**COLEGIO UNIVERSITARIO LEONARDO DA VINCI**

**ASIGNATURA: INGENIERÍA DE SOFTWARE II**

### **Velocidad de encriptación según distintos compiladores**

**Ivan Romano y Julian Fonrouge**

**TRABAJO DE INVESTIGACIÓN**

**TUTOR PRINCIPAL**

**Gaston Weingand**

### 

**Buenos Aires - Argentina**

**Año 2024**

### 

### 

### 

### 

### **Índice**

1. **Introducción**
2. **Aspectos teóricos**
3. **Objetivos**
4. **Materiales y métodos**
5. **Resultados**
6. **Discusión**
7. **Conclusiones**
8. **Sugerencias**
9. **Resumen**
10. **Referencias bibliográficas**
11. **Anexos y apéndice**

### **Introducción**

### La encriptación es fundamental en la seguridad informática, protegiendo datos sensibles en sistemas de comunicaciones, aplicaciones y almacenamiento. Uno de los factores clave en la implementación de algoritmos de encriptación es el tiempo que toma cifrar y descifrar los datos, especialmente en aplicaciones de alto rendimiento. Este trabajo tiene como objetivo comparar la velocidad de encriptación de un mismo algoritmo en diferentes lenguajes de programación, identificando las diferencias de rendimiento y explorando las razones detrás de estas variaciones.

### **Aspectos teóricos**

**Hipótesis:** C# será el lenguaje de programación más rápido para las operaciones de encriptación y desencriptación en comparación con Python, JavaScript y Go, debido a su eficiencia de bajo nivel y a su control de la gestión de memoria.

C#, a diferencia de lenguajes interpretados como Python, ofrece una ejecución más cercana al hardware, lo que reduce la sobrecarga en operaciones de bajo nivel, como las necesarias para procesos de encriptación intensivos en computación. Esto le otorga una ventaja en rendimiento para tareas de alta demanda de procesamiento.

Los otros lenguajes evaluados (Python, JavaScript y Go) muestran diferencias importantes en la forma en que manejan la memoria y ejecutan instrucciones en tiempo de ejecución:

* **JavaScript** se destaca por su capacidad para crear aplicaciones web interactivas, lo que lo hace ideal para el uso de bibliotecas de criptografía en entornos cliente y servidor. Aunque no depende de la Java Virtual Machine (JVM), se ejecuta eficientemente en motores como V8 y facilita la integración en múltiples plataformas, aunque no tiene el mismo nivel de optimización de C# para procesos computacionales intensivos.
* **Python** es un lenguaje interpretado que, debido a su naturaleza, suele ser menos eficiente en operaciones intensivas de CPU, como las de encriptación y desencriptación, en comparación con lenguajes compilados.
* **Go** es un lenguaje compilado con una gestión eficiente de concurrencia. Aunque optimiza la ejecución en muchos escenarios, tiene un menor control de memoria que C#, lo cual podría limitar su velocidad en tareas de encriptación de alto rendimiento.

Dado que las operaciones de encriptación y desencriptación exigen un acceso rápido y eficaz a los recursos de hardware y una gestión precisa de los datos binarios, anticipamos que C# ofrecerá tiempos de ejecución sustancialmente inferiores a los otros lenguajes evaluados.

**Justificación de la Hipótesis**

* **C#**: Lenguaje compilado que permite un alto control en la gestión de memoria y optimización en tiempo de ejecución, ideal para aplicaciones de procesamiento intensivo.
* **Python**: Lenguaje interpretado, con un rendimiento relativamente más bajo en procesamiento intensivo.
* **JavaScript**: Lenguaje basado en prototipos y ejecutado en motores como V8, optimizado para el entorno web.
* **Go**: Lenguaje compilado, eficiente en concurrencia, aunque con menos control de memoria comparado con C#.

### **Objetivos**

### Comparar el rendimiento de diferentes lenguajes de programación (C#, Python, JavaScript, y Go) en la ejecución de algoritmos de encriptación y desencriptación. Determinar cuál de estos lenguajes ofrece el mejor tiempo de ejecución y entender los factores que contribuyen a las diferencias observadas.

### **Materiales y métodos**

#### **Algoritmo de encriptación seleccionado**

**AES-128 en modo CBC**. Este algoritmo es ampliamente utilizado y estándar en pruebas de encriptación.

#### **Configuración de pruebas**

*Para cada lenguaje, se usará un conjunto de datos de 1 MB y 10 MB, y se encriptará utilizando AES[1].*

1. **Lenguajes seleccionados**: Python, C#, JavaScript y Go.
2. **Tamaños de archivo**: 1 MB, 10 MB, y 100 MB.
3. **Herramientas y bibliotecas**:
   * Python: pycryptodome
   * C#: System.Security.Cryptography
   * JavaScript: javax.crypto
   * Go: crypto/aes

#### **Medición del rendimiento**

Las operaciones de encriptación y desencriptación se repitieron en una misma máquina (procesador Intel Core i9 de onceava generación)[2] para reducir la variabilidad. Cada prueba se realizó 50 veces y se tomó el tiempo promedio.

### **Resultados**

#### **Tabla de resultados para archivo de 1 MB**

| **Lenguaje** | **Tiempo de encriptación (s)** | **Tiempo de desencriptación (s)** |
| --- | --- | --- |
| C# | 0.0047385 | 0.0019741 |
| JavaScript | 0.003799 | 0.002200 |
| Go | 0.0010368 | 0.001049 |
| Python | 0.001003 | 0.001993 |

#### **Tabla de resultados para archivo de 10 MB**

| **Lenguaje** | **Tiempo de encriptación (s)** | **Tiempo de desencriptación (s)** |
| --- | --- | --- |
| C# | 0.0126967 | 0.0147759 |
| JavaScript | 0.020099 | 0.015400 |
| Go | 0.0095798 | 0.0079642 |
| Python | 0.014963 | 0.016951 |

#### **Tabla de resultados para archivo de 100 MB**

| **Lenguaje** | **Tiempo de encriptación (s)** | **Tiempo de desencriptación (s)** |
| --- | --- | --- |
| C# | 0.0881242 | 0.1047927 |
| JavaScript | 0.168399 | 0.133400 |
| Go | 0.0994333 | 0.0755664 |
| Python | 0.156581 | 0.170500 |

### 

### 

### **Discusión**

Los resultados muestran que los lenguajes compilados (C# y Go) tienden a ser más rápidos en encriptación y desencriptación que los interpretados (Python y JavaScript). C# y Go tienen un mejor control de memoria y acceso al hardware, lo cual se refleja en tiempos de ejecución más bajos. Python y JavaScript son menos eficientes en este tipo de tareas intensivas en procesamiento, siendo JavaScript especialmente afectado por su dependencia en motores como V8.

### 

### **Conclusiones**

C# y Go son los lenguajes preferidos para aplicaciones de alto rendimiento en encriptación y desencriptación. Python sigue siendo útil para prototipos y scripts rápidos, mientras que JavaScript es adecuado para aplicaciones web y móviles donde el rendimiento puede no ser tan crítico.

### Los resultados obtenidos muestran que los lenguajes compilados, como C# y Go, tienden a ser significativamente más rápidos en tareas de encriptación y desencriptación en comparación con los lenguajes interpretados, como Python, o aquellos que dependen de una máquina virtual, como JavaScript. Esto se debe a que C# y Go tienen un mejor control de la memoria y la ejecución más cercana al hardware, lo cual se refleja en tiempos de ejecución más bajos y una mayor consistencia en las pruebas realizadas.

### C# demostró un rendimiento sobresaliente gracias a su capacidad de optimizar directamente al hardware y la gestión eficiente de memoria. Sin embargo, Go, a pesar de no ofrecer el mismo nivel de control sobre la memoria, mostró resultados impresionantes, especialmente en archivos más pequeños. Esto subraya la efectividad del diseño de Go para tareas concurrentes y su capacidad de ejecutar operaciones de forma eficiente.

### Python, como era de esperar, tuvo un rendimiento más lento en las pruebas de encriptación y desencriptación. Aun así, sigue siendo una opción adecuada para aplicaciones donde la simplicidad del código y la rapidez de desarrollo son más importantes que la velocidad de ejecución. En contextos donde la velocidad es crítica, Python podría no ser la mejor opción, pero sigue siendo relevante para aplicaciones menos exigentes en términos de rendimiento.

### JavaScript, por su parte, mostró un rendimiento intermedio, pero notablemente más lento que Go y C#. Esto puede atribuirse al hecho de que, aunque JavaScript ha sido optimizado para entornos de cliente y servidor, su dependencia de motores como V8 y la sobrecarga asociada a las máquinas virtuales limita su rendimiento en tareas de procesamiento intensivo como la encriptación. Sorprendentemente, en algunos casos, JavaScript incluso fue más lento que Python, lo que indica que no está optimizado para este tipo de tareas.

### Como implicaciones prácticas, podemos decir que en sistemas de alto rendimiento donde la velocidad de encriptación y desencriptación es crítica, como en servidores de alto rendimiento o aplicaciones en tiempo real, C# y Go son claramente las opciones preferidas. C# es ideal cuando se necesita un control más preciso del hardware y la memoria, mientras que Go ofrece ventajas únicas en aplicaciones que requieren manejo concurrente.

### Prototipos y scripts rápidos: Python sigue siendo adecuado para prototipos, pruebas o scripts rápidos donde la velocidad no es un factor decisivo. Es especialmente útil en situaciones donde el tiempo de desarrollo y la legibilidad del código tienen prioridad.

### Aplicaciones web y móviles: A pesar de sus limitaciones, JavaScript sigue siendo una opción sólida para aplicaciones a gran escala, especialmente en el desarrollo web y móvil, donde la seguridad y la escalabilidad son tan importantes como el rendimiento.

### **Expansiones**

### Go en archivos pequeños: Los resultados de Go en archivos pequeños (1 MB) fueron particularmente impresionantes, superando incluso a C#. Esto se debe a la eficiencia de su compilador y a la forma en que maneja las operaciones de memoria y procesamiento concurrente. Esto sugiere que Go podría ser especialmente útil en aplicaciones que requieren el procesamiento rápido de grandes cantidades de pequeños bloques de datos.

### JavaScript y su rendimiento: Es interesante observar que JavaScript tuvo tiempos de encriptación más lentos que Python en ciertas pruebas. Esto podría deberse a la forma en que los motores de JavaScript gestionan las operaciones criptográficas y las limitaciones inherentes a la ejecución en una máquina virtual. Sería beneficioso explorar optimizaciones adicionales o incluso considerar el uso de otros lenguajes para tareas críticas en rendimiento.

### 

### **Sugerencias**

1. **Sistemas de alto rendimiento**: Usar C# y Go para tareas críticas en rendimiento.
2. **Prototipos y scripts rápidos**: Python es adecuado donde la simplicidad y velocidad de desarrollo son prioritarias.
3. **Aplicaciones web**: JavaScript sigue siendo relevante, pero sus limitaciones en rendimiento deben considerarse.

### **Referencias bibliográficas**

**Libros y artículos sobre encriptación y criptografía:**

* Stallings, W. (2017). *Cryptography and Network Security: Principles and Practice*. Pearson.
* Schneier, B. (2015). *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons.
  + Bruce Schneier es un experto en criptografía y en este libro discute cómo implementar varios algoritmos criptográficos, con un enfoque en C.

### **Artículos sobre rendimiento en lenguajes de programación:**

* + Rother, C., & Schlitt, H. (2021). *Performance Comparison of Programming Languages*. *Journal of Computer Science*, 35(4), 289-301.

### **Documentación de bibliotecas de criptografía utilizadas:**

* PyCryptodome:<https://www.pycryptodome.org/>
* Crypto++:<http://www.cryptopp.com/>
* javax.crypto para JavaScript:<https://docs.oracle.com/javase/8/docs/api/javax/crypto/package-summary.html>

### **Anexos y apéndice**

[1]: **AES (Advanced Encryption Standard)** en modo CBC (Cipher Block Chaining), un algoritmo ampliamente utilizado en la industria por su equilibrio entre seguridad y velocidad.

[2]:**11th Gen Intel(R) Core(TM) i9-11900KF @ 3.50GHz (16CPUs)**

**Códigos:**

***Python:***

from Crypto.Cipher import AES

import time

from Crypto.Util.Padding import pad, unpad # Para asegurar que los datos tengan un tamaño múltiplo de 16 bytes

# Clave y cifrador en modo CBC

key = b'Sixteen byte key'

cipher\_encrypt = AES.new(key, AES.MODE\_CBC)

cipher\_decrypt = AES.new(key, AES.MODE\_CBC, iv=cipher\_encrypt.iv) # Usa el mismo IV para descifrar

# Tamaño del archivo en MB

file\_size\_mb = 10 # Cambia a 1, 10, 100, etc., según el tamaño deseado

# Datos a encriptar y asegurar que el tamaño sea múltiplo de 16 bytes

data = b'0' \* (file\_size\_mb \* 1024 \* 1024) # Genera `file\_size\_mb` MB de datos

padded\_data = pad(data, AES.block\_size)

print(f"Encriptando y desencriptando un archivo de {file\_size\_mb} MB...")

# Función de encriptación

def encrypt(data):

return cipher\_encrypt.encrypt(data)

# Función de desencriptación

def decrypt(encrypted\_data):

return cipher\_decrypt.decrypt(encrypted\_data)

# Medir el tiempo de encriptación

start\_time = time.time()

encrypted = encrypt(padded\_data)

end\_time = time.time()

print(f"Tiempo de encriptación: {end\_time - start\_time:.6f} segundos")

# Medir el tiempo de desencriptación

start\_time = time.time()

decrypted = unpad(decrypt(encrypted), AES.block\_size)

end\_time = time.time()

print(f"Tiempo de desencriptación: {end\_time - start\_time:.6f} segundos")

***C#:***

using System;

using System.Diagnostics;

using System.IO;

using System.Security.Cryptography;

class Program

{

static void Main()

{

// Definimos el tamaño del archivo en MB

int fileSizeMB = 1; // Cambiar a 1 MB

byte[] data = new byte[fileSizeMB \* 1024 \* 1024]; // Crear datos de 1 MB

new Random().NextBytes(data); // Rellenar con datos aleatorios

// Crear una clave y un IV para AES

using (Aes aes = Aes.Create())

{

aes.KeySize = 128; // Usar AES-128

aes.GenerateKey(); // Generar una clave aleatoria

aes.GenerateIV(); // Generar un IV aleatorio

// Medición de tiempo de encriptación

Stopwatch stopwatch = Stopwatch.StartNew();

byte[] encryptedData;

using (var encryptor = aes.CreateEncryptor(aes.Key, aes.IV))

{

encryptedData = encryptor.TransformFinalBlock(data, 0, data.Length);

}

stopwatch.Stop();

Console.WriteLine($"Tiempo de encriptación: {stopwatch.Elapsed.TotalSeconds} segundos");

// Medición de tiempo de desencriptación

stopwatch.Restart();

byte[] decryptedData;

using (var decryptor = aes.CreateDecryptor(aes.Key, aes.IV))

{

decryptedData = decryptor.TransformFinalBlock(encryptedData, 0, encryptedData.Length);

}

stopwatch.Stop();

Console.WriteLine($"Tiempo de desencriptación: {stopwatch.Elapsed.TotalSeconds} segundos");

}

}

}

***JavaScript :***

* **Archivo .JS**

async function createRandomData(size) {

const chunkSize = 65536; // Tamaño del bloque

const numChunks = Math.ceil(size / chunkSize);

const data = new Uint8Array(size);

for (let i = 0; i < numChunks; i++) {

const randomValues = new Uint8Array(Math.min(chunkSize, size - i \* chunkSize));

window.crypto.getRandomValues(randomValues);

data.set(randomValues, i \* chunkSize);

}

return data;

}

async function encryptData(data, key) {

const iv = window.crypto.getRandomValues(new Uint8Array(16)); // Vector de inicialización

const cipher = await window.crypto.subtle.importKey('raw', key, 'AES-CBC', false, ['encrypt']);

const encrypted = await window.crypto.subtle.encrypt({ name: 'AES-CBC', iv: iv }, cipher, data);

return { encrypted, iv };

}

async function decryptData(encryptedData, key, iv) {

const cipher = await window.crypto.subtle.importKey('raw', key, 'AES-CBC', false, ['decrypt']);

const decrypted = await window.crypto.subtle.decrypt({ name: 'AES-CBC', iv: iv }, cipher, encryptedData);

return new Uint8Array(decrypted);

}

async function main() {

const sizeMB = 1;

const size = sizeMB \* 1024 \* 1024; // Convertir MB a bytes

const key = window.crypto.getRandomValues(new Uint8Array(16)); // Clave AES de 16 bytes

// Crear datos aleatorios

const data = await createRandomData(size);

console.log(`Datos de ${sizeMB} MB creados.`);

// Medir tiempo de encriptación

const startEncrypt = performance.now();

const { encrypted, iv } = await encryptData(data, key);

const endEncrypt = performance.now();

console.log(`Tiempo de encriptación: ${(endEncrypt - startEncrypt) / 1000} segundos`);

// Medir tiempo de desencriptación

const startDecrypt = performance.now();

const decryptedData = await decryptData(encrypted, key, iv);

const endDecrypt = performance.now();

console.log(`Tiempo de desencriptación: ${(endDecrypt - startDecrypt) / 1000} segundos`);

}

// Ejecutar la función principal

main().catch(console.error);

* **Archivo .HTML**

<!DOCTYPE html>

<html lang="es">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Encriptar y Desencriptar</title>

</head>

<body>

<h1>Encriptar y Desencriptar AES en JavaScript</h1>

<script src="encrypt\_decrypt.js"></script>

</body>

</html>

***GO:***

package main

import (

"crypto/aes"

"crypto/cipher"

"fmt"

"time"

)

func main() {

fileSizeMB := 10

data := make([]byte, fileSizeMB\*1024\*1024) // Datos de `fileSizeMB` MB

key := []byte("examplekey123456")

block, err := aes.NewCipher(key)

if err != nil {

panic(err)

}

fmt.Printf("Encriptando y desencriptando un archivo de %d MB...\n", fileSizeMB)

// Medición de tiempo de encriptación

start := time.Now()

encrypter := cipher.NewCBCEncrypter(block, key[:aes.BlockSize])

encrypter.CryptBlocks(data, data)

fmt.Printf("Tiempo de encriptación: %v segundos\n", time.Since(start).Seconds())

// Medición de tiempo de desencriptación

start = time.Now()

decrypter := cipher.NewCBCDecrypter(block, key[:aes.BlockSize])

decrypter.CryptBlocks(data, data)

fmt.Printf("Tiempo de desencriptación: %v segundos\n", time.Since(start).Seconds())

}