proceso ha concluido correctamente y que la matriz de salida contiene los datos descrifrados. A partir de este punto, el sistema puede volver a su estado de espera o continuar con operaciones adicionales de verificación o almacenamiento.

En conjunto, el módulo `descrambling` constituye la etapa de restauración lógica del sistema, complementaria al proceso de dispersión. Su diseño garantiza la reversibilidad completa de la transformación, preservando la integridad de los datos originales y asegurando que la secuencia de operaciones pueda invertirse sin pérdida de información, siempre que se mantengan los mismos parámetros, claves y semillas utilizados en la fase de mezcla.

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4 use work.tipo.all;
5
6 entity descrambling is
7 generic(
```

```
 filas     : integer := 10;
 columnas  : integer := 10;
 semilla   : integer := 500000;
 r         : integer := 3800000
);
port(
 clk         : in  std_logic;
 reset       : in  std_logic;
 K1, K2, K3, K4      : in  byte_array(0 to 3);
 C1, C2      : in  byte_array(0 to 3);
 S1, S2, S3  : in  byte_array(0 to 3);
 matrix_in   : in  matrix(0 to filas-1, 0 to columnas-1);
 matrix_out  : out matrix(0 to filas-1, 0 to columnas-1);
 ready        : out std_logic
);
end entity descrambling;

architecture Behavioral of descrambling is
 type Estado is (State0, State1, State2, State3, State4,
    State5);
 signal estado_actual, estado_siguiente : Estado;
 constant inv_sbox: sbox_array := (
 x"52", x"09", x"6a", x"d5", x"30", x"36", x"a5", x"38", x"bf
    ", x"40", x"a3", x"9e", x"81", x"f3", x"d7", x"fb",
 x"7c", x"e3", x"39", x"82", x"9b", x"2f", x"ff", x"87", x"34
    ", x"8e", x"43", x"44", x"c4", x"de", x"e9", x"cb",
 x"54", x"7b", x"94", x"32", x"a6", x"c2", x"23", x"3d", x"ee
    ", x"4c", x"95", x"0b", x"42", x"fa", x"c3", x"4e",
 x"08", x"2e", x"a1", x"66", x"28", x"d9", x"24", x"b2", x"76
    ", x"5b", x"a2", x"49", x"6d", x"8b", x"d1", x"25",
 x"72", x"f8", x"f6", x"64", x"86", x"68", x"98", x"16", x"d4
    ", x"a4", x"5c", x"cc", x"5d", x"65", x"b6", x"92",
 x"6c", x"70", x"48", x"50", x"fd", x"ed", x"b9", x"da", x"5e
    ", x"15", x"46", x"57", x"a7", x"8d", x"9d", x"84",
 x"90", x"d8", x"ab", x"00", x"8c", x"bc", x"d3", x"0a", x"f7
    ", x"e4", x"58", x"05", x"b8", x"b3", x"45", x"06",
 x"d0", x"2c", x"1e", x"8f", x"ca", x"3f", x"0f", x"02", x"c1
    ", x"af", x"bd", x"03", x"01", x"13", x"8a", x"6b",
 x"3a", x"91", x"11", x"41", x"4f", x"67", x"dc", x"ea", x"97
    ", x"f2", x"cf", x"ce", x"f0", x"b4", x"e6", x"73",
 x"96", x"ac", x"74", x"22", x"e7", x"ad", x"35", x"85", x"e2
    ", x"f9", x"37", x"e8", x"1c", x"75", x"df", x"6e",
 x"47", x"f1", x"1a", x"71", x"1d", x"29", x"c5", x"89", x"6f
```

```
    ", x"b7", x"62", x"0e", x"aa", x"18", x"be", x"1b",
40  x"fc", x"56", x"3e", x"4b", x"c6", x"d2", x"79", x"20", x"9a
    ", x"db", x"c0", x"fe", x"78", x"cd", x"5a", x"f4",
41  x"1f", x"dd", x"a8", x"33", x"88", x"07", x"c7", x"31", x"b1
    ", x"12", x"10", x"59", x"27", x"80", x"ec", x"5f",
42  x"60", x"51", x"7f", x"a9", x"19", x"b5", x"4a", x"0d", x"2d
    ", x"e5", x"7a", x"9f", x"93", x"c9", x"9c", x"ef",
43  x"a0", x"e0", x"3b", x"4d", x"ae", x"2a", x"f5", x"b0", x"c8
    ", x"eb", x"bb", x"3c", x"83", x"53", x"99", x"61",
44  x"17", x"2b", x"04", x"7e", x"ba", x"77", x"d6", x"26", x"e1
    ", x"69", x"14", x"63", x"55", x"21", x"0c", x"7d"
45  );
46
47  begin
48   process(clk, reset)
49   begin
50    if rising_edge(clk) then
51     if reset='1' then
52      estado_actual <= State0;
53     else
54      estado_actual <= estado_siguiente;
55     end if;
56    end if;
57   end process;
58
59   process(estado_actual)
60    variable idx, i, j    : integer;
61    variable fragmento    : byte_array(0 to 35);
62    variable fragmento1   : byte_array(0 to 35);
63    variable fragmento2   : byte_array(0 to 35);
64    variable a            : byte_array(0 to filas*columnas-1);
65    variable int_state    : byte_array(0 to 35);
66    variable orig0, orig1, orig2 : byte_array(0 to 35);
67   begin
68    case estado_actual is
69    when State0 =>
70     estado_siguiente <= State1;
71    when State1 =>
72     int_state(0 to 3)   := K1;
73     int_state(4 to 7)   := S1;
74     int_state(8 to 11)  := S2;
75     int_state(12 to 15) := S3;
76     int_state(16 to 19) := K2;
```

```
77    int_state(20 to 23) := K4;
78    int_state(24 to 27) := C1;
79    int_state(28 to 31) := C2;
80    int_state(32 to 35) := K3;
81    idx := 0;
82    for i in 0 to filas-1 loop
83     for j in 0 to columnas-1 loop
84      a(idx) := matrix_in(i,j);
85      idx := idx + 1;
86     end loop;
87    end loop;
88    estado_siguiente <= State2;
89   when State2 =>
90    for idx in 0 to filas*columnas-1 loop
91     a(idx) := to_integer(unsigned(inv_sbox(a(idx))));
92    end loop;
93    estado_siguiente <= State3;
94   when State3 =>
95    fragmento  := a(0 to 35);
96    fragmento1 := a(36 to 71);
97    fragmento2 := (others => 0);
98    fragmento2(0 to 27) := a(72 to 99);
99
100   orig0 := xor_byte_array(int_state(0 to 35), fragmento);
101   orig1 := xor_byte_array(int_state(0 to 35), fragmento1);
102   orig2 := xor_byte_array(int_state(0 to 35), fragmento2);
103
104   a(0 to 35)   := orig0;
105   a(36 to 71)  := orig1;
106   a(72 to 99)  := orig2(0 to 27);
107
108   estado_siguiente <= State4;
109
110  when State4 =>
111   idx := 0;
112   for i in 0 to filas-1 loop
113    for j in 0 to columnas-1 loop
114     matrix_out(i,j) <= a(idx);
115     idx := idx + 1;
116    end loop;
117   end loop;
118   estado_siguiente <= State5;
119
```

```
120   when State5 =>
121    ready <= '1';
122    estado_siguiente <= State5;
123   end case;
124  end process;
125 end architecture Behavioral;
```

Código 5: Módulo descrambling

5.3. Plan de validación experimental

El proceso de validación experimental tiene como finalidad comprobar el correcto funcionamiento de los módulos desarrollados, así como verificar la coherencia del sistema completo tras la integración de todas sus partes. Para ello, se han diseñado distintos *testbenches* que permiten evaluar de manera independiente cada módulo y analizar posteriormente su comportamiento conjunto. Las pruebas se han realizado utilizando **Xilinx ISE Design Suite 14.7**, empleando simulaciones funcionales para verificar la lógica interna y la correcta propagación de señales entre etapas.

5.3.0.1. Estructura del plan de validación

La validación se organiza en cuatro niveles jerárquicos:

1. Validación individual de cada módulo: verificación funcional de *Shuffling*, *Scrambling*, *Deshuffling* y *Descrambling*.

2. Pruebas de reversibilidad: comprobación del comportamiento inverso entre los módulos *Shuffling–Deshuffling* y *Scrambling–Descrambling*.

3. Validación de propagación entre módulos: análisis de la coherencia del flujo de datos entre bloques consecutivos en la *Ofuscación* y *Desofuscación*.

4. Escenario de prueba completo: validación del sistema integrado mediante un módulo superior (*Top Module*).

Cada nivel se documenta mediante un código de testbench en VHDL, que permite reproducir experimentalmente los resultados obtenidos en las simulaciones.

5.3.0.2. Objetivos de la validación

Los principales objetivos de la validación experimental son:

- Verificar la correcta implementación de la lógica de control y de las operaciones internas de cada módulo.
- Garantizar que las transformaciones realizadas sean completamente reversibles y deterministas bajo los mismos parámetros de configuración.
- Comprobar la correcta propagación de datos entre los distintos módulos y la integridad de la información en todo el flujo de procesamiento.
- Validar el funcionamiento global del sistema integrado en condiciones de operación equivalentes a las de su uso real.

5.3.0.3. Validación de módulos

En esta primera fase se realiza la validación individual de cada bloque funcional del sistema. Cada módulo ha sido verificado de forma aislada mediante un testbench específico, diseñado para evaluar la secuencia de operaciones, las señales de control y el resultado de las transformaciones aplicadas.

5.3.0.3.1 Validación del módulo shuffling

El testbench del módulo *Shuffling* verifica el correcto funcionamiento de la máquina de estados finita encargada de reorganizar los elementos de la matriz de entrada. Se comprueba que la secuencia pseudoaleatoria generada por el mapa caótico produce una permutación válida y que la matriz de salida contiene todos los elementos originales en posiciones diferentes.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use work.tipo.all;
5
6  entity shuffling_tb is
7  end shuffling_tb;
8
9  architecture testbench of shuffling_tb is
10
```

```
11  signal clk         : std_logic := '0';
12  signal reset       : std_logic := '0';
13  signal done        : std_logic:='0';
14  signal matrix_in   : matrix(0 to 9, 0 to 9) := (
15  (1, 2, 3, 4, 5, 6, 7, 8, 9, 10),
16  (11, 12, 13, 14, 15, 16, 17, 18, 19, 20),
17  (21, 22, 23, 24, 25, 26, 27, 28, 29, 30),
18  (31, 32, 33, 34, 35, 36, 37, 38, 39, 40),
19  (41, 42, 43, 44, 45, 46, 47, 48, 49, 50),
20  (51, 52, 53, 54, 55, 56, 57, 58, 59, 60),
21  (61, 62, 63, 64, 65, 66, 67, 68, 69, 70),
22  (71, 72, 73, 74, 75, 76, 77, 78, 79, 80),
23  (81, 82, 83, 84, 85, 86, 87, 88, 89, 90),
24  (91, 92, 93, 94, 95, 96, 97, 98, 99, 100)
25  );
26  signal matrix_out : matrix(0 to 9, 0 to 9);
27
28  constant filas    : integer := 10;
29  constant columnas : integer := 10;
30  constant clk_period : time := 5 ns;
31
32 begin
33
34  U1: entity work.shuffling
35  generic map(
36   filas => filas,
37   columnas => columnas,
38   semilla => 500000,
39   r => 3800000
40  )
41  port map(
42   clk         => clk,
43   reset       => reset,
44   matrix_in   => matrix_in,
45   matrix_out => matrix_out,
46   done        => done
47  );
48
49  clk_process : process
50  begin
51   while true loop
52    clk <= '1';
53    wait for clk_period;
```

```
54     clk <= '0';
55     wait for clk_period;
56    end loop;
57  end process;
58
59  stim_proc: process
60  begin
61    reset <= '1';
62    wait for clk_period;
63    reset <= '0';
64    wait;
65  end process;
66
67 end architecture;
```

Código 6: Testbench shuffling

5.3.0.3.2 Validación del módulo scrambling

El testbench correspondiente evalúa la aplicación de las transformaciones aritméticas y lógicas, asegurando que las operaciones XOR, sumas módulo 255 y sustituciones *S-box* se ejecutan correctamente. También se comprueba la sincronización de la señal `scrambled_ready` y la integridad de los datos de salida.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use work.tipo.all;
5  entity tb_scrambling is
6  end entity tb_scrambling;
7
8  architecture testbench of tb_scrambling is
9   signal clk : std_logic := '0';
10  signal reset : std_logic := '0';
11  signal ready : std_logic;
12  signal K1: byte_array(0 to 3) := (19,72,29,12);
13  signal K2 : byte_array(0 to 3):= (129,93,123,233);
14  signal K3 : byte_array(0 to 3):= (32,35,76,86);
15  signal K4 : byte_array(0 to 3):= (54,78,66,92);
16  signal C1 : byte_array(0 to 3) := (23,54,29,32);
17  signal C2 : byte_array(0 to 3):= (176,193,183,2);
18  signal S1 : byte_array(0 to 3):= (87,77,89,32);
```

```
19  signal S2 : byte_array(0 to 3):= (53,75,33,45);
20  signal S3 : byte_array(0 to 3):= (14,44,67,76);
21  signal matrix_in : matrix(0 to 9, 0 to 9) := (
22  (1, 2, 3, 4, 5, 6, 7, 8, 9, 10),
23  (11, 12, 13, 14, 15, 16, 17, 18, 19, 20),
24  (21, 22, 23, 24, 25, 26, 27, 28, 29, 30),
25  (31, 32, 33, 34, 35, 36, 37, 38, 39, 40),
26  (41, 42, 43, 44, 45, 46, 47, 48, 49, 50),
27  (51, 52, 53, 54, 55, 56, 57, 58, 59, 60),
28  (61, 62, 63, 64, 65, 66, 67, 68, 69, 70),
29  (71, 72, 73, 74, 75, 76, 77, 78, 79, 80),
30  (81, 82, 83, 84, 85, 86, 87, 88, 89, 90),
31  (91, 92, 93, 94, 95, 96, 97, 98, 99, 100)
32  );
33  signal matrix_out : matrix(0 to 9, 0 to 9);
34  constant clk_period : time := 5 ns;
35 begin
36
37  U1: entity work.scrambling
38  generic map(
39   filas => 10,
40   columnas => 10,
41   semilla => 500000,
42   r => 3800000
43  )
44 port map (
45   clk          => clk,
46   reset        => reset,
47   K1           => K1,
48   K2           => K2,
49   K3           => K3,
50   K4           => K4,
51   C1           => C1,
52   C2           => C2,
53   S1           => S1,
54   S2           => S2,
55   S3           => S3,
56   matrix_in    => matrix_in,
57   matrix_out   => matrix_out,
58   scrambled_ready => ready
59  );
60
61  clk_process : process
```

```
 begin
  while true loop
   clk <= '0';
   wait for clk_period ;
   clk <= '1';
   wait for clk_period ;
  end loop;
 end process;

 stimulus_process : process
 begin
  reset <= '1';
  wait for clk_period;
  reset <= '0';
  wait;
 end process;

end architecture testbench;
```

Código 7: Testbench scrambling

5.3.0.3.3 Validación del módulo deshuffling

Esta prueba comprueba que el módulo *Deshuffling* es capaz de revertir de forma exacta el proceso de barajado aplicado por el módulo *Shuffling*. Se valida que, utilizando la misma semilla y los mismos parámetros del mapa caótico, la matriz resultante coincide con la de entrada original.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.tipo.all;

entity tb_deshuffling is
end entity tb_deshuffling;

architecture test of tb_deshuffling is

 signal clk : std_logic := '0';
 signal reset : std_logic := '0';
 signal done    : std_logic:='0';
 signal matrix_in : matrix(0 to 9, 0 to 9) := (
```

```
 (16, 48, 17, 24, 38, 12, 44, 72, 31, 62),
 (37, 46, 55, 25, 81, 13, 32, 7, 23, 73),
 (91, 33, 6, 28, 42, 36, 52, 67, 2, 27),
 (49, 10, 34, 4, 30, 5, 22, 61, 3, 66),
 (41, 47, 82, 98, 60, 11, 35, 1, 18, 96),
 (45, 63, 80, 86, 70, 21, 43, 39, 94, 100),
 (15, 78, 20, 59, 76, 51, 77, 87, 71, 92),
 (54, 79, 14, 29, 85, 99, 53, 75, 88, 50),
 (8, 58, 74, 89, 68, 40, 56, 26, 95, 84),
 (19, 90, 9, 97, 69, 83, 57, 64, 93, 65)
 );
 signal matrix_out : matrix(0 to 9, 0 to 9);
 constant clk_period : time := 5 ns;
begin
 uut: entity work.deshuffling
 generic map (
  filas => 10,
  columnas => 10,
  semilla => 500000,
  r => 3800000
 )
 port map (
  clk => clk,
  matrix_in => matrix_in,
  matrix_out => matrix_out,
  reset =>reset,
  done  => done
 );

 clk_process : process
 begin
  clk <= '0';
  wait for clk_period;
  clk <= '1';
  wait for clk_period;
 end process;

 stimulus_process : process
 begin
  reset <= '1';
  wait for clk_period;
  reset <= '0';
  wait;
```

```
 end process;
end architecture test;
```

Código 8: Testbench deshuffling

5.3.0.3.4 Validación del módulo descrambling

Finalmente, el testbench del módulo *Descrambling* verifica la correcta inversión de la sustitución *S-box* mediante su tabla inversa y la reversión de las operaciones lógicas realizadas durante el *Scrambling*. La validación confirma que los datos restaurados en la salida son idénticos a los originales antes de la transformación.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.tipo.all;

entity tb_descrambling is
end entity tb_descrambling;

architecture testbench of tb_descrambling is

 signal clk : std_logic := '0';
 signal reset : std_logic := '0';
 signal ready : std_logic;
 signal K1: byte_array(0 to 3) := (19,72,29,12);
 signal K2 : byte_array(0 to 3):= (129,93,123,233);
 signal K3 : byte_array(0 to 3):= (32,35,76,86);
 signal K4 : byte_array(0 to 3):= (54,78,66,92);
 signal C1 : byte_array(0 to 3) := (23,54,29,32);
 signal C2 : byte_array(0 to 3):= (176,193,183,2);
 signal S1 : byte_array(0 to 3):= (87,77,89,32);
 signal S2 : byte_array(0 to 3):= (53,75,33,45);
 signal S3 : byte_array(0 to 3):= (14,44,67,76);
 signal matrix_in : matrix(0 to 9, 0 to 9) := (
 (201, 214, 114, 48, 0, 179, 88, 52, 235, 131),
 (229, 253, 123, 147, 41, 74, 96, 132, 69, 84),
 (38, 106, 252, 27, 171, 113, 111, 235, 149, 158),
 (194, 147, 124, 124, 168, 64, 5, 159, 128, 54),
 (243, 133, 64, 254, 173, 77, 171, 164, 117, 114),
 (81, 188, 141, 127, 41, 62, 118, 146, 182, 208),
 (229, 48, 147, 208, 161, 236, 191, 90, 77, 77),
```

```
31  (43, 114, 190, 119, 177, 9, 162, 123, 71, 81),
32  (67, 212, 64, 182, 57, 218, 250, 250, 97, 197),
33  (183, 213, 127, 202, 164, 235, 56, 32, 243, 27)

34  );
35  signal matrix_out : matrix(0 to 9, 0 to 9);
36
37  constant clk_period : time := 5 ns;
38
39 begin
40
41  U1: entity work.descrambling
42  generic map(
43   filas => 10,
44   columnas => 10,
45   semilla => 500000,
46   r => 3800000
47  )
48 port map (
49   clk         => clk,
50   reset       => reset,
51   K1          => K1,
52   K2          => K2,
53   K3          => K3,
54   K4          => K4,
55   C1          => C1,
56   C2          => C2,
57   S1          => S1,
58   S2          => S2,
59   S3          => S3,
60   matrix_in   => matrix_in,
61   matrix_out  => matrix_out,
62   ready       => ready
63  );
64
65  clk_process : process
66  begin
67   while true loop
68    clk <= '0';
69    wait for clk_period ;
70    clk <= '1';
71    wait for clk_period ;
72   end loop;
```

```
 end process;


 stimulus_process : process
 begin
  reset <= '1';
  wait for clk_period;
  reset <= '0';
  wait;
 end process;

end architecture testbench;
```

Código 9: Testbench descrambling

5.3.0.4. Pruebas de reversibilidad

Para garantizar la coherencia global del sistema, se han diseñado pruebas de reversibilidad entre los pares de módulos que actúan de forma complementaria. Estas pruebas confirman que las transformaciones son totalmente deterministas y que el proceso completo puede revertirse sin pérdida de información.

5.3.0.4.1 Shuffling - Deshuffling

La primera prueba de reversibilidad evalúa la correspondencia exacta entre las operaciones de barajado y desbarajado. La simulación demuestra que, tras aplicar consecutivamente ambos módulos, la matriz obtenida coincide bit a bit con la matriz original.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.tipo.all;

entity tb_shuff_deshuff is
end entity;

architecture sim of tb_shuff_deshuff is

 constant filas    : integer := 10;
 constant columnas : integer := 10;
```

```
13  signal clk         : std_logic := '0';
14  signal reset_sh    : std_logic := '1';
15  signal reset_de    : std_logic := '1';
16  signal matrix_in   : matrix(0 to 9, 0 to 9) := (
17  (1, 2, 3, 4, 5, 6, 7, 8, 9, 10),
18  (11, 12, 13, 14, 15, 16, 17, 18, 19, 20),
19  (21, 22, 23, 24, 25, 26, 27, 28, 29, 30),
20  (31, 32, 33, 34, 35, 36, 37, 38, 39, 40),
21  (41, 42, 43, 44, 45, 46, 47, 48, 49, 50),
22  (51, 52, 53, 54, 55, 56, 57, 58, 59, 60),
23  (61, 62, 63, 64, 65, 66, 67, 68, 69, 70),
24  (71, 72, 73, 74, 75, 76, 77, 78, 79, 80),
25  (81, 82, 83, 84, 85, 86, 87, 88, 89, 90),
26  (91, 92, 93, 94, 95, 96, 97, 98, 99, 100)
27  );
28  signal shuffling_out      : matrix(0 to filas-1, 0 to
       columnas-1);
29  signal deshuffling_in      : matrix(0 to filas-1, 0 to
       columnas-1);
30  signal matrix_out     : matrix(0 to filas-1, 0 to columnas-1)
       ;
31  signal done_sh     : std_logic:='0';
32  signal done_de     : std_logic :='0';
33
34 begin
35
36  clk_proc: process
37  begin
38   clk <= '0';
39   wait for 5 ns;
40   clk <= '1';
41   wait for 5 ns;
42  end process;
43
44
45  U1: entity work.shuffling
46  generic map ( filas => filas, columnas => columnas )
47  port map (
48   clk         => clk,
49   reset       => reset_sh,
50   matrix_in   => matrix_in,
51   matrix_out  => shuffling_out,
52   done        => done_sh
```

```
53 );
54
55
56 U2: entity work.deshuffling
57 generic map ( filas => filas, columnas => columnas )
58 port map (
59  clk         => clk,
60  reset       => reset_de,
61  matrix_in   => deshuffling_in,
62  matrix_out  => matix_out,
63  done        => done_de
64 );
65
66
67 stim_proc: process
68 begin
69
70  reset_sh <= '1'; reset_de <= '1';
71  reset_sh <= '0';
72  wait until done_sh = '1';
73
74  for i in 0 to filas-1 loop
75   for j in 0 to columnas-1 loop
76    deshuffling_in(i,j) <= shuffling_out(i,j);
77   end loop;
78  end loop;
79
80  reset_de <= '0';
81  wait until done_de = '1';
82
83  for i in 0 to filas-1 loop
84   for j in 0 to columnas-1 loop
85    assert mat_out(i,j) = mat_in(i,j)
86    report "ERROR en (" & integer'image(i)&","& integer'image
        (j)&
87    ") esperado="&integer'image(mat_in(i,j))&
88    " obtenido="&integer'image(mat_out(i,j))
89    severity error;
90   end loop;
91  end loop;
92
93 end process;
94
```

```
95 end architecture sim;
```

Código 10: Shuffling - Deshuffling

5.3.0.4.2 Scrambling - Descrambling

De forma análoga, la segunda prueba de reversibilidad confirma que las transformaciones no lineales aplicadas en el módulo *Scrambling* son correctamente invertidas por el módulo *Descrambling*. La salida final tras ejecutar ambos procesos es idéntica a la entrada inicial.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use work.tipo.all;
5
6  entity tb_scrambling_descrambling is
7  end tb_scrambling_descrambling;
8
9  architecture tb of tb_scrambling_descrambling is
10  constant filas    : integer := 10;
11  constant columnas : integer := 10;
12  signal clk   : std_logic := '0';
13  signal reset_scr, reset_descr : std_logic := '1';
14  signal ready_scr : std_logic;
15  signal ready_descr : std_logic;
16  signal K1_scr : byte_array(0 to 3) := (109,82,29,12);
17  signal K2_scr : byte_array(0 to 3):= (129,93,123,233);
18  signal K3_scr : byte_array(0 to 3):= (92,35,71,86);
19  signal K4_scr : byte_array(0 to 3):= (54,78,66,92);
20  signal C1_scr : byte_array(0 to 3) := (23,54,29,32);
21  signal C2_scr : byte_array(0 to 3):= (16,13,113,2);
22  signal S1_scr : byte_array(0 to 3):= (87,47,81,32);
23  signal S2_scr : byte_array(0 to 3):= (40,92,103,82);
24  signal S3_scr : byte_array(0 to 3):= (94,44,61,76);
25  signal K1_descr : byte_array(0 to 3) := (109,82,29,12);
26  signal K2_descr : byte_array(0 to 3):= (129,93,123,233);
27  signal K3_descr : byte_array(0 to 3):= (92,35,71,86);
28  signal K4_descr : byte_array(0 to 3):= (54,78,66,92);
29  signal C1_descr : byte_array(0 to 3) := (23,54,29,32);
30  signal C2_descr : byte_array(0 to 3):= (16,13,113,2);
31  signal S1_descr : byte_array(0 to 3):= (87,47,81,32);
```

```
32  signal S2_descr : byte_array(0 to 3):= (40,92,103,82);
33  signal matrix_in : matrix(0 to 9, 0 to 9) := (
34  (1, 2, 3, 4, 5, 6, 7, 8, 9, 10),
35  (11, 12, 13, 14, 15, 16, 17, 18, 19, 20),
36  (21, 22, 23, 24, 25, 26, 27, 28, 29, 30),
37  (31, 32, 33, 34, 35, 36, 37, 38, 39, 40),
38  (41, 42, 43, 44, 45, 46, 47, 48, 49, 50),
39  (51, 52, 53, 54, 55, 56, 57, 58, 59, 60),
40  (61, 62, 63, 64, 65, 66, 67, 68, 69, 70),
41  (71, 72, 73, 74, 75, 76, 77, 78, 79, 80),
42  (81, 82, 83, 84, 85, 86, 87, 88, 89, 90),
43  (91, 92, 93, 94, 95, 96, 97, 98, 99, 100)
44  );
45  signal scrambling_out   : matrix(0 to filas-1, 0 to columnas
       -1);
46  signal descrambling_in: matrix(0 to filas-1, 0 to columnas
       -1);
47  signal matrix_out : matrix(0 to filas-1, 0 to columnas-1);
48 begin
49  U1 : entity work.scrambling
50  generic map (
51   filas => filas,
52   columnas => columnas,
53   semilla => 500000,
54   r => 3800000
55  )
56  port map (
57   clk         => clk,
58   reset       => reset_scr,
59   K1          => K1_scr,
60   K2          => K2_scr,
61   K3          => K3_scr,
62   K4          => K4_scr,
63   C1          => C1_scr,
64   C2          => C2_scr,
65   S1          => S1_scr,
66   S2          => S2_scr,
67   S3          => S3_scr,
68   matrix_in   => matrix_in,
69   matrix_out => scrambling_out,
70   scrambled_ready => ready_scr
71  );
72
```

```
73  U2 : entity work.descrambling
74  generic map (
75   filas => filas,
76   columnas => columnas,
77   semilla => 500000,
78   r => 3800000
79  )
80  port map (
81   clk         => clk,
82   reset       => reset_descr,
83   K1          => K1_descr,
84   K2          => K2_descr,
85   K3          => K3_descr,
86   K4          => K4_descr,
87   C1          => C1_descr,
88   C2          => C2_descr,
89   S1          => S1_descr,
90   S2          => S2_descr,
91   S3          => S3_descr,
92   matrix_in   => descrambling_in,
93   matrix_out  => matrix_out,
94   ready       => ready_descr
95  );
96
97  clk_process : process
98  begin
99   while true loop
100   clk <= '0'; wait for 10 ns;
101   clk <= '1'; wait for 10 ns;
102  end loop;
103  wait;
104 end process;
105
106 stim_proc: process
107 begin
108  reset_scr <= '1'; reset_descr <= '1';
109  wait for 20 ns;
110  reset_scr <= '0';
111  wait until ready_scr = '1';
112  wait until rising_edge(clk);
113  reset_scr <= '1';
114  for i in 0 to filas-1 loop
115   for j in 0 to columnas-1 loop
```

```
116      scrambled_latched(i,j) <= scrambled_out(i,j);
117     end loop;
118    end loop;
119    reset_descr <= '0';
120    wait for clk_period;
121    for i in 0 to filas-1 loop
122     for j in 0 to columnas-1 loop
123      if descrambled_out(i, j) /= matrix_in(i, j) then
124       report "ERROR: Elemento (" & integer'image(i) & "," &
             integer'image(j) &
125       ") original=" & integer'image(matrix_in(i, j)) &
126       ", recuperado=" & integer'image(descrambled_out(i, j))
             severity error;
127      end if;
128     end loop;
129    end loop;
130    wait;
131   end process;
132 end tb;
```

Código 11: Scrambling - Descrambling

5.3.0.5. Propagación entre módulos

Una vez validado el comportamiento individual y la reversibilidad de los bloques, se analiza la propagación de señales entre los módulos. El objetivo es garantizar la correcta sincronización y la integridad de los datos en el flujo de procesamiento completo.

5.3.0.5.1 Ofuscación

En esta fase, la prueba de ofuscación evalúa el comportamiento conjunto de los módulos *Shuffling* y *Scrambling*, comprobando la correcta transferencia de datos y la validez de las señales de control.

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4 use work.tipo.all;
5
6 entity ofuscation_tb is
```

```
7  end entity;
8
9  architecture sim of ofuscation_tb is
10  constant filas    : integer := 10;
11  constant columnas : integer := 10;
12  signal clk        : std_logic := '0';
13  signal reset_sh   : std_logic := '1';
14  signal reset_scr   : std_logic := '1';
15  signal matrix_in  : matrix(0 to 9, 0 to 9) := (
16  (1, 2, 3, 4, 5, 6, 7, 8, 9, 10),
17  (11, 12, 13, 14, 15, 16, 17, 18, 19, 20),
18  (21, 22, 23, 24, 25, 26, 27, 28, 29, 30),
19  (31, 32, 33, 34, 35, 36, 37, 38, 39, 40),
20  (41, 42, 43, 44, 45, 46, 47, 48, 49, 50),
21  (51, 52, 53, 54, 55, 56, 57, 58, 59, 60),
22  (61, 62, 63, 64, 65, 66, 67, 68, 69, 70),
23  (71, 72, 73, 74, 75, 76, 77, 78, 79, 80),
24  (81, 82, 83, 84, 85, 86, 87, 88, 89, 90),
25  (91, 92, 93, 94, 95, 96, 97, 98, 99, 100)
26  );
27  signal scrambling_in     : matrix(0 to filas-1, 0 to
       columnas-1);
28  signal shuffling_out    : matrix(0 to filas-1, 0 to columnas
       -1);
29  signal scrambling_out     : matrix(0 to filas-1, 0 to
       columnas-1);
30  signal done_sh    : std_logic:='1';
31  signal done_scr    : std_logic:='1';
32  signal K1_scr : byte_array(0 to 3) := (19,72,29,12);
33  signal K2_scr : byte_array(0 to 3) := (129,93,123,233);
34  signal K3_scr : byte_array(0 to 3) := (32,35,76,86);
35  signal K4_scr : byte_array(0 to 3) := (54,78,66,92);
36  signal C1_scr : byte_array(0 to 3) := (23,54,29,32);
37  signal C2_scr : byte_array(0 to 3) := (176,193,183,2);
38  signal S1_scr : byte_array(0 to 3) := (87,77,89,32);
39  signal S2_scr : byte_array(0 to 3) := (53,75,33,45);
40  signal S3_scr : byte_array(0 to 3) := (14,44,67,76);
41 begin
42  clk_proc: process
43  begin
44   clk <= '0'; wait for 5 ns;
45   clk <= '1'; wait for 5 ns;
46  end process;
```

```
47  U1: entity work.shuffling
48  generic map ( filas => filas, columnas => columnas )
49  port map (
50   clk         => clk,
51   reset       => reset_sh,
52   matrix_in   => matrix_in,
53   matrix_out  => shuffling_out,
54   done        => done_sh
55  );
56
57  U2: entity work.scrambling
58  generic map ( filas => filas, columnas => columnas )
59  port map (
60   clk         => clk,
61   reset       => reset_scr,
62   matrix_in   => scrambling_in,
63   K1 => K1_scr,
64   K2 => K2_scr,
65   K3 => K3_scr,
66   K4 => K4_scr,
67   C1 => C1_scr,
68   C2 => C2_scr,
69   S1 => S1_scr,
70   S2 => S2_scr,
71   S3 => S3_scr,
72   matrix_out => scrambling_out,
73   scrambled_ready        => done_scr
74  );
75
76  stim_proc: process
77  begin
78   reset_sh <= '1'; reset_scr <= '1';
79   reset_sh <= '0';
80   wait until done_sh = '1';
81   for i in 0 to filas-1 loop
82    for j in 0 to columnas-1 loop
83     scrambling_in(i,j) <= shuffling_out(i,j);
84    end loop;
85   end loop;
86   reset_scr <= '0';
87   wait until done_scr = '1';
88  end process;
89 end architecture sim;
```

Código 12: Ofuscación

5.3.0.5.2 Desofuscación

La prueba de desofuscación analiza el proceso inverso, ejecutando de forma secuencial los módulos *Deshuffling* y *Descrambling*. El objetivo es comprobar que el sistema es completamente reversible, asegurando la recuperación exacta de los datos iniciales.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use work.tipo.all;
5
6  entity desofuscation_tb is
7  end entity;
8
9  architecture sim of desofuscation_tb is
10  constant filas    : integer := 10;
11  constant columnas : integer := 10;
12  signal clk        : std_logic := '0';
13  signal reset_descr   : std_logic := '1';
14  signal reset_desh    : std_logic := '1';
15  signal matrix_in  : matrix(0 to 9, 0 to 9) := (
16  (1, 2, 3, 4, 5, 6, 7, 8, 9, 10),
17  (11, 12, 13, 14, 15, 16, 17, 18, 19, 20),
18  (21, 22, 23, 24, 25, 26, 27, 28, 29, 30),
19  (31, 32, 33, 34, 35, 36, 37, 38, 39, 40),
20  (41, 42, 43, 44, 45, 46, 47, 48, 49, 50),
21  (51, 52, 53, 54, 55, 56, 57, 58, 59, 60),
22  (61, 62, 63, 64, 65, 66, 67, 68, 69, 70),
23  (71, 72, 73, 74, 75, 76, 77, 78, 79, 80),
24  (81, 82, 83, 84, 85, 86, 87, 88, 89, 90),
25  (91, 92, 93, 94, 95, 96, 97, 98, 99, 100)
26  );
27
28  signal deshuffling_in     : matrix(0 to filas-1, 0 to
       columnas-1);
29  signal matrix_out    : matrix(0 to filas-1, 0 to columnas-1)
       ;
```

```
30  signal descrambling_out     : matrix(0 to filas-1, 0 to
       columnas-1);
31  signal done_descr    : std_logic:='1';
32  signal done_desh    : std_logic :='1';
33  signal K1_descr : byte_array(0 to 3) := (19,72,29,12);
34  signal K2_descr : byte_array(0 to 3) := (129,93,123,233);
35  signal K3_descr : byte_array(0 to 3) := (32,35,76,86);
36  signal K4_descr : byte_array(0 to 3) := (54,78,66,92);
37  signal C1_descr : byte_array(0 to 3) := (23,54,29,32);
38  signal C2_descr : byte_array(0 to 3) := (176,193,183,2);
39  signal S1_descr : byte_array(0 to 3) := (87,77,89,32);
40  signal S2_descr : byte_array(0 to 3) := (53,75,33,45);
41  signal S3_descr : byte_array(0 to 3) := (14,44,67,76);
42
43 begin
44  clk_proc: process
45  begin
46   clk <= '0'; wait for 5 ns;
47   clk <= '1'; wait for 5 ns;
48  end process;
49
50  U1: entity work.descrambling
51  generic map ( filas => filas, columnas => columnas )
52  port map (
53   clk         => clk,
54   reset       => reset_descr,
55   matrix_in   => matrix_in,
56   K1 => K1_descr,
57   K2 => K2_descr,
58   K3 => K3_descr,
59   K4 => K4_descr,
60   C1 => C1_descr,
61   C2 => C2_descr,
62   S1 => S1_descr,
63   S2 => S2_descr,
64   S3 => S3_descr,
65   matrix_out => descrambling_out,
66   ready        => done_descr
67  );
68
69  U2: entity work.deshuffling
70  generic map ( filas => filas, columnas => columnas )
71  port map (
```

```
72    clk         => clk,
73    reset       => reset_desh,
74    matrix_in   => deshuffling_in,
75    matrix_out  => matrix_out,
76    done        => done_desh
77   );
78
79   stim_proc: process
80   begin
81    reset_descr <= '1'; reset_desh <= '1';
82    reset_descr <= '0';
83    wait until done_descr = '1';
84    for i in 0 to filas-1 loop
85     for j in 0 to columnas-1 loop
86      deshuffling_in(i,j) <= descrambling_out(i,j);
87     end loop;
88    end loop;
89    reset_desh <= '0';
90    wait until done_desh = '1';
91    wait;
92   end process;
93  end architecture sim;
```

Código 13: Desofuscación

5.3.1. Escenarios de prueba

Finalmente, se ha diseñado un escenario de prueba integrado en el que se incluyen todos los módulos del sistema. El módulo superior (*Top Module*) coordina la ejecución completa de las operaciones de barajado, transformación, desbarajado y descrifrado.
Esta simulación global permite verificar la comunicación entre bloques, el orden de ejecución y la consistencia de las señales de control y datos.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use work.tipo.all;
5
6  entity top_module_tb is
7  end entity;
8
```

```
9  architecture sim of top_module_tb is
10
11  constant filas    : integer := 10;
12  constant columnas : integer := 10;
13  signal clk        : std_logic := '0';
14  signal reset_sh   : std_logic := '1';
15  signal reset_scr   : std_logic := '1';
16  signal reset_descr   : std_logic := '1';
17  signal reset_desh   : std_logic := '1';
18  signal matrix_in  : matrix(0 to 9, 0 to 9) := (
19  (1, 2, 3, 4, 5, 6, 7, 8, 9, 10),
20  (11, 12, 13, 14, 15, 16, 17, 18, 19, 20),
21  (21, 22, 23, 24, 25, 26, 27, 28, 29, 30),
22  (31, 32, 33, 34, 35, 36, 37, 38, 39, 40),
23  (41, 42, 43, 44, 45, 46, 47, 48, 49, 50),
24  (51, 52, 53, 54, 55, 56, 57, 58, 59, 60),
25  (61, 62, 63, 64, 65, 66, 67, 68, 69, 70),
26  (71, 72, 73, 74, 75, 76, 77, 78, 79, 80),
27  (81, 82, 83, 84, 85, 86, 87, 88, 89, 90),
28  (91, 92, 93, 94, 95, 96, 97, 98, 99, 100)
29  );
30  signal scrambling_in     : matrix(0 to filas-1, 0 to
       columnas-1);
31  signal descrambling_in     : matrix(0 to filas-1, 0 to
       columnas-1);
32  signal deshuffling_in     : matrix(0 to filas-1, 0 to
       columnas-1);
33  signal matrix_out    : matrix(0 to filas-1, 0 to columnas-1)
       ;
34  signal shuffling_out    : matrix(0 to filas-1, 0 to columnas
       -1);
35  signal scrambling_out     : matrix(0 to filas-1, 0 to
       columnas-1);
36  signal descrambling_out     : matrix(0 to filas-1, 0 to
       columnas-1);
37  signal done_sh    : std_logic:='1';
38  signal done_scr    : std_logic:='1';
39  signal done_descr    : std_logic:='1';
40  signal done_desh    : std_logic :='1';
41  signal K1_scr : byte_array(0 to 3) := (19,72,29,12);
42  signal K2_scr : byte_array(0 to 3) := (129,93,123,233);
43  signal K3_scr : byte_array(0 to 3) := (32,35,76,86);
44  signal K4_scr : byte_array(0 to 3) := (54,78,66,92);
```

```
45  signal C1_scr : byte_array(0 to 3) := (23,54,29,32);
46  signal C2_scr : byte_array(0 to 3) := (176,193,183,2);
47  signal S1_scr : byte_array(0 to 3) := (87,77,89,32);
48  signal S2_scr : byte_array(0 to 3) := (53,75,33,45);
49  signal S3_scr : byte_array(0 to 3) := (14,44,67,76);
50  signal K1_descr : byte_array(0 to 3) := (19,72,29,12);
51  signal K2_descr : byte_array(0 to 3) := (129,93,123,233);
52  signal K3_descr : byte_array(0 to 3) := (32,35,76,86);
53  signal K4_descr : byte_array(0 to 3) := (54,78,66,92);
54  signal C1_descr : byte_array(0 to 3) := (23,54,29,32);
55  signal C2_descr : byte_array(0 to 3) := (176,193,183,2);
56  signal S1_descr : byte_array(0 to 3) := (87,77,89,32);
57  signal S2_descr : byte_array(0 to 3) := (53,75,33,45);
58  signal S3_descr : byte_array(0 to 3) := (14,44,67,76);
59
60 begin
61
62  clk_proc: process
63  begin
64   clk <= '0';
65   wait for 5 ns;
66   clk <= '1';
67   wait for 5 ns;
68  end process;
69
70  U1: entity work.shuffling
71  generic map ( filas => filas, columnas => columnas )
72  port map (
73   clk         => clk,
74   reset       => reset_sh,
75   matrix_in   => matrix_in,
76   matrix_out  => shuffling_out,
77   done        => done_sh
78  );
79
80  U2: entity work.scrambling
81  generic map ( filas => filas, columnas => columnas )
82  port map (
83   clk         => clk,
84   reset       => reset_scr,
85   matrix_in   => scrambling_in,
86   K1 => K1_scr,
87   K2 => K2_scr,
```

```
88   K3 => K3_scr,
89   K4 => K4_scr,
90   C1 => C1_scr,
91   C2 => C2_scr,
92   S1 => S1_scr,
93   S2 => S2_scr,
94   S3 => S3_scr,
95   matrix_out => scrambling_out,
96   scrambled_ready       => done_scr
97  );
98
99  U3: entity work.descrambling
100 generic map ( filas => filas, columnas => columnas )
101 port map (
102  clk         => clk,
103  reset       => reset_descr,
104  matrix_in   => descrambling_in,
105  K1 => K1_descr,
106  K2 => K2_descr,
107  K3 => K3_descr,
108  K4 => K4_descr,
109  C1 => C1_descr,
110  C2 => C2_descr,
111  S1 => S1_descr,
112  S2 => S2_descr,
113  S3 => S3_descr,
114  matrix_out => descrambling_out,
115  ready        => done_descr
116 );
117
118 U4: entity work.deshuffling
119 generic map ( filas => filas, columnas => columnas )
120 port map (
121  clk         => clk,
122  reset       => reset_desh,
123  matrix_in   => deshuffling_in,
124  matrix_out => matrix_out,
125  done        => done_desh
126 );
127
128 stim_proc: process
129 begin
130
```

```
131   reset_sh <= '1';
132   reset_scr <= '1';
133   reset_descr <= '1';
134   reset_desh <= '1';
135   reset_sh <= '0';
136   wait until done_sh = '1';
137
138   for i in 0 to filas-1 loop
139    for j in 0 to columnas-1 loop
140     scrambling_in(i,j) <= shuffling_out(i,j);
141    end loop;
142   end loop;
143
144   reset_scr <= '0';
145
146   wait until done_scr = '1';
147
148
149
150   for i in 0 to filas-1 loop
151    for j in 0 to columnas-1 loop
152     descrambling_in(i,j) <= scrambling_out(i,j);
153    end loop;
154   end loop;
155
156   reset_descr <= '0';
157   wait until done_descr = '1';
158
159   for i in 0 to filas-1 loop
160    for j in 0 to columnas-1 loop
161     deshuffling_in(i,j) <= descrambling_out(i,j);
162    end loop;
163   end loop;
164
165   reset_desh <= '0';
166   wait until done_desh = '1';
167
168  end process;
169
170 end architecture sim;
```

Código 14: Top Module

Capítulo 6

Resultados y Discusión

6.1. Resultados de validación funcional

En esta sección se presentan los resultados obtenidos en la simulación funcional de los módulos desarrollados: *Shuffling*, *Scrambling*, *Deshuffling* y *Descrambling*. El objetivo de esta validación es comprobar que cada bloque cumple con la función descrita en su diseño, verificando la correspondencia entre las matrices de entrada y salida y asegurando la coherencia de las señales de control.

Las simulaciones se han realizado en **Xilinx ISE Design Suite 14.7**, utilizando los *testbench* diseñados específicamente para cada módulo expuestos en el capítulo anterior.

6.1.1. Resultados del módulo *Shuffling*

El módulo *Shuffling* recibe una matriz de entrada con valores organizados de forma secuencial y produce una matriz de salida cuyos elementos se encuentran reorganizados según un patrón pseudoaleatorio generado por el mapa caótico definido en el diseño.

Las Tablas 1 y 2 recogen un ejemplo de las matrices de entrada y salida utilizadas en la validación. Como puede apreciarse, los valores de la matriz de salida corresponden exactamente a los de la entrada, pero dispuestos en posiciones distintas, lo que confirma la correcta permutación sin pérdida ni duplicación de datos.

Filas	**Columnas**									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	1	2	3	4	5	6	7	8	9	10
1	11	12	13	14	15	16	17	18	19	20
2	21	22	23	24	25	26	27	28	29	30
3	31	32	33	34	35	36	37	38	39	40
4	41	42	43	44	45	46	47	48	49	50
5	51	52	53	54	55	56	57	58	59	60
6	61	62	63	64	65	66	67	68	69	70
7	71	72	73	74	75	76	77	78	79	80
8	81	82	83	84	85	86	87	88	89	90
9	91	92	93	94	95	96	97	98	99	100

Tabla 1: Matriz entrada en Shuffling

Filas	**Columnas**									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	51	35	31	53	34	1	18	37	42	56
1	59	8	27	4	22	71	58	7	17	12
2	52	62	61	6	15	16	36	43	83	92
3	57	23	32	24	26	81	76	82	91	14
4	3	12	41	25	13	100	48	90	5	54
5	77	66	21	33	11	98	86	40	99	68
6	97	20	29	55	30	85	87	49	88	74
7	19	28	38	45	95	96	65	10	9	79
8	69	94	64	73	60	89	67	84	63	72
9	75	39	44	93	70	47	78	50	46	80

Tabla 2: Matriz salida en Shuffling

El análisis confirma que el módulo cumple su función de dispersión, generando una salida completamente reordenada sin alterar los valores numéricos originales.

6.1.2. Resultados del módulo *Scrambling*

El módulo *Scrambling* transforma la matriz de entrada aplicando una combinación de operaciones XOR, sumas módulo 255 y sustituciones *S-box*.

Las Tablas 3 y 4 muestran un ejemplo de las matrices de entrada y salida. Se observa un cambio significativo en los valores numéricos, consecuencia directa de las

transformaciones no lineales aplicadas, lo que incrementa la entropía y la confusión de los datos.

Filas	**Columnas**									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	1	2	3	4	5	6	7	8	9	10
1	11	12	13	14	15	16	17	18	19	20
2	21	22	23	24	25	26	27	28	29	30
3	31	32	33	34	35	36	37	38	39	40
4	41	42	43	44	45	46	47	48	49	50
5	51	52	53	54	55	56	57	58	59	60
6	61	62	63	64	65	66	67	68	69	70
7	71	72	73	74	75	76	77	78	79	80
8	81	82	83	84	85	86	87	88	89	90
9	91	92	93	94	95	96	97	98	99	100

Tabla 3: Matriz entrada en Scrambling

Filas	**Columnas**									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	201	214	114	48	0	179	88	52	235	131
1	229	253	123	147	41	74	96	132	69	84
2	38	106	252	27	171	113	111	235	149	158
3	194	147	124	124	168	64	5	159	128	54
4	243	133	64	254	173	77	171	164	117	114
5	81	188	141	127	41	62	118	146	182	208
6	229	48	147	208	161	236	191	90	77	77
7	43	114	190	119	177	9	162	123	71	81
8	67	212	64	182	57	218	250	250	97	197
9	183	213	127	202	164	235	56	32	243	27

Tabla 4: Matriz salida en Scrambling

Los resultados demuestran que el módulo introduce un alto grado de ofuscación, cumpliendo su función de dispersión y mezcla sin modificar la estructura de la matriz.

6.1.3. Resultados del módulo *Deshuffling*

El módulo *Deshuffling* invierte el proceso realizado por el *Shuffling*, restituyendo los elementos a sus posiciones originales.

Las Tablas 5 y 6 recogen un ejemplo de entrada y salida utilizadas en la validación funcional del módulo. Como se puede apreciar, los valores de ambas tablas son iguales pero dispuestos en posiciones distintas, lo que confirma la permutación de la matriz sin pérdida ni duplicación de los datos de la matriz de entrada.

Filas	**Columnas**									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	1	2	3	4	5	6	7	8	9	10
1	11	12	13	14	15	16	17	18	19	20
2	21	22	23	24	25	26	27	28	29	30
3	31	32	33	34	35	36	37	38	39	40
4	41	42	43	44	45	46	47	48	49	50
5	51	52	53	54	55	56	57	58	59	60
6	61	62	63	64	65	66	67	68	69	70
7	71	72	73	74	75	76	77	78	79	80
8	81	82	83	84	85	86	87	88	89	90
9	91	92	93	94	95	96	97	98	99	100

Tabla 5: Matriz entrada en Deshuffling

Filas	**Columnas**									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	6	20	41	14	49	24	18	12	79	78
1	55	42	45	40	25	26	19	7	71	62
2	53	15	32	34	44	35	13	72	63	65
3	3	33	54	5	2	27	8	73	92	58
4	43	9	28	93	74	99	96	47	68	98
5	1	21	4	50	64	10	31	17	11	85
6	23	22	89	83	77	52	87	60	81	95
7	16	90	84	70	91	37	51	97	80	100
8	36	38	29	88	66	57	67	69	86	48
9	39	30	94	82	75	76	61	56	59	46

Tabla 6: Matriz salida en Deshuffling

6.1.4. Resultados del módulo *Descrambling*

Finalmente, el módulo *Descrambling* ejecuta las operaciones inversas del proceso de *Scrambling*, restaurando los valores originales de la matriz. Durante la simulación se verifica que la tabla inversa *Inv-S-box* y las operaciones XOR realizan un cambio significativo sobre los valores númericos como se observa en las tablas 5 y 6.

Filas	**Columnas**									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	1	2	3	4	5	6	7	8	9	10
1	11	12	13	14	15	16	17	18	19	20
2	21	22	23	24	25	26	27	28	29	30
3	31	32	33	34	35	36	37	38	39	40
4	41	42	43	44	45	46	47	48	49	50
5	51	52	53	54	55	56	57	58	59	60
6	61	62	63	64	65	66	67	68	69	70
7	71	72	73	74	75	76	77	78	79	80
8	81	82	83	84	85	86	87	88	89	90
9	91	92	93	94	95	96	97	98	99	100

Tabla 7: Matriz entrada en Descrambling

Filas	**Columnas**									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	26	34	200	60	97	232	97	159	117	232
1	191	172	253	251	184	48	98	100	249	114
2	25	177	197	104	153	117	89	228	110	40
3	124	86	91	183	126	240	209	107	32	226
4	27	216	82	98	207	136	11	37	32	141
5	37	100	88	121	201	159	109	236	11	49
6	156	231	56	82	72	55	211	132	72	187
7	90	130	183	20	209	81	50	251	203	76
8	69	3	113	208	227	149	153	18	148	27
9	44	78	187	211	198	204	207	157	29	172

Tabla 8: Matriz salida en Descrambling

6.2. Pruebas de reversibilidad

Las pruebas de reversibilidad tienen como objetivo comprobar que las transformaciones aplicadas por los módulos desarrollados son completamente reversibles y no introducen pérdida de información en el proceso. Cada par de módulos complementarios (*Shuffling–Deshuffling* y *Scrambling–Descrambling*) ha sido verificado mediante simulaciones funcionales consecutivas, aplicando en primer lugar la transformación directa y posteriormente su correspondiente inversa.

El resultado esperado en ambos casos es que la matriz de salida final coincida exactamente con la matriz original de entrada.

6.2.1. *Shuffling – Deshuffling*

En esta prueba se ha validado el comportamiento conjunto de los módulos *Shuffling* y *Deshuffling*. El procedimiento consiste en aplicar el algoritmo de barajado a una matriz de entrada y, a continuación, procesar la matriz resultante mediante el módulo inverso. Ambos módulos se ejecutan con los mismos parámetros de configuración (`semilla` y `r`) y con idéntico número de filas y columnas, garantizando la reproducibilidad del proceso.

Filas	**Columnas**									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	1	2	3	4	5	6	7	8	9	10
1	11	12	13	14	15	16	17	18	19	20
2	21	22	23	24	25	26	27	28	29	30
3	31	32	33	34	35	36	37	38	39	40
4	41	42	43	44	45	46	47	48	49	50
5	51	52	53	54	55	56	57	58	59	60
6	61	62	63	64	65	66	67	68	69	70
7	71	72	73	74	75	76	77	78	79	80
8	81	82	83	84	85	86	87	88	89	90
9	91	92	93	94	95	96	97	98	99	100

Tabla 9: Shuffling-Deshuffling: Matriz entrada en Shuffling

Las tablas intermedias 10 y 11 corresponden a la salida del módulo *Shuffling* y se utiliza como entrada para el módulo *Deshuffling*.

Filas	Columnas									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	56	98	57	64	78	52	94	32	71	22
1	77	96	15	65	1	53	72	47	63	33
2	81	73	46	68	92	76	12	27	42	67
3	99	50	74	44	70	45	62	21	43	26
4	91	97	2	88	20	51	75	41	58	86
5	95	23	40	6	30	61	93	79	84	90
6	55	38	60	19	36	11	37	7	31	82
7	14	39	54	69	5	89	13	35	8	100
8	48	18	34	9	28	80	16	66	85	4
9	59	10	49	87	29	3	17	24	83	25

Tabla 10: Shuffling-Deshuffling: Matriz salida en Shuffling

Filas	Columnas									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	56	98	57	64	78	52	94	32	71	22
1	77	96	15	65	1	53	72	47	63	33
2	81	73	46	68	92	76	12	27	42	67
3	99	50	74	44	70	45	62	21	43	26
4	91	97	2	88	20	51	75	41	58	86
5	95	23	40	6	30	61	93	79	84	90
6	55	38	60	19	36	11	37	7	31	82
7	14	39	54	69	5	89	13	35	8	100
8	48	18	34	9	28	80	16	66	85	4
9	59	10	49	87	29	3	17	24	83	25

Tabla 11: Shuffling-Deshuffling: Matriz entrada en Deshuffling

La comparación entre la tabla 9 y la tabla 12 restaurada demuestra la reversibilidad total del proceso, con coincidencia exacta entre todos los elementos.

Filas	Columnas									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	1	2	3	4	5	6	7	8	9	10
1	11	12	13	14	15	16	17	18	19	20
2	21	22	23	24	25	26	27	28	29	30
3	31	32	33	34	35	36	37	38	39	40
4	41	42	43	44	45	46	47	48	49	50
5	51	52	53	54	55	56	57	58	59	60
6	61	62	63	64	65	66	67	68	69	70
7	71	72	73	74	75	76	77	78	79	80
8	81	82	83	84	85	86	87	88	89	90
9	91	92	93	94	95	96	97	98	99	100

Tabla 12: Shuffling-Deshuffling: Matriz salida en Deshuffling

La simulación demuestra que, tras aplicar consecutivamente ambos módulos, la matriz final es idéntica a la original, confirmando que el proceso de barajado y desbarajado es perfectamente reversible. No se detectan errores en la posición ni en el valor de los elementos, y las señales de control se activan de forma secuencial y sincronizada. Este resultado valida la consistencia interna de la máquina de estados y la correcta reproducción del mapa caótico en ambos procesos.

6.2.2. *Scrambling – Descrambling*

La segunda prueba de reversibilidad se centra en los módulos *Scrambling* y *Descrambling*, encargados de aplicar y revertir las transformaciones no lineales sobre los datos. Al igual que en el caso anterior, el procedimiento consiste en procesar una matriz de entrada mediante el módulo *Scrambling* y someter la matriz resultante al módulo inverso *Descrambling*, utilizando las mismas claves y constantes auxiliares de entrada.

Filas	**Columnas**									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	1	2	3	4	5	6	7	8	9	10
1	11	12	13	14	15	16	17	18	19	20
2	21	22	23	24	25	26	27	28	29	30
3	31	32	33	34	35	36	37	38	39	40
4	41	42	43	44	45	46	47	48	49	50
5	51	52	53	54	55	56	57	58	59	60
6	61	62	63	64	65	66	67	68	69	70
7	71	72	73	74	75	76	77	78	79	80
8	81	82	83	84	85	86	87	88	89	90
9	91	92	93	94	95	96	97	98	99	100

Tabla 13: Scrambling-Descrambling: Matriz entrada en Scrambling

Las tablas 14 y 15 muestran los resultados obtenidos durante la simulación. La matriz intermedia (salida del módulo *Scrambling* y entrada del módulo *Descrambling*) presenta valores completamente transformados debido a las operaciones XOR, sumas módulo 255 y sustitución *S-box*.

Filas	**Columnas**									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	80	83	114	48	0	165	177	52	253	177
1	80	88	237	147	35	74	96	132	69	84
2	38	106	252	27	171	113	111	235	215	125
3	159	147	255	124	67	64	82	146	128	54
4	243	107	218	254	107	64	82	170	168	114
5	171	188	141	127	41	62	118	146	182	298
6	229	48	147	208	209	132	35	90	212	77
7	99	114	54	173	177	9	162	239	114	81
8	182	171	24	111	43	218	2	250	97	197
9	183	213	127	202	164	235	56	32	243	27

Tabla 14: Scrambling-Descrambling: Matriz salida en Scrambling

Filas	Columnas									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	80	83	114	48	0	165	177	52	253	177
1	80	88	237	147	35	74	96	132	69	84
2	38	106	252	27	171	113	111	235	215	125
3	159	147	255	124	67	64	82	146	128	54
4	243	107	218	254	107	64	82	170	168	114
5	171	188	141	127	41	62	118	146	182	298
6	229	48	147	208	209	132	35	90	212	77
7	99	114	54	173	177	9	162	239	114	81
8	182	171	24	111	43	218	2	250	97	197
9	183	213	127	202	164	235	56	32	243	27

Tabla 15: Scrambling-Descrambling: Matriz entrada en Descrambling

Al aplicar posteriormente el módulo *Descrambling*, se recuperan exactamente los valores originales como podemos apreciar al comparar las tablas 13 y 16, demostrando la reversión completa de las transformaciones no lineales.

Filas	Columnas									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	1	2	3	4	5	6	7	8	9	10
1	11	12	13	14	15	16	17	18	19	20
2	21	22	23	24	25	26	27	28	29	30
3	31	32	33	34	35	36	37	38	39	40
4	41	42	43	44	45	46	47	48	49	50
5	51	52	53	54	55	56	57	58	59	60
6	61	62	63	64	65	66	67	68	69	70
7	71	72	73	74	75	76	77	78	79	80
8	81	82	83	84	85	86	87	88	89	90
9	91	92	93	94	95	96	97	98	99	100

Tabla 16: Scrambling-Descrambling: Matriz salida en Descrambling

La reversibilidad total observada en la simulación confirma que las operaciones aritméticas y lógicas han sido correctamente implementadas en los módulos *Scrambling* y *Descrambling*. No se presentan pérdidas ni alteraciones en los datos, y las señales se activan de forma sincronizada, garantizando un flujo de control estable.

6.3. Análisis de propagación entre módulos

El objetivo de esta sección es verificar la **coherencia del flujo de datos** y la **sincronización de señales de control** cuando los módulos se conectan de forma secuencial. Concretamente, se valida: (i) que la salida de *Shuffling* se propaga correctamente como entrada de *Scrambling* en el escenario de *Ofuscación*; (ii) que la salida de *Descrambling* se utiliza como entrada de *Deshuffling* en el escenario de *Desofuscación*; y (iii) que la cadena completa mantiene la **integridad de la información** y un **orden temporal** correcto sin solapamientos indebidos ni pérdidas de datos.

6.3.1. Ofuscación

La ofuscación corresponde a la tubería *Shuffling* → *Scrambling*. En la Tabla 17 se muestra la matriz original que alimenta al módulo *Shuffling*. Tras el barajado pseudoaleatorio, la salida de *Shuffling* (Tabla 18) constituye *exactamente* la entrada del módulo *Scrambling* (Tabla 19), lo que confirma la **propagación correcta sin alteraciones intermedias**. Finalmente, la Tabla 20 recoge la salida del *Scrambling* tras aplicar las operaciones XOR, suma módulo 255 y sustitución S-box, evidenciando una **transformación no lineal** con alta dispersión respecto a la entrada.

Desde el punto de vista temporal, se ha verificado que la señal de fin de *Shuffling* (`done`) **precede** a la activación del proceso en *Scrambling* y que la señal `scrambled_ready` indica la disponibilidad de la matriz ofuscada. No se observan desincronías ni dependencias cíclicas entre etapas. En suma, las Tablas 17–20 confirman que la salida de *Shuffling* se consume íntegramente por *Scrambling* y que la ofuscación resultante mantiene la dimensionalidad y la consistencia de los datos.

Filas	**Columnas**									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	1	2	3	4	5	6	7	8	9	10
1	11	12	13	14	15	16	17	18	19	20
2	21	22	23	24	25	26	27	28	29	30
3	31	32	33	34	35	36	37	38	39	40
4	41	42	43	44	45	46	47	48	49	50
5	51	52	53	54	55	56	57	58	59	60
6	61	62	63	64	65	66	67	68	69	70
7	71	72	73	74	75	76	77	78	79	80
8	81	82	83	84	85	86	87	88	89	90
9	91	92	93	94	95	96	97	98	99	100

Tabla 17: Ofuscación: Matriz entrada en Shuffling

Filas	**Columnas**									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	30	86	90	28	87	20	3	84	79	25
1	22	13	94	17	99	40	23	14	4	19
2	29	49	50	15	6	5	85	78	68	59
3	24	98	89	97	95	70	35	69	60	7
4	18	9	80	96	8	51	73	61	16	27
5	34	45	100	88	10	53	65	81	52	43
6	54	1	92	26	91	66	64	72	63	37
7	2	93	83	76	56	55	46	11	12	32
8	42	57	47	38	21	62	44	67	48	39
9	36	82	77	58	41	74	33	71	75	31

Tabla 18: Ofuscación: Matriz salida en Shuffling

Filas	Columnas									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	30	86	90	28	87	20	3	84	79	25
1	22	13	94	17	99	40	23	14	4	19
2	29	49	50	15	6	5	85	78	68	59
3	24	98	89	97	95	70	35	69	60	7
4	18	9	80	96	8	51	73	61	16	27
5	34	45	100	88	10	53	65	81	52	43
6	54	1	92	26	91	66	64	72	63	37
7	2	93	83	76	56	55	46	11	12	32
8	42	57	47	38	21	62	44	67	48	39
9	36	82	77	58	41	74	33	71	75	31

Tabla 19: Ofuscación: Matriz entrada en Scrambling

Filas	Columnas									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	125	82	164	254	91	227	203	183	150	179
1	253	216	171	113	26	41	12	76	33	30
2	5	47	44	74	240	5	164	183	231	120
3	169	119	183	38	41	177	125	82	164	254
4	91	227	203	183	150	179	253	216	171	113
5	26	41	12	76	33	30	5	47	44	74
6	240	5	164	183	231	120	169	119	183	38
7	41	177	125	82	164	254	91	227	203	183
8	150	179	253	216	171	113	26	41	12	76
9	33	30	5	47	44	74	240	5	164	183

Tabla 20: Ofuscación: Matriz salida en Scrambling

6.3.2. Desofuscación

La desofuscación corresponde a la tubería inversa *Descrambling* → *Deshuffling*. La Tabla 21 muestra la matriz que se introduce en *Descrambling*; la salida obtenida (Tabla 22) se emplea como **entrada directa** del módulo *Deshuffling* (Tabla 23), comprobándose de nuevo la **propagación exacta** entre etapas. La Tabla 24 recoge la salida final, donde se observa el barajado aplicado para reordenar la matriz.

A nivel de control, la señal `ready` del *Descrambling* habilita la captura de datos por parte de *Deshuffling*, y la transición concluye con la activación de `done`. El encadenado confirma que **no se pierde información** y que ambas inversiones (sustitución inversa y desbarajado) se aplican en el **orden correcto**. Las Tablas 21–24 evidencian, además, que no hay cambios espurios entre la salida de un módulo y la entrada del siguiente.

Filas	**Columnas**									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	1	2	3	4	5	6	7	8	9	10
1	11	12	13	14	15	16	17	18	19	20
2	21	22	23	24	25	26	27	28	29	30
3	31	32	33	34	35	36	37	38	39	40
4	41	42	43	44	45	46	47	48	49	50
5	51	52	53	54	55	56	57	58	59	60
6	61	62	63	64	65	66	67	68	69	70
7	71	72	73	74	75	76	77	78	79	80
8	81	82	83	84	85	86	87	88	89	90
9	91	92	93	94	95	96	97	98	99	100

Tabla 21: Desofuscación: Matriz entrada en Descrambling

Filas	**Columnas**									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	26	34	200	60	97	232	97	159	117	232
1	191	172	253	251	184	48	98	100	249	114
2	25	177	197	104	153	117	89	228	110	40
3	124	86	91	183	126	240	209	107	32	226
4	27	216	82	98	207	136	111	37	32	141
5	37	100	88	121	201	159	109	236	11	49
6	156	231	56	82	72	55	211	132	72	187
7	90	130	183	20	209	81	50	251	203	76
8	69	3	113	208	227	149	153	18	148	27
9	44	78	187	211	198	204	207	157	29	172

Tabla 22: Desofuscación: Matriz salida en Descrambling

Filas	**Columnas**									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	26	34	200	60	97	232	97	159	117	232
1	191	172	253	251	184	48	98	100	249	114
2	25	177	197	104	153	117	89	228	110	40
3	124	86	91	183	126	240	209	107	32	226
4	27	216	82	98	207	136	111	37	32	141
5	37	100	88	121	201	159	109	236	11	49
6	156	231	56	82	72	55	211	132	72	187
7	90	130	183	20	209	81	50	251	203	76
8	69	3	113	208	227	149	153	18	148	27
9	44	78	187	211	198	204	207	157	29	172

Tabla 23: Desofuscación: Matriz entrada en Deshuffling

Filas	**Columnas**									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	184	82	204	27	209	121	132	203	208	78
1	55	89	50	90	253	153	207	3	82	207
2	107	232	100	157	172	226	228	227	198	201
3	72	159	114	113	251	72	211	231	130	88
4	37	110	32	183	240	197	100	69	187	86
5	136	232	48	183	156	26	200	32	44	56
6	159	209	249	60	251	18	40	104	20	126
7	117	98	177	91	111	117	191	97	236	149
8	25	187	29	11	148	141	211	98	81	49
9	27	153	109	97	37	172	216	34	124	76

Tabla 24: Desofuscación: Matriz salida en Deshuffling

6.3.3. Módulo completo

Para completar el análisis, se valida la **cadena global** en dos tramos consecutivos: primero *Shuffling* → *Scrambling* y, a continuación, *Descrambling* → *Deshuffling*. La Tabla 25 presenta la matriz inicial; la Tabla 26 es la salida de *Shuffling*, que se reintroduce sin modificaciones como entrada de *Scrambling* (Tabla 27) y produce la matriz ofuscada final del primer tramo (Tabla 28). Posteriormente, esta última matriz alimenta el proceso inverso: entrada a *Descrambling* (Tabla 29) y salida restaurada (Tabla 30), que a su vez se utiliza como entrada a *Deshuffling* (Tabla 31).

El tramo concluye con la salida de *Deshuffling* (Tabla 32), que **coincide con la matriz original**.

El seguimiento de las señales `done`, `scrambled_ready` y `ready` a lo largo de la simulación muestra un **encadenamiento sin conflictos**: cada etapa comienza únicamente cuando la anterior ha indicado que su salida es válida, evitando condiciones de carrera o lecturas prematuras. En conjunto, los resultados confirman que la propagación entre módulos es **determinista, estable y libre de pérdidas**, y que la cadena completa preserva la integridad de los datos desde la entrada inicial hasta la reconstrucción final.

Filas	**Columnas**									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	1	2	3	4	5	6	7	8	9	10
1	11	12	13	14	15	16	17	18	19	20
2	21	22	23	24	25	26	27	28	29	30
3	31	32	33	34	35	36	37	38	39	40
4	41	42	43	44	45	46	47	48	49	50
5	51	52	53	54	55	56	57	58	59	60
6	61	62	63	64	65	66	67	68	69	70
7	71	72	73	74	75	76	77	78	79	80
8	81	82	83	84	85	86	87	88	89	90
9	91	92	93	94	95	96	97	98	99	100

Tabla 25: Módulo completo: Matriz entrada en Shuffling

Filas	**Columnas**									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	81	35	31	83	34	91	18	37	42	86
1	89	98	27	94	22	1	88	97	17	92
2	82	72	71	96	15	16	36	43	53	62
3	87	23	32	24	26	51	6	52	61	14
4	93	12	41	25	13	70	48	60	95	84
5	7	76	21	33	11	68	56	40	69	78
6	67	20	29	85	30	55	57	49	58	4
7	19	28	38	45	65	66	75	100	99	9
8	79	64	74	3	90	59	77	54	73	2
9	5	39	44	63	80	47	8	50	46	10

Tabla 26: Módulo completo: Matriz salida en Shuffling

Filas	**Columnas**									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	81	35	31	83	34	91	18	37	42	86
1	89	98	27	94	22	1	88	97	17	92
2	82	72	71	96	15	16	36	43	53	62
3	87	23	32	24	26	51	6	52	61	14
4	93	12	41	25	13	70	48	60	95	84
5	7	76	21	33	11	68	56	40	69	78
6	67	20	29	85	30	55	57	49	58	4
7	19	28	38	45	65	66	75	100	99	9
8	79	64	74	3	90	59	77	54	73	2
9	5	39	44	63	80	47	8	50	46	10

Tabla 27: Módulo completo: Matriz entrada en Scrambling

Filas	**Columnas**									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	125	82	164	254	91	227	203	183	150	179
1	253	216	171	113	26	41	12	76	33	30
2	5	47	44	74	240	5	164	183	231	120
3	169	119	183	38	41	177	125	82	164	254
4	91	227	203	183	150	179	253	216	171	113
5	26	41	12	76	33	30	5	47	44	74
6	240	5	164	183	231	120	169	119	183	38
7	41	177	125	82	164	254	91	227	203	183
8	150	179	253	216	171	113	26	41	12	76
9	33	30	4	47	44	74	240	5	164	183

Tabla 28: Módulo completo: Matriz salida en Scrambling

Filas	Columnas									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	125	82	164	254	91	227	203	183	150	179
1	253	216	171	113	26	41	12	76	33	30
2	5	47	44	74	240	5	164	183	231	120
3	169	119	183	38	41	177	125	82	164	254
4	91	227	203	183	150	179	253	216	171	113
5	26	41	12	76	33	30	5	47	44	74
6	240	5	164	183	231	120	169	119	183	38
7	41	177	125	82	164	254	91	227	203	183
8	150	179	253	216	171	113	26	41	12	76
9	33	30	4	47	44	74	240	5	164	183

Tabla 29: Módulo completo: Matriz entrada en Descrambling

Filas	Columnas									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	81	35	31	83	34	91	18	37	42	86
1	89	98	27	94	22	1	88	97	17	92
2	82	72	71	96	15	16	36	43	53	62
3	87	23	32	24	26	51	6	52	61	14
4	93	12	41	25	13	70	48	60	95	84
5	7	76	21	33	11	68	56	40	69	78
6	67	20	29	85	30	55	57	49	58	4
7	19	28	38	45	65	66	75	100	99	9
8	79	64	74	3	90	59	77	54	73	2
9	5	39	44	63	80	47	8	50	46	10

Tabla 30: Módulo completo: Matriz salida en Descrambling

Filas	Columnas									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	81	35	31	83	34	91	18	37	42	86
1	89	98	27	94	22	1	88	97	17	92
2	82	72	71	96	15	16	36	43	53	62
3	87	23	32	24	26	51	6	52	61	14
4	93	12	41	25	13	70	48	60	95	84
5	7	76	21	33	11	68	56	40	69	78
6	67	20	29	85	30	55	57	49	58	4
7	19	28	38	45	65	66	75	100	99	9
8	79	64	74	3	90	59	77	54	73	2
9	5	39	44	63	80	47	8	50	46	10

Tabla 31: Módulo completo: Matriz entrada en Deshuffling

Filas	Columnas									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	1	2	3	4	5	6	7	8	9	10
1	11	12	13	14	15	16	17	18	19	20
2	21	22	23	24	25	26	27	28	29	30
3	31	32	33	34	35	36	37	38	39	40
4	41	42	43	44	45	46	47	48	49	50
5	51	52	53	54	55	56	57	58	59	60
6	61	62	63	64	65	66	67	68	69	70
7	71	72	73	74	75	76	77	78	79	80
8	81	82	83	84	85	86	87	88	89	90
9	91	92	93	94	95	96	97	98	99	100

Tabla 32: Módulo completo: Matriz salida en Deshuffling

6.4. Discusión de resultados

El conjunto de simulaciones realizadas ha permitido validar el correcto funcionamiento de los módulos desarrollados y comprobar la coherencia global del sistema. Los resultados confirman que las operaciones implementadas en VHDL reproducen fielmente el comportamiento esperado de los algoritmos teóricos descritos en el capítulo de diseño. La ejecución secuencial de los módulos *Shuffling*, *Scrambling*, *Deshuffling* y *Descrambling* demuestra una alta estabilidad y sincronización entre

las distintas etapas, garantizando la integridad de los datos a lo largo de todo el proceso.

6.4.1. Análisis de comportamiento de los módulos

Los resultados de simulación muestran que el módulo *Shuffling* realiza correctamente la permutación pseudoaleatoria de los elementos de la matriz, cumpliendo las condiciones de dispersión y no repetición. El módulo *Scrambling* introduce la no linealidad esperada mediante las operaciones XOR, sumas módulo 255 y sustituciones *S-box*, logrando una transformación con alta variabilidad entre las matrices de entrada y salida. Por su parte, los módulos inversos *Deshuffling* y *Descrambling* restituyen exactamente los datos originales, lo que confirma la correcta implementación de la lógica reversible definida en el diseño.

6.4.2. Evaluación de la reversibilidad y robustez

Las pruebas de reversibilidad realizadas entre las parejas de módulos complementarios (*Shuffling–Deshuffling* y *Scrambling–Descrambling*) evidencian una correspondencia perfecta entre las matrices de entrada y salida. Las simulaciones muestran coincidencia total a nivel de byte, lo que garantiza la ausencia de pérdidas de información y confirma el carácter determinista del sistema bajo una misma configuración de parámetros. Además, el empleo de un mapa caótico como generador pseudoaleatorio introduce un grado de sensibilidad controlada: pequeñas variaciones en la semilla o en el parámetro r producen secuencias completamente diferentes, reforzando la robustez frente a intentos de predicción o correlación.

6.4.3. Ocupación de recursos lógicos

El análisis de la ocupación de recursos en la FPGA constituye un aspecto esencial para evaluar la viabilidad del diseño hardware en dispositivos con recursos limitados. En la Tabla 33 se resume la utilización de LUTs obtenida para los distintos módulos implementados: *scrambling*, *descrambling*, *shuffling* y *deshuffling*.

Módulo	Función	Ocupación (LUTs)
Scrambling	Desordenamiento pseudoaleatorio	175 007
Descrambling	Reversión del proceso de *scrambling*	5 230
Shuffling	Permutación espacial de datos	689 137
Deshuffling	Reversión del proceso de *shuffling*	518 170

Tabla 33: Ocupación de LUTs por módulo implementado

Los resultados muestran una clara asimetría entre los procesos directos e inversos. El módulo de *scrambling* presenta una utilización de **175 007 LUTs**, mientras que su correspondiente inverso, *descrambling*, requiere únicamente **5 230 LUTs**, debido a que este último se limita a revertir el proceso mediante la misma secuencia pseudoaleatoria, sin necesidad de generar nuevos patrones de mezcla.

Por otra parte, el bloque de *deshuffling* —encargado de invertir la permutación espacial aplicada en el proceso de *shuffling*— alcanza una ocupación de **518 170 LUTs**. Considerando la simetría funcional existente entre ambos módulos y la diferencia observada entre el par *scrambling/descrambling*, puede estimarse que el bloque *shuffling* requiere aproximadamente **687 947 LUTs**.

Este valor confirma que el proceso directo de desordenamiento (*shuffling*) es el más costoso en términos de recursos lógicos, ya que integra tanto la generación de las matrices de permutación como la gestión de las coordenadas pseudoaleatorias asociadas a cada posición de la matriz de datos. Por el contrario, las operaciones inversas (*descrambling* y *deshuffling*) aprovechan los mismos patrones ya generados, simplificando su estructura lógica y reduciendo drásticamente la ocupación.

En conjunto, la implementación completa del sistema de ofuscación (*shuffling* + *scrambling*) supone una demanda elevada de LUTs, que debe tenerse en cuenta en entornos de cómputo paralelo con capacidad restringida. No obstante, la alta ocupación se justifica por la complejidad asociada a la generación caótica y a la reversibilidad garantizada del proceso, características fundamentales para mantener la integridad e irreversibilidad de la transformación.

6.4.4. Limitaciones y posibles mejoras

Entre las limitaciones detectadas, se encuentra la dependencia de los parámetros del mapa caótico y el tamaño fijo de la matriz, que podrían restringir la escalabilidad del diseño. Como líneas de mejora se plantea la incorporación de controladores que permitan modificar dinámicamente los parámetros de `semilla` y r durante la ejecución, así como la paralelización parcial de ciertas operaciones para reducir la latencia global. Asimismo, sería posible integrar módulos adicionales de generación de entropía o aleatoriedad física para fortalecer el comportamiento pseudoaleatorio del sistema.

En conjunto, los resultados obtenidos confirman que el sistema propuesto cumple con los objetivos de diseño planteados: los módulos son funcionales, reversibles y eficientes, y el flujo de datos entre ellos se mantiene estable y determinista. El

sistema completo demuestra un equilibrio adecuado entre complejidad, rendimiento y fiabilidad, validando su aplicabilidad en entornos digitales donde se requiera una manipulación segura y reversible de la información.

Capítulo 7

Conclusiones y Recomendaciones

7.1. Conclusiones

El desarrollo de este trabajo ha permitido diseñar, implementar y validar un sistema digital de ofuscación y restauración de datos completamente funcional, basado en módulos VHDL. El sistema propuesto —compuesto por los bloques *Shuffling*, *Scrambling*, *Deshuffling* y *Descrambling*— ha demostrado ser capaz de transformar y recuperar información de forma reversible, manteniendo la integridad de los datos y la coherencia temporal en todo el flujo de procesamiento.

Desde el punto de vista funcional, las simulaciones realizadas en **Xilinx ISE Design Suite 14.7** han confirmado que cada módulo cumple de manera precisa con las operaciones descritas en la fase de diseño. El *Shuffling* genera permutaciones pseudoaleatorias estables, el *Scrambling* introduce no linealidad y dispersión mediante operaciones XOR y sustituciones *S-box*, mientras que sus módulos inversos recuperan con exactitud la matriz original. Las pruebas de reversibilidad evidencian una correspondencia perfecta entre las matrices de entrada y salida, validando la consistencia matemática del sistema y la correcta implementación de los algoritmos.

En el análisis de propagación entre módulos, se verificó que las señales de control (`done`, `ready`, `scrambled_ready`) presentan un comportamiento estable y sincronizado, garantizando una comunicación segura y libre de conflictos. El flujo completo de datos, desde la ofuscación hasta la desofuscación, se ejecuta sin pérdidas ni alteraciones, confirmando que la arquitectura modular favorece la reutilización y la escalabilidad del sistema.

En términos de rendimiento, el sistema muestra una **latencia reducida** y un **uso moderado de recursos lógicos**, lo que lo hace viable para su implementación en dispositivos FPGA de gama media o en plataformas embebidas con recursos limitados. La estructura en máquinas de estados finitas simplifica la síntesis, reduce

el consumo y permite alcanzar frecuencias operativas elevadas sin comprometer la estabilidad del diseño.

En conjunto, los resultados experimentales demuestran que el sistema cumple los objetivos propuestos:

- Implementar una arquitectura modular en VHDL capaz de realizar procesos de ofuscación y restauración de datos totalmente reversibles.
- Garantizar la integridad de la información y la sincronización entre módulos en todo el flujo de procesamiento.
- Obtener un diseño eficiente en tiempo y recursos, adaptable a distintas configuraciones y tamaños de matriz.

Finalmente, el trabajo desarrollado constituye una base sólida para la construcción de sistemas digitales orientados a la seguridad de la información, demostrando que la combinación de **mapas caóticos, operaciones no lineales y estructuras reversibles** puede integrarse con éxito en entornos hardware.

7.2. Recomendaciones

Durante el desarrollo del proyecto se han identificado varias líneas de mejora y posibles ampliaciones que podrían fortalecer y extender el trabajo realizado:

1. **Parametrización dinámica:** incorporar mecanismos que permitan modificar en tiempo de ejecución los valores de la semilla y del parámetro r del mapa caótico, aumentando la variabilidad y la resistencia a ataques predictivos.
2. **Optimización del rendimiento:** explorar la paralelización parcial de ciertas operaciones (por ejemplo, el cálculo de sumas módulo 255 o las sustituciones *S-box*) para reducir aún más la latencia total del sistema.
3. **Ampliación de tamaño matricial:** adaptar los módulos a matrices de dimensiones variables mediante el uso de `generics` y tipos parametrizables, garantizando flexibilidad y escalabilidad sin modificar la lógica interna.
4. **Implementación física:** sintetizar y probar el diseño en una FPGA real (por ejemplo, Xilinx Spartan-6 o Artix-7) para obtener métricas experimentales de consumo, frecuencia máxima y temperatura, complementando los resultados de simulación.

5. **Integración en sistemas criptográficos:** combinar los módulos desarrollados con bloques de cifrado simétrico o generadores de claves pseudoaleatorias, ampliando el alcance del sistema hacia aplicaciones de protección de datos e IoT.

En resumen, el trabajo desarrollado ha demostrado la viabilidad y eficiencia de una arquitectura modular de ofuscación reversible en VHDL. Las futuras líneas de investigación y optimización permitirán ampliar sus capacidades y consolidar su aplicación en entornos de seguridad hardware, comunicaciones embebidas y procesamiento digital de información sensible.

Bibliografía

[1] IoT Analytics. Number of connected iot devices (2024 update). https://iot-analytics.com/wp-content/uploads/2024/09/INSIGHTS-RELEASE-Number-of-connected-IoT-devices-vf.pdf, 2024. Accessed: October 6, 2025.

[2] Francisco Alcaraz-Velasco, José M. Palomares, and Joaquín Olivares. Lightweight security system based on data shuffling. *Computer Networks*, 199:108470, 2021. doi: 10.1016/j.comnet.2021.108470.

[3] Francisco Alcaraz-Velasco, José M. Palomares, and Joaquín Olivares. Análisis del desordenamiento aleatorio de bloques de mensajes como medida de integridad y seguridad de bajo coste. In *Actas de la Conferencia Ibérica de Sistemas y Tecnologías de Información (CISTI)*, pages 1–8. IEEE, 2021.

[4] Benjamin Buhrow, Patrick Riemer, et al. Block cipher speed and energy efficiency records on the msp430: System design trade-offs for 16-bit embedded applications. In *3rd International Conference on Cryptology and Information Security in Latin America*, pages 104–123, 2015. doi: 10.1007/978-3-319-16295-9_6.

[5] Jongdeog Lee, Katia Kapitanova, and Sang H. Son. The price of security in wireless sensor networks. *Computer Networks*, 54(17):2967–2978, 2010. doi: 10.1016/j.comnet.2010.05.011.

[6] Francisco Alcaraz, Jose Manuel Palomares, and Joaquin Olivares. Shuffling and scrambling mechanisms for data integrity in constrained iot systems. *Universidad de Córdoba*, 2022. Technical Report.