ALBA – Mediciones

# Docker

## Introducción a Docker

Docker es una plataforma de software que facilita la creación, el despliegue y la gestión de aplicaciones en contenedores. Los contenedores son entornos ligeros que incluyen todo lo necesario para ejecutar una aplicación de forma aislada del sistema operativo host.

## Conceptos Básicos

* **Imagen** (Image): Plantilla inmutable que contiene el sistema operativo y el software necesario para ejecutar una aplicación.
* **Contenedor** (Container): Instancia ejecutable de una imagen, aislada del sistema operativo host.
* **Dockerfile**: Archivo con instrucciones para construir una imagen de Docker.
* **Registro** (Registry): Repositorio para almacenar y distribuir imágenes de Docker.
* **Docker** Compose: Herramienta para definir y ejecutar aplicaciones Docker multi-contenedor usando un archivo YAML.
* **Orquestación**: Coordinación de múltiples contenedores en un entorno de producción usando herramientas como Docker Swarm o Kubernetes.

## Comandos Básicos

* **docker run**: Ejecuta un contenedor basado en una imagen específica.
  + docker run [opciones] imagen [comando] [argumentos]
* **docker ps**: Lista los contenedores en ejecución.
  + docker ps
* **docker build**: Crea una imagen a partir de un Dockerfile.
  + docker build -t nombre\_imagen .
    - -t etiqueta la imagen con un nombre específico.
* **docker images**: Muestra todas las imágenes locales.
  + docker images
* **docker pull**: Descarga una imagen desde un registro.
  + docker pull nombre\_imagen
* **docker push**: Sube una imagen a un registro.
  + docker push nombre\_imagen
* **docker stop**: Detiene un contenedor en ejecución.
  + docker stop id\_contenedor
* **docker rm**: Elimina un contenedor.
  + docker rm id\_contenedor
* **docker rmi**: Elimina una imagen.
  + docker rmi nombre\_imagen

## Permisos en Ubuntu

Para permitir que el usuario actual utilice Docker sin necesidad de sudo, se debe agregar el usuario al grupo docker:

sudo usermod -aG docker $USER

Después de ejecutar este comando, es necesario cerrar la sesión y volver a iniciarla para que los cambios tengan efecto.

## Creación de un Dockerfile Básico

Para ejecutar una aplicación Java, podemos crear un archivo llamado Dockerfile con el siguiente contenido:

# Usa la imagen base de OpenJDK 20

FROM openjdk:20

# Copia todo el contenido al directorio de trabajo dentro del contenedor

COPY . /usr/app

# Establece el directorio de trabajo en /usr/app

WORKDIR /usr/app

Este Dockerfile especifica que queremos usar la imagen base de OpenJDK 20, copiar el contenido actual al contenedor y establecer el directorio de trabajo en /usr/app.

## Construcción de la Imagen y el Contenedor

* **Construcción de la imagen**:
  + docker build -t [image\_name] [app\_path]
    - -t [image\_name] etiqueta la imagen con el nombre especificado.
    - [app\_path] es la ruta al directorio que contiene el Dockerfile.
* **Listar imágenes disponibles**:
  + docker image ls
* **Crear y ejecutar un contenedor**:
  + docker run -it {--rm} --name [container\_name] [image\_name] {command\_line}
    - -it permite la interacción con la terminal del contenedor.
    - --rm elimina el contenedor después de que se detenga.
    - --name [container\_name] asigna un nombre al contenedor.
    - [command\_line] especifica el comando que se ejecutará dentro del contenedor.
* **Listar contenedores**:
  + docker container ls -al
* **Eliminar contenedores**:
  + docker rm [container\_id]
* **Eliminar imágenes**:
  + docker rmi [image\_id]

## Ejecutar una Aplicación Java con Docker

Para ejecutar una aplicación Java empaquetada en un archivo .jar:

docker run -it --name [container\_name] [image\_name] java -jar [jar\_file\_name] [input\_file] [output\_file] [algoritmo]

* java -jar [jar\_file\_name] ejecuta el archivo JAR especificado.
* [input\_file] y [output\_file] son los archivos de entrada y salida, respectivamente.
* [algoritmo] puede ser un argumento adicional pasado a la aplicación.

## Docker Compose

Docker Compose permite definir y gestionar aplicaciones multi-contenedor. La configuración se define en un archivo docker-compose.yml.

Ejemplo de archivo docker-compose.yml:

version: '3'

services:

web:

image: nginx

ports:

- "80:80"

db:

image: mysql

environment:

MYSQL\_ROOT\_PASSWORD: example

* **Levantar los servicios**:
  + docker-compose up
* **Detener y eliminar los servicios**:
  + docker-compose down

## Redes y Volúmenes en Docker

* **Redes**: Conectar contenedores entre sí para permitir la comunicación.
  + docker network create mi\_red
  + docker run -d --name contenedor1 --network mi\_red nginx
  + docker run -d --name contenedor2 --network mi\_red alpine sleep 1000
* **Volúmenes**: Persistir datos fuera del ciclo de vida de los contenedores.
  + docker volume create mi\_volumen
  + docker run -d --name mi\_contenedor -v mi\_volumen:/data nginx

## Orquestación con Docker Swarm y Kubernetes

* **Docker Swarm**: Herramienta de orquestación integrada en Docker.
  + docker swarm init
  + docker service create --name mi\_servicio -p 80:80 nginx
* **Kubernetes**: Plataforma de orquestación de contenedores más avanzada y ampliamente utilizada.

## Buenas Prácticas con Docker

* Mantén las imágenes ligeras y optimizadas.
* Usa Docker Compose para aplicaciones complejas.
* Versiona y documenta tu Dockerfile.
* Usa variables de entorno para configuraciones sensibles.
* Mantén los contenedores y volúmenes limpios y organizados.
* Automatiza la construcción y despliegue de contenedores.

# Algoritmos Implementados

## AES (Advanced Encryption Standard)

### Origen y Propósito

AES, o Estándar de Encriptación Avanzada, fue adoptado por el Instituto Nacional de Estándares y Tecnología (NIST) de los Estados Unidos en 2001 como un estándar de encriptación para reemplazar el DES (Data Encryption Standard). Fue diseñado para ser seguro y eficiente en una amplia gama de dispositivos y aplicaciones.

### Estructura y Funcionamiento

AES opera mediante un cifrado simétrico, lo que significa que la misma clave se utiliza tanto para la encriptación como para la desencriptación de los datos.

Funciona sobre bloques de datos de 128 bits, utilizando claves de longitud variable (128, 192 o 256 bits).

El algoritmo opera en rondas, donde cada ronda aplica una serie de transformaciones sobre el bloque de datos utilizando la clave.

### Proceso de Encriptación

**Inicialización**: Se preparan los datos y se expande la clave para generar subclaves para cada ronda.

**Rondas Principales**: Se aplican varias rondas (10, 12 o 14, dependiendo de la longitud de la clave) de transformaciones, que incluyen sustituciones no lineales, permutaciones y mezclas de columnas.

**Finalización**: Después de las rondas principales, se aplica una ronda final que omite la mezcla de columnas.

### Seguridad y Aplicaciones

AES se considera altamente seguro y resistente a los ataques criptográficos conocidos cuando se utiliza correctamente con claves suficientemente largas.

Es ampliamente utilizado en aplicaciones de seguridad informática, incluyendo la protección de datos sensibles, como la información personal y financiera en aplicaciones y servicios en línea.

### Implementaciones y Variaciones

Existen diversas implementaciones optimizadas de AES para diferentes plataformas, incluyendo hardware dedicado que acelera el proceso de encriptación y desencriptación.

AES ha sido adoptado internacionalmente y es utilizado por gobiernos, empresas y particulares en todo el mundo debido a su robustez y eficiencia.

### Clase ‘Aes’ Implementada en el Proyecto

* **Atributos**:
  + private byte[] key;: Almacena la clave secreta utilizada para AES.
  + private String instanceString = "AES/";: String que especifica el modo y el padding para el cifrado AES.
  + private byte[] iv;: Vector de inicialización (IV), necesario para algunos modos de operación.
* **Constructores**:
  + public Aes(String mode, String padding, String keyPath): Constructor que inicializa un objeto AES con un modo específico, padding y clave cargada desde un archivo especificado por keyPath.
  + public Aes(int keySize): Constructor que genera una nueva clave AES de tamaño especificado (keySize) y la guarda en un archivo.
* **Métodos privados**:
  + generateKey(int keySize): Genera una clave AES de tamaño keySize (128, 192, o 256 bits).
  + generateIv(int size): Genera un IV aleatorio de tamaño size (12 bytes para GCM y 16 bytes para otros modos).
* **Métodos públicos**:
  + encryptFile(File inputFile, File outputFile): Encripta un archivo de entrada utilizando AES y guarda el resultado en un archivo de salida.
    - Lee la clave y el IV.
    - Inicializa el Cipher según el modo y padding especificados.
    - Lee los bytes del archivo de entrada, los encripta y escribe en el archivo de salida.
  + decryptFile(File inputFile, File outputFile): Desencripta un archivo de entrada cifrado con AES y guarda el resultado en un archivo de salida.
    - Lee la clave y el IV (si no es ECB).
    - Inicializa el Cipher para desencriptar según el modo y padding.
    - Lee los bytes del archivo de entrada, los desencripta y escribe en el archivo de salida.

### Modos y Paddings

El código soporta varios modos de cifrado (ECB, CBC, CFB, OFB, CTR, GCM) y opciones de padding (NoPadding, PKCS5Padding, PKCS7Padding, ISO10126Padding).

Se utilizan diferentes clases de Java (Cipher, SecretKeySpec, IvParameterSpec, GCMParameterSpec) para configurar y operar con AES en diferentes modos y configuraciones de seguridad.

|  |  |  |
| --- | --- | --- |
| **Modo de operación** | **Paddings Disponibles** | **Descripción** |
| **ECB** (Electronic Codebook) | NoPadding, PKCS5Padding, PKCS7Padding, ISO10126Padding | Cada bloque de texto plano se cifra de forma independiente. Es el modo más sencillo, pero menos seguro debido a que bloques idénticos de texto plano generan bloques cifrados idénticos. No se recomienda para datos sensibles. |
| **CBC** (Cipher Block Chaining) | NoPadding, PKCS5Padding, PKCS7Padding, ISO10126Padding | Cada bloque de texto plano se combina (XOR) con el bloque cifrado anterior antes de ser cifrado. Necesita un IV (Vector de Inicialización) aleatorio para cada cifrado. Mejora la seguridad en comparación con ECB ya que no se generan bloques cifrados idénticos para el mismo texto plano. |
| **CFB** (Cipher Feedback) | NoPadding | Cifra bloques de texto plano en lugar de bloques completos. Permite trabajar con tamaños de bloques menores que los de cifrado. |
| **OFB** (Output Feedback) | NoPadding | Similar a CFB, pero opera en modo stream, cifrando un stream de bytes. No requiere padding. |
| **CTR** (Counter) | NoPadding | Convierte un bloque cifrado en un generador de secuencias de caracteres pseudoaleatorias. Permite cifrar datos de manera paralela y es útil para transmisión de datos en tiempo real. |
| **GCM** (Galois/Counter Mode) | NoPadding | Proporciona cifrado autenticado, donde además de cifrar los datos, calcula un código de autenticación para garantizar la integridad y autenticidad de los datos. Es eficiente y ampliamente utilizado en comunicaciones seguras. |

**Modos de Operación**:

* **ECB**: Cada bloque de texto plano se cifra de manera independiente. Es simple pero menos seguro debido a la repetición de bloques cifrados.
* **CBC**: Combina cada bloque de texto plano con el bloque cifrado anterior antes de cifrarlo, utilizando un IV. Mejora la seguridad respecto a ECB al eliminar patrones repetitivos.
* **CFB y OFB**: Operan en modo stream, permitiendo trabajar con flujos de datos más pequeños que los bloques de cifrado estándar. No requieren padding.
* **CTR**: Convierte el cifrado en un generador de stream de datos, lo que permite cifrar datos en tiempo real y de manera paralela.

**Paddings**:

* **NoPadding**: No se añade ningún byte adicional al texto plano antes de cifrarlo. El texto plano debe ser múltiplo del tamaño del bloque de cifrado.
* **PKCS5Padding y PKCS7Padding**: Añaden bytes adicionales para que el texto plano sea múltiplo del tamaño del bloque. PKCS7Padding es una extensión de PKCS5Padding.
* **ISO10126Padding**: Similar a PKCS7Padding pero utiliza bytes aleatorios en el último bloque para el padding, lo que dificulta la deducción del padding original.

### Consideraciones

* **Seguridad**: ECB es el menos seguro debido a la repetición de patrones cifrados. CBC, CFB, OFB y CTR ofrecen mejor seguridad al evitar esta repetición.
* **Integridad**: GCM proporciona cifrado autenticado, lo que asegura tanto la confidencialidad como la integridad de los datos.
* **Compatibilidad**: La elección del modo y padding depende de los requisitos específicos de seguridad y las restricciones de la aplicación. Es importante seleccionar el más adecuado según el contexto de uso.

### Manejo de Excepciones

El código maneja varios tipos de excepciones (IOException, InvalidAlgorithmParameterException, InvalidKeyException, NoSuchAlgorithmException, BadPaddingException, IllegalBlockSizeException, NoSuchPaddingException) que pueden ocurrir durante las operaciones de encriptación y desencriptación.

## Camellia

### Origen y Propósito

Camellia fue desarrollado como una alternativa segura y eficiente al Estándar de Encriptación Avanzada (AES). Fue publicado en 2000 y adoptado como un estándar internacional por la ISO/IEC y la IETF (Internet Engineering Task Force).

Su diseño se basa en principios criptográficos sólidos, incluyendo sustituciones no lineales y permutaciones, para asegurar la confidencialidad de los datos.

### Estructura y Funcionamiento

* Cifrado Simétrico:
  + Camellia opera como un cifrado simétrico, utilizando la misma clave para encriptar y desencriptar los datos.
* Tamaño de Bloque y Longitud de Clave:
  + Funciona con bloques de datos de 128 bits, al igual que AES.
  + Soporta claves de longitud variable: 128, 192 y 256 bits, proporcionando flexibilidad en la seguridad según los requisitos del sistema.
* Operación por Rondas:
  + Al igual que AES, Camellia opera en rondas de transformación.
  + El número de rondas varía dependiendo de la longitud de la clave: 18 rondas para claves de 128 y 192 bits, y 24 rondas para claves de 256 bits.

### Proceso de Encriptación

**Inicialización**: Los datos se preparan y la clave se expande para generar subclaves utilizadas en cada ronda.

**Rondas Principales**: Se aplican varias rondas de transformación que incluyen sustituciones no lineales, permutaciones y mezclas de columnas para asegurar la confidencialidad de los datos.

**Finalización**: Después de las rondas principales, se realiza una fase final que asegura la integridad de los datos cifrados.

### Implementación y Uso

Es utilizado en una amplia gama de aplicaciones y sistemas que requieren protección de datos sensibles, incluyendo redes seguras, comunicaciones en línea, almacenamiento de datos y más.

La eficiencia y la seguridad de Camellia lo hacen adecuado para aplicaciones donde se requiere un alto nivel de protección criptográfica con un rendimiento óptimo.

### Comparación con AES

Aunque AES es más ampliamente conocido y utilizado, Camellia ofrece una alternativa sólida y compatible en términos de seguridad y rendimiento.

Ambos algoritmos están diseñados para ser seguros contra una amplia gama de ataques criptográficos y se utilizan ampliamente en la industria y en aplicaciones gubernamentales.

### Clase ‘Camellia’ Implementada en el Proyecto

* **Atributos**
  + private byte[] key: Almacena la clave utilizada para el cifrado y descifrado.
  + private String instanceString: Cadena que especifica el modo y padding para la instancia de cifrado.
  + private byte[] iv: Vector de inicialización utilizado en algunos modos de operación para inicializar el cifrado de manera segura.
* **Constructor** Camellia(String mode, String padding, String keyPath):
  + Recibe parámetros que especifican el modo de cifrado, padding y la ruta del archivo que contiene la clave.
  + Lee la clave desde el archivo especificado utilizando FileHandler.readKeyFromFile(keyPath).
  + Construye la cadena instanceString concatenando el modo y padding especificados.
  + Añade el proveedor de seguridad BouncyCastleProvider al entorno de seguridad usando Security.addProvider(new BouncyCastleProvider()).
  + Genera un vector de inicialización (iv) dependiendo del modo seleccionado.
* **Constructor** Camellia(int keySize):
  + Recibe el tamaño de la clave como parámetro.
  + Genera una nueva clave usando generateKey(keySize).
  + Guarda la clave generada en un archivo usando FileHandler.saveKeyToFile().
* **Método** generateKey(int keySize):
  + Genera una clave de tamaño específico (keySize) usando el generador de claves KeyGenerator de Camellia.
  + Asegura que el tamaño de la clave sea válido (128, 192 o 256 bits).
  + Utiliza el proveedor de seguridad Bouncy Castle ("BC") para obtener la instancia del generador de claves.
* **Método** generateIv(int size):
  + Genera un vector de inicialización (iv) de tamaño especificado (size).
  + Utiliza un generador de números aleatorios seguro (SecureRandom) para generar los bytes del vector de inicialización.
* **Método** encryptFile(File inputFile, File outputFile):
  + Encripta un archivo de entrada (inputFile) y guarda el resultado en un archivo de salida (outputFile).
  + Lee todos los bytes del archivo de entrada.
  + Inicializa el cifrado usando la clave y el modo especificados en instanceString.
  + Escribe el vector de inicialización (iv) antes de los datos cifrados en el archivo de salida si no se está utilizando el modo ECB.
  + Captura y maneja las excepciones relacionadas con la encriptación.
* **Método** decryptFile(File inputFile, File outputFile):
  + Descifra un archivo de entrada cifrado (inputFile) y guarda el resultado en un archivo de salida (outputFile).
  + Lee todos los bytes del archivo de entrada.
  + Extrae el vector de inicialización (iv) del archivo cifrado si no se está utilizando el modo ECB.
  + Inicializa el descifrado usando la clave y el modo especificados en instanceString.
  + Escribe los bytes descifrados en el archivo de salida.
  + Captura y maneja las excepciones relacionadas con el descifrado.

### Modos y Paddings

|  |  |  |
| --- | --- | --- |
| **Modo** | **Paddings** | **Descripción** |
| ECB | NoPadding,  PKCS5Padding,  ISO10126Padding | **Modo de Bloques Electrónico (ECB)**: Cifra cada bloque de datos de forma independiente. Es el modo más simple y menos seguro debido a la repetición de cifrados idénticos para bloques de texto plano iguales. |
| CBC | NoPadding,  PKCS5Padding,  ISO10126Padding | **Cifrado en Cadena (CBC)**: Cada bloque de texto plano se combina con el bloque previamente cifrado antes de ser cifrado. Requiere un vector de inicialización (IV) único y aleatorio para cada mensaje. Proporciona seguridad mejorada respecto a ECB. |
| CFB | NoPadding | **Cifrado de Retroalimentación por Cifrado (CFB)**: Convierte un cifrado de bloque en un cifrado de flujo, donde cada bit o byte del texto plano se cifra separadamente antes de ser devuelto a la operación de cifrado. No requiere padding. |
| OFB | NoPadding | **Cifrado de Retroalimentación por Salida (OFB)**: Similar a CFB, convierte un cifrado de bloque en un cifrado de flujo, pero utiliza una función de cifrado para generar un flujo de bits que se combina con el texto plano. No requiere padding. |
| CTR | NoPadding | **Cifrado de Contador (CTR)**: Convierte un cifrado de bloque en un cifrado de flujo mediante la generación de un flujo de bits que se combina con el texto plano mediante una operación XOR. No requiere padding. |
| GCM | NoPadding | **Modo Galois/Counter (GCM)**: Proporciona autenticación de mensajes junto con cifrado, utilizando un modo de operación de contadores para cifrar los datos y un polinomio de autenticación de Galois para autenticar los datos cifrados. |

### Manejo de Excepciones

La clase Camellia implementa manejo de excepciones para asegurar operaciones seguras y controladas durante el cifrado y descifrado de archivos. Los tipos de excepciones manejadas incluyen:

* **IOException**: Capturada durante operaciones de lectura y escritura de archivos.
* **InvalidAlgorithmParameterException, InvalidKeyException, NoSuchAlgorithmException, BadPaddingException, IllegalBlockSizeException, NoSuchPaddingException**: Excepciones de criptografía que pueden surgir por problemas con parámetros, claves, algoritmos o relleno incorrecto.

Cada excepción se maneja capturándola en el contexto adecuado y lanzando una excepción general (Exception) encapsulando la original para manejo consistente y notificación clara de errores.

## ChaCha20

### Origen y Propósito

ChaCha20 es un algoritmo de cifrado desarrollado por Daniel J. Bernstein como una variante mejorada del cifrado Salsa20. ChaCha20 fue diseñado para ser más seguro y eficiente que sus predecesores. Es ampliamente utilizado por su rendimiento y su capacidad para ofrecer cifrado seguro con una implementación sencilla y robusta. ChaCha20 es parte de la familia de cifrados de flujo y es particularmente efectivo en entornos con limitaciones de recursos, como dispositivos móviles.

### Estructura y Funcionamiento

ChaCha20 es un cifrado de flujo basado en un generador de números pseudoaleatorios que opera en una secuencia de 512 bits organizada en una matriz de 4x4. La estructura de ChaCha20 incluye:

* **Clave**: Utiliza una clave simétrica de 256 bits.
* **Nonce (Number only used once)**: Es un valor único por mensaje, que evita la repetición de secuencias de cifrado.
* **Contador**: Inicialmente se puede establecer a 0, y se incrementa por cada bloque cifrado.
* **Bloques de 64 bytes**: El cifrado procesa los datos en bloques de 64 bytes, generando un flujo de claves a partir del cual se obtiene el texto cifrado mediante una operación XOR.

El algoritmo realiza 20 rondas de operaciones sobre la matriz, incluyendo sumas, rotaciones y operaciones XOR, lo que asegura la dispersión y mezcla de los datos de entrada.

### Proceso de Encriptación

1. **Inicialización**:
   1. Configura la matriz de 4x4 con la clave, el nonce y un valor constante.
2. **Generación de Clave de Flujo**:
   1. Procesa la matriz a través de 20 rondas de operaciones para producir el flujo de claves de 64 bytes.
3. **Cifrado de Datos**:
   1. Cada bloque de datos se cifra mediante una operación XOR con el flujo de claves generado.
4. **Incremento del Contador**:
   1. Después de procesar cada bloque de 64 bytes, el contador se incrementa para la siguiente operación.

### Seguridad y Aplicaciones

ChaCha20 se considera altamente seguro cuando se utiliza correctamente, gracias a su resistencia comprobada contra ataques criptográficos conocidos. Es resistente a ataques de canal lateral y es eficiente en hardware y software, lo que lo hace adecuado para una amplia gama de aplicaciones, desde cifrado de datos en sistemas embebidos hasta comunicaciones seguras en la web, como en TLS (Transport Layer Security).

### Implementación y Uso

ChaCha20 se implementa fácilmente en muchas plataformas y lenguajes de programación debido a su diseño simple y eficiente. En Java, su implementación puede beneficiarse de bibliotecas de criptografía como BouncyCastle.

### Clase ‘ChaCha20’ Implementada en el Proyecto

La clase ChaCha20 en tu proyecto implementa el cifrado y descifrado utilizando ChaCha20 con soporte para varios modos:

* **Campos y Variables**:
  + key: Almacena la clave simétrica.
  + nonce: Almacena el nonce utilizado para la operación.
  + instanceString: Define la configuración del algoritmo (modo).
* **Constructores**:
  + ChaCha20(String mode, String keyPath): Lee la clave desde un archivo y configura el modo y el nonce basado en la selección del usuario.
  + ChaCha20(): Genera una nueva clave de 256 bits y la guarda en un archivo.
* **Métodos Principales**:
  + encryptFile(File inputFile, File outputFile): Cifra el contenido de un archivo y guarda el resultado.
  + decryptFile(File inputFile, File outputFile): Descifra el contenido de un archivo cifrado previamente.
* **Manejo de Modos**:
  + setupMode(String mode): Configura el algoritmo para usar ChaCha20 en diferentes modos (ChaCha20, XChaCha20, ChaCha20-Poly1305, XChaCha20-Poly1305).

### Variaciones y Modos

ChaCha20 tiene varias variaciones y modos que ofrecen diferentes características de seguridad y usabilidad:

* **ChaCha20**: La versión básica que usa un nonce de 12 bytes y un contador de 32 bits.
* **XChaCha20**: Una extensión de ChaCha20 que usa un nonce de 24 bytes, permitiendo más combinaciones únicas.
* **ChaCha20-Poly1305**: Combina ChaCha20 con el autenticador Poly1305 para proporcionar autenticación de mensajes junto con el cifrado.
* **XChaCha20-Poly1305**: Combina XChaCha20 con Poly1305, ofreciendo mayor seguridad en el uso de nonces extendidos.

### Manejo de Excepciones

La clase ChaCha20 implementa un manejo robusto de excepciones para capturar y manejar errores comunes que pueden ocurrir durante las operaciones de cifrado y descifrado:

* **IOException**: Captura errores en la lectura y escritura de archivos.
* **InvalidAlgorithmParameterException**: Maneja errores en los parámetros del algoritmo.
* **InvalidKeyException**: Maneja errores relacionados con claves inválidas.
* **NoSuchAlgorithmException**: Captura errores cuando el algoritmo ChaCha20 no está disponible.
* **BadPaddingException**: Maneja errores de relleno de datos.
* **IllegalBlockSizeException**: Maneja errores cuando los bloques de datos no tienen el tamaño esperado.
* **NoSuchPaddingException**: Captura errores relacionados con configuraciones de padding no disponibles.

Este manejo de excepciones asegura que la clase ChaCha20 puede manejar de manera adecuada y segura las diversas condiciones de error que pueden ocurrir durante las operaciones de cifrado y descifrado.