UNIVERSIDAD DE SONORA

"El Saber de mis Hijos hará mi Grandeza"

Universidad de Sonora

División de ingeniería

Ingeniería Mecatrónica

Examen 02

Materia: Ciberseguridad

Integrante:

García Saavedra Ian Alan - 221207167

Hora de clase:

Hermosillo, Sonora a 03 abril del 2025

Introducción

El análisis y procesamiento de imágenes se ha convertido en una herramienta esencial en diversas aplicaciones tecnológicas, y en este proyecto, se explora el uso de un microcontrolador ESP32 para realizar detección de bordes en imágenes. Para abordar este desafío, se comienza con una imagen encriptada que oculta un mensaje, el cual es extraído mediante técnicas de criptografía. Una vez desencriptado el mensaje, se instruye al sistema para descargar la imagen, procesarla en un computador, enviarla al ESP32 para la detección de bordes y finalmente recuperar la imagen procesada para su visualización en una pantalla. Este enfoque integra técnicas de seguridad, procesamiento de imágenes y programación en microcontroladores para lograr un sistema eficiente y funcional.

Dado que las imágenes pueden ser de gran tamaño y difíciles de manejar, el proyecto emplea un método de particionamiento, dividiendo la imagen en fragmentos más pequeños (16x16 subdivisiones) antes de enviarlos al ESP32. El microcontrolador realiza la detección de bordes utilizando el operador Sobel y envía las partes procesadas de vuelta al computador para su reconstrucción. Posteriormente, se aplica un proceso de posprocesamiento que incluye mejoras en la resolución, el contraste y la nitidez para obtener una imagen final de calidad superior. Este enfoque modular permite una manipulación eficiente de las imágenes, manteniendo un balance entre la calidad del procesamiento y la capacidad de manejo del microcontrolador.

Desarrollo

Este fue la imagen que se nos dio con un peso de 13,107,200 bytes o 12.5 MB por lo que se tuvo que hacer una partición y un preprocesado para poder hacer que la imagen pueda ser procesada para la detección de bordes en el ESP32. A continuación mostrare los códigos que utilice

Desencriptador.py: Este código se utilizo para desencriptar la imagen y poder revelar el mensaje secreto el cual nos decía “Download this image to your computer, load it to your microcontroller and perform border detection, then retrieve the image and display it in your screen. Deliverables are the block diagram for the methodology, algorithm and a picture of the system working.”

```
from PIL import Image
import sys

# Abrir archivo para guardar el mensaje desencriptado
with open("mensaje_desencriptado.txt", "w") as output_file:
    # Redirigir la salida estándar al archivo
    sys.stdout = output_file

    # Cargar la imagen encriptada
    image = Image.open("../images/encrypted_logo.png")
    image = image.convert("RGB")
    pixels = image.load()

    # Extraer mensaje del LSB del canal azul
    binary_message = ""
    message = ""

    for y in range(image.height):
        for x in range(image.width):
            r, g, b = pixels[x, y]
            binary_message += str(b & 1)  # Extrae el bit menos significativo del
azul

            # Procesar caracteres de 8 en 8 bits
            if len(binary_message) % 8 == 0:
                byte = binary_message[-8:]  # Últimos 8 bits
                if byte == "00000000":  # Señal de fin de mensaje
                    break
                message += chr(int(byte, 2))

                # Mostrar el mensaje progresivamente
                sys.stdout.write(f"\rMensaje desencriptado: {message}")
                sys.stdout.flush()

        else:
            continue  # Solo se ejecuta si el bucle interno no se interrumpe
        break  # Rompe el bucle externo si se encontró el byte de terminación

    sys.stdout.write("\nProceso terminado.")
```

ImageSplitter.py: Luego parti la imagen original en 16x16 divisiones para poder procesarlas y le di un preprocesador para que el peso no sea tan grande y que pueda ser enviado al ESP32.

```
from PIL import Image
import os

# Parámetros
IMAGE_PATH = "../images/imagen_original.png"  # Ruta de la imagen original
OUTPUT_FOLDER = "../results"  # Carpeta donde se guardarán las particiones
NUM_PARTS = 16  # Número de divisiones en cada eje (32x32 = 1024 imágenes)
RESIZE_FACTOR = 2  # Factor de reducción (1 = no reducir, 2 = reducir a la mitad)
OUTPUT_FORMAT = "JPEG"  # Formato de salida ("JPEG", "PNG", etc.)
OUTPUT_QUALITY = 50  # Calidad de compresión (solo para JPEG)

def split_image(image_path, output_folder, num_parts, resize_factor=1,
output_format="JPEG", output_quality=10):
    """Divide una imagen en partes y las guarda en una carpeta."""

    # Crear la carpeta si no existe
    os.makedirs(output_folder, exist_ok=True)

    # Cargar la imagen en escala de grises
    img = Image.open(image_path).convert("L")

    # Reducir la resolución si es necesario
    if resize_factor > 1:
        img = img.resize((img.width // resize_factor, img.height //
resize_factor))

    # Obtener tamaño de la imagen ajustada
    width, height = img.size
    part_width = width // num_parts
    part_height = height // num_parts

    # Cortar la imagen en partes
    for i in range(num_parts):
        for j in range(num_parts):
            left = j * part_width
            upper = i * part_height
            right = left + part_width
            lower = upper + part_height

            # Recortar la subimagen
            img_part = img.crop((left, upper, right, lower))
```

```
            # Construir la ruta del archivo de salida
            output_filename = f"part_{i}_{j}.{output_format.lower()}"
            output_path = os.path.join(output_folder, output_filename)

            # Guardar la imagen fragmentada con opciones de formato
            if output_format == "JPEG":
                img_part.save(output_path, output_format, quality=output_quality,
optimize=True)
            else:
                img_part.save(output_path, output_format)

    print(f"Total de imágenes generadas: {num_parts * num_parts}")

# Ejecutar la función con los parámetros ajustados
split_image(IMAGE_PATH, OUTPUT_FOLDER, NUM_PARTS, RESIZE_FACTOR, OUTPUT_FORMAT,
OUTPUT_QUALITY)
```

Sender&Receiver: Este código lo que hace es mandar cada subdivion y esperar a que este lista para cuando sea procesada por el ESP32.

```
import serial
import time
import argparse
import numpy as np
from PIL import Image
import os
import glob

# Parámetros por defecto
DEFAULT_SEND_IMAGE_PATH = "../results/part_1_1.jpeg"  # Solo para referencia
SERIAL_PORT = "COM3"                                   # Puerto serial al que
está conectado el ESP32
BAUD_RATE = 115200                                     # Velocidad de baudios
WAIT_BOOT_LOGS = 2                                     # Tiempo para esperar boot
logs del ESP32

# Función que procesa una imagen enviándola al ESP32 y recibiendo la imagen
procesada
def process_image(image_path, output_path):
    print(f"\nProcesando imagen: {image_path}")
    # Abrir la imagen y preprocesarla
    img = Image.open(image_path)
    img = img.convert("L")  # Convertir a escala de grises
    img = img.resize((80, 63))  # Redimensionar a 80x63
```

```
    img_bytes = img.tobytes()

    # Verificar que el tamaño sea correcto
    if len(img_bytes) != 80 * 63:
        print(f"Error: La imagen procesada tiene {len(img_bytes)} bytes, pero se
esperaban {80 * 63}.")
        return False

    # Abrir conexión serial
    ser = serial.Serial(SERIAL_PORT, BAUD_RATE, timeout=None)
    time.sleep(2)  # Esperar a que el ESP32 esté listo
    ser.reset_input_buffer()

    # Enviar la imagen al ESP32
    print("Enviando imagen al ESP32...")
    ser.write(img_bytes)
    ser.flush()

    # Esperar confirmación del ESP32
    while True:
        if ser.in_waiting > 0:
            response = ser.readline().decode("utf-8").strip()
            print(f"Respuesta del ESP32: {response}")
            if response == "OK":
                break

    # Recibir la imagen procesada
    print("Esperando imagen procesada...")
    processed_data = bytearray()
    while True:
        if ser.in_waiting > 0:
            chunk = ser.read(ser.in_waiting)
            processed_data.extend(chunk)
            if b"\nOK\n" in processed_data:
                processed_data = processed_data[:80 * 63]
                break

    # Reconstruir la imagen procesada
    processed_img = Image.frombytes("L", (80, 63), bytes(processed_data))
    processed_img.save(output_path)
    print(f"Imagen procesada guardada en: {output_path}")

    ser.close()
    return True
```

```python
if __name__ == "__main__":
    # Carpeta de entrada donde se encuentran las imágenes a procesar
    input_folder = os.path.join("..", "results")
    # Carpeta de salida donde se guardarán las imágenes procesadas
    output_folder = os.path.join("..", "final")
    os.makedirs(output_folder, exist_ok=True)

    # Buscar todas las imágenes en la carpeta ../results con extensión .jpeg
    image_paths = glob.glob(os.path.join(input_folder, "*.jpeg"))
    if not image_paths:
        print("No se encontraron imágenes en la carpeta ../results")
        exit(1)

    # Procesar cada imagen secuencialmente
    for path in sorted(image_paths):
        # Se omiten las imágenes que ya contengan "_processed" en su nombre
        base_name = os.path.basename(path)
        if "_processed" in base_name:
            continue
        name, ext = os.path.splitext(base_name)
        output_filename = f"{name}_processed{ext}"
        output_path = os.path.join(output_folder, output_filename)
        success = process_image(path, output_path)
        if not success:
            print(f"Error al procesar la imagen {path}.")
        else:
            print(f"Procesamiento de {path} completado.")
```

border_detection.ino: Aquí procesamos la imagen, aplicamos el algoritmo de detección de bordes y después lo devolvemos al código de Python que esta esperando a guardarlo de forma local en la computadora.

```
#define BAUD_RATE 115200

// Dimensiones de la imagen (ajusta según sea necesario)
#define IMAGE_WIDTH 80
#define IMAGE_HEIGHT 63
#define MAX_IMAGE_SIZE (IMAGE_WIDTH * IMAGE_HEIGHT)

// Umbral para detectar bordes (ajusta según la intensidad deseada)
#define THRESHOLD 100

#define TIMEOUT_MS 2000  // Timeout de 2 segundos
```

```
byte imageBuffer[MAX_IMAGE_SIZE];
int imageLength = 0;

void performBorderDetection(byte* data, int length) {
  // Se asume que la imagen ocupa toda la matriz y está en escala de grises.
  // Se crea un buffer temporal para la imagen procesada.
  byte output[MAX_IMAGE_SIZE];

  // Inicializa todo a negro.
  for (int i = 0; i < length; i++) {
    output[i] = 0;
  }

  // Se aplica el operador Sobel, omitiendo los bordes de la imagen.
  for (int y = 1; y < IMAGE_HEIGHT - 1; y++) {
    for (int x = 1; x < IMAGE_WIDTH - 1; x++) {
      int idx = y * IMAGE_WIDTH + x;

      int a = data[(y - 1) * IMAGE_WIDTH + (x - 1)];
      int b = data[(y - 1) * IMAGE_WIDTH + x];
      int c = data[(y - 1) * IMAGE_WIDTH + (x + 1)];
      int d = data[y * IMAGE_WIDTH + (x - 1)];
      // int e = data[y * IMAGE_WIDTH + x]; // no usado
      int f = data[y * IMAGE_WIDTH + (x + 1)];
      int g = data[(y + 1) * IMAGE_WIDTH + (x - 1)];
      int h = data[(y + 1) * IMAGE_WIDTH + x];
      int i = data[(y + 1) * IMAGE_WIDTH + (x + 1)];

      int gx = (-a) + c + (-2 * d) + (2 * f) + (-g) + i;
      int gy = (-a) + (-2 * b) + (-c) + g + (2 * h) + i;

      // Aproximación de la magnitud del gradiente.
      int grad = abs(gx) + abs(gy);

      // Si el gradiente es mayor al umbral, se marca como borde (blanco).
      output[idx] = (grad > THRESHOLD) ? 255 : 0;
    }
  }

  // Copia la imagen procesada de nuevo en data.
  for (int i = 0; i < length; i++) {
    data[i] = output[i];
  }
}
```

```
void setup() {
  Serial.begin(BAUD_RATE);
  // Espera a que se establezca la conexión Serial
  delay(1000);

  // Recibir la imagen enviada por el PC con un timeout ampliado.
  unsigned long lastReceiveTime = millis();
  while (true) {
    while (Serial.available() > 0) {
      if (imageLength < MAX_IMAGE_SIZE) {
        imageBuffer[imageLength++] = Serial.read();
      }
      lastReceiveTime = millis();
    }
    if (millis() - lastReceiveTime > TIMEOUT_MS) {
      break;
    }
  }

  // Enviar confirmación al PC indicando que la imagen fue recibida.
  Serial.println("OK");

  // Ejecutar la detección de bordes (en este ejemplo, usando el operador Sobel).
  performBorderDetection(imageBuffer, imageLength);

  // Enviar la imagen procesada de vuelta al PC.
  for (int i = 0; i < imageLength; i++) {
    Serial.write(imageBuffer[i]);
  }

  // Enviar el delimitador de fin de transmisión.
  Serial.print("\nOK\n");
}

void loop() {
  // No se requiere ejecución continua.
}
```

finalImage.py: Este código lo que hace es que une todas las imágenes ya después de ser procesadas para poder verlo de mejor forma.

```
import os
from PIL import Image, ImageEnhance, ImageFilter

# Carpeta de entrada con las 256 imágenes (fragmentos)
```

```python
input_folder = "../final"
# Ruta de salida para la imagen final postprocesada
output_path = "imagen_completa_postprocessed.jpeg"

# Recopilar archivos que sigan el patrón "part_<fila>_<columna>_processed.jpeg"
files = [f for f in os.listdir(input_folder) if f.endswith("_processed.jpeg")]

# Lista para almacenar (fila, columna, nombre de archivo)
parts = []
for f in files:
    try:
        base, _ = os.path.splitext(f)
        # Se espera un nombre en el formato: part_<fila>_<columna>_processed
        tokens = base.split("_")
        # tokens = ["part", fila, columna, "processed"]
        row = int(tokens[1])
        col = int(tokens[2])
        parts.append((row, col, f))
    except Exception as e:
        print(f"Se omite el archivo {f}: {e}")

if not parts:
    print("No se encontraron imágenes con el patrón esperado.")
    exit(1)

# Determinar la cantidad de filas y columnas a partir de los índices máximos
max_row = max(p[0] for p in parts)
max_col = max(p[1] for p in parts)
grid_rows = max_row + 1
grid_cols = max_col + 1

# Cargar una imagen de muestra para obtener el tamaño de cada parte
sample_path = os.path.join(input_folder, parts[0][2])
sample_img = Image.open(sample_path)
part_width, part_height = sample_img.size

# Crear una imagen nueva del tamaño total
total_width = grid_cols * part_width
total_height = grid_rows * part_height
final_img = Image.new("L", (total_width, total_height))

# Pegar cada fragmento en su posición correcta
for row, col, filename in parts:
    img_path = os.path.join(input_folder, filename)
    part_img = Image.open(img_path)
```

```
    final_img.paste(part_img, (col * part_width, row * part_height))

# --- Postprocesado ---
# 1. Upscaling para mejorar la resolución (se utiliza LANCZOS para alta calidad)
scale_factor = 2  # Puedes ajustar el factor de escalado
upscaled = final_img.resize((final_img.width * scale_factor, final_img.height *
scale_factor), Image.LANCZOS)

# 2. Mejorar contraste
enhancer = ImageEnhance.Contrast(upscaled)
upscaled = enhancer.enhance(1.2)  # Aumenta el contraste (ajusta según sea
necesario)

# 3. Aplicar filtro de nitidez
upscaled = upscaled.filter(ImageFilter.SHARPEN)

# Guardar la imagen final postprocesada
upscaled.save(output_path, "JPEG")
print("Imagen final guardada en:", output_path)
```

Conclusión

Este proyecto ha demostrado cómo integrar diversos componentes tecnológicos para llevar a cabo el procesamiento y análisis de imágenes de manera eficiente utilizando un microcontrolador ESP32. A través de la manipulación de imágenes encriptadas y su fragmentación, se logró implementar una solución que no solo desencripta y divide la imagen, sino que también realiza la detección de bordes en tiempo real. El uso del operador Sobel en el ESP32 permitió identificar los bordes con eficacia, a pesar de las limitaciones en el procesamiento de este tipo de dispositivos.

Además, la reconstrucción final de la imagen, una vez procesada y enviada de vuelta al computador, ha permitido obtener una imagen de alta calidad gracias a la postprocesación que mejora la resolución, el contraste y la nitidez. Este enfoque modular y escalable demuestra el potencial de los microcontroladores para manejar tareas complejas de procesamiento de imágenes, incluso en condiciones de recursos limitados. Finalmente, el proyecto no solo muestra cómo trabajar con imágenes, sino que también resalta la importancia de combinar la criptografía y el procesamiento en tiempo real para aplicaciones prácticas que requieren alta seguridad y eficiencia.

Imagen Resultado