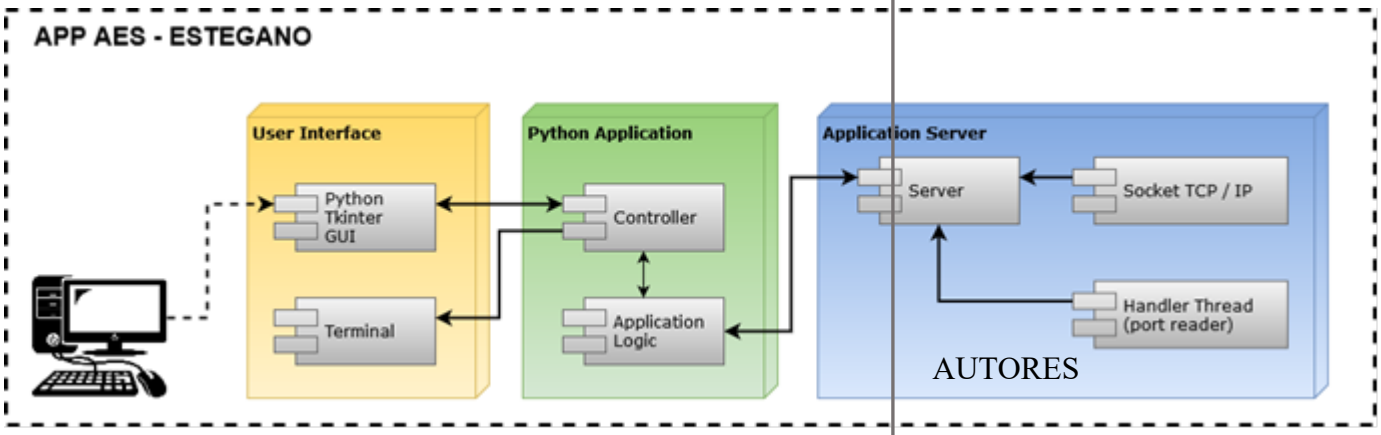
GITUD

GRUPO DE INVESTIGACIÓN DE TELECOMUNICACIONES DE LA UNIVERSIDAD DISTRITAL

AES STEGANO:

Herramienta criptográfica y esteganografía.

MANUAL TÉCNICO AES STEGANO



ELVIS EDUARDO GAONA
Docente Facultad Ingeniería
egaona@udistrital.edu.co

EDWAR JACINTO GÓMEZ.
Docente Facultad Tecnológica
ejacintog@udistrital.edu.co

THOMAS DANIEL AVILA
Ingeniero de Sistemas
tdavilab@correo.udistrital.edu.co

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Versión: 1.0.1

2021

TABLA DE CONTENIDO

I. INTRODUCCIÓN

La esteganografía se ha usado desde el siglo XV y con la era de la informática ha evolucionado bastante, permitiendo ocultar información dentro de los videos audios o imágenes de manera inadvertida a terceros y solo puede recuperarse por el destinatario legítimo. Es frecuente confundir la esteganografía con la criptografía, en lo único en que son similares, es que ambos son procesos que permiten la protección de la información, teniendo aplicaciones y objetivos diferentes. Por un lado, la criptografía se emplea para cifrar información de tal manera que sea ininteligible para los usuarios que no tengan la clave de cifrado y por el otro lado, la esteganografía tiene como objetivo ocultar la información sin que los usuarios se percaten de la existencia de un mensaje oculto. Combinando ambas técnicas, se obtiene un nivel de seguridad muy superior dándole mayor privacidad a los mensajes enviados.

La aplicación AES STEGANO está orientada a mostrar el funcionamiento de la combinación de ambas técnicas: un método de cifrado robusto como lo es AES (Advance Encryption Estándar) con un modo de operación CBC (Cipher Block Chaining) empleado en el mensaje a ocultar en una imagen empleando esteganografía. Este manual ofrece al usuario una descripción de la herramienta, estructura, funcionamiento, requerimientos y demás aspectos relevantes para una adecuada experiencia de usuario, derivada de un correcto y exitoso uso de la herramienta que logre cubrir las necesidades por las cuales se desarrolló la aplicación. Esta aplicación es parte de los resultados del proyecto de investigación HSM (hardware security module) para su uso en red de sensores en aplicaciones de gestión energética y robótica, financiado por el Centro de Investigación y desarrollo científico (**CIDC**).

En el repositorio de github (https://github.com/ElvisGaona/AES-STEGANO) está publicado los archivos necesarios para ejecutar el software, copia de los manuales técnicos y del programador y el registro en la Dirección Nacional de Derecho de Autor.

II. ÁMBITO DEL SISTEMA

AES STEGANO es una aplicación que permite el intercambio de imágenes con contenido secreto en su interior, utilizando técnicas criptográficas denominada esteganografía. El software permite enviar un archivo de texto oculto en una imagen utilizando criptografía y esteganografía, insertando los datos en el bit menos significativo del valor numérico de cada pixel de la imagen.

El cifrado del mensaje se realiza usando AES en su modo de operación CBC, y para el intercambio de la clave de manera segura, se utiliza el algoritmo de cifrado asimétrico RSA (Rivest, Shamir y Adleman), logrando un intercambio fiable de la clave agregando una capa de seguridad adicional al método.

1. DIAGRAMA DEL SISTEMA

En la figura 1 se muestra la estructura de la aplicación por bloques, compuesta por una interfaz de usuario, la aplicación en Python y el servidor de aplicaciones.

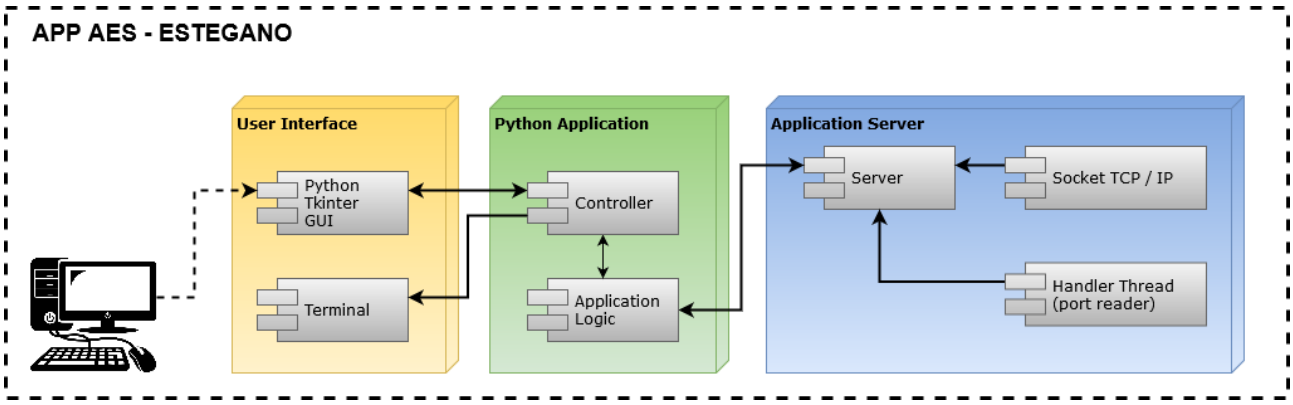


Figura 1. Diagrama del sistema

2. DESCRIPCIÓN DEL SISTEMA

La aplicación *AES STEGANO*, cuenta con archivos ejecutables para los sistemas operativos en Windows o Linux únicamente, los cuales funcionan sin necesidad de instalar software adicional. El usuario puede compilar la aplicación a partir del código fuente, por lo que se recomienda cumplir con los requerimientos de software y librerías relacionados en el manual técnico. En la figura 2 se muestra a manera de bloques la estructura funcional de la aplicación.

3. DIAGRAMA JERÁRQUICO DEL SISTEMA

La aplicación AES ESTEGANO cuenta con cinco módulos: el primero es el módulo de conexión que permite realizar la conexión al servidor e iniciarlo; el módulo de envío de imágenes tiene la funcionalidad de adjuntar el archivo secreto, la imagen original, el algoritmo de cifrado, enviar la imagen nueva y guardarla; el módulo de recepción de imágenes recibe la imagen, inicia el algoritmo de descifrado y guarda el archivo secreto; el módulo de esteganografía se encarga de emplear el método LSB (Least Significant Bit) para la codificación y decodificación de la imagen, finalmente el módulo de seguridad encargado de realizar el cifrado del mensaje empleado AES CBC.

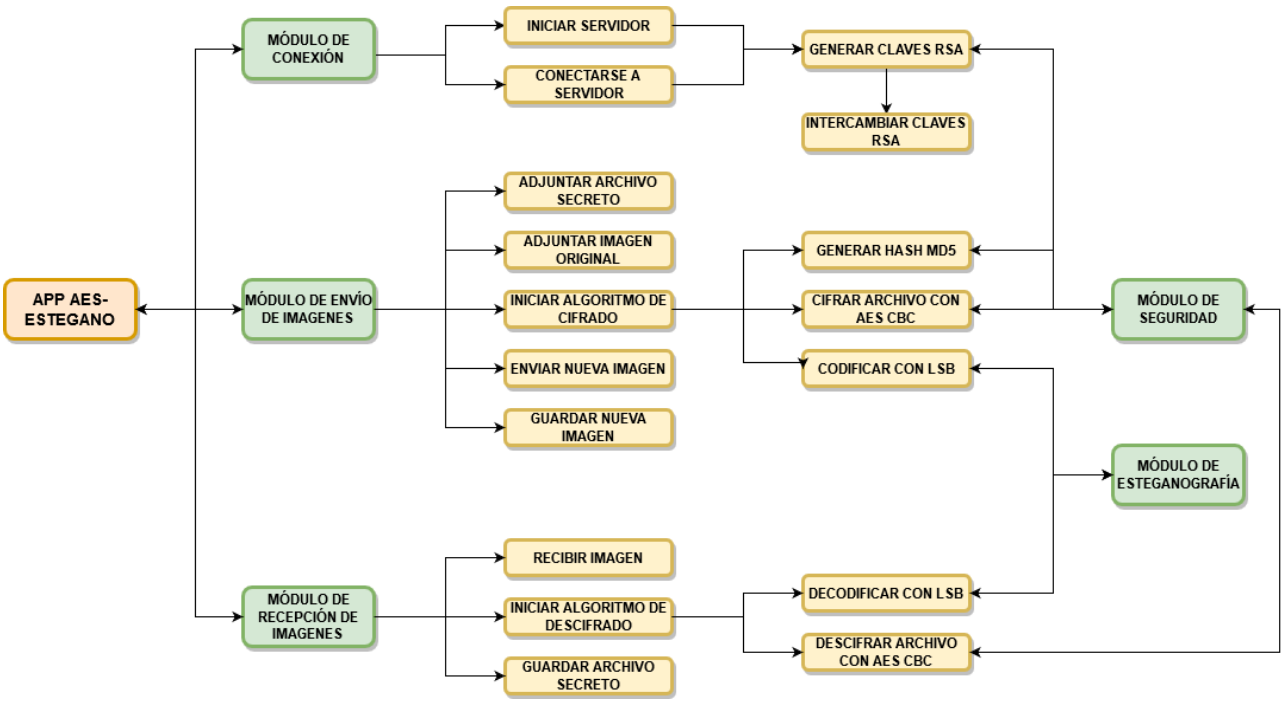


Figura 2. Diagrama jerárquico del sistema

4. REQUERIMIENTOS DEL SISTEMA

El sistema requiere para su funcionamiento en la aplicación ejecutable:

- Un computador o dos computadores con conexión local.
- Sistema operativo Windows 10 o Linux.
- Memoria RAM de al menos 4GB
- 44MB de espacio de almacenamiento en el disco duro.

En caso de compilar la aplicación a partir del código fuente:

- Permisos de Administrador en la cuenta de usuario del sistema operativo.
- Python 3.8 o superior instalado en el sistema operativo.
- Pip en la versión 19.2.3 o superior.
- Virtualenv en la versión 20.0.25 o superior.

Librerías:

- opencv-python
- pillow
- numpy
- pickle-mixin
- pycryptodome

5. MÓDULO DE CONEXIÓN

La conexión entre los dos Peers se realiza mediante sockets, a partir de unos métodos de una clase llamada AppSocket en el archivo appsocket.py. Para que la conexión sea realizada satisfactoriamente, primero un peer debe iniciarse como servidor, y luego el otro debe conectarse a este servidor. Para garantizar la comunicación segura de los archivos, al iniciar estos servicios se va a realizar la generación de claves RSA en cada peer, y posteriormente el intercambio de claves entre estos.

Estos dos sistemas son bidireccionales, por lo que no interesa quien sea el que envía o reciba la imagen. Esto es posible ya que en la clase Facade existe un método llamado escuchar_mensajes, el cual es llamado cada iteración del hilo principal. Este método va a estar escuchando todo el tiempo para verificar si el otro peer ha enviado algún mensaje.

Estos mensajes pueden ser de dos tipos, fijos como una clave RSA pública, o variables como la imagen, por lo que en el método se va a llevar un control del paso de comunicación en el que se encuentra. En la figura 3 se muestran los campos a completarse para la conexión entre los terminales que se van a transferir la imagen.

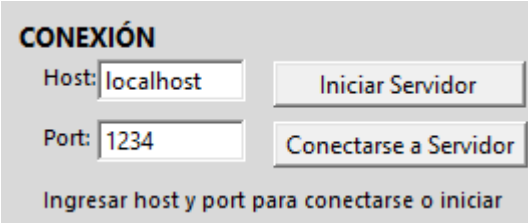


Figura 3. Módulo conexión

6. MÓDULO DE CIFRADO Y ENVÍO

En la figura 4 se muestra la sección que permite cifrar y enviar la imagen, donde el usuario selecciona y guarda el archivo con la imagen para posteriormente cifrar el mensaje y enviarlo al terminal de destino.

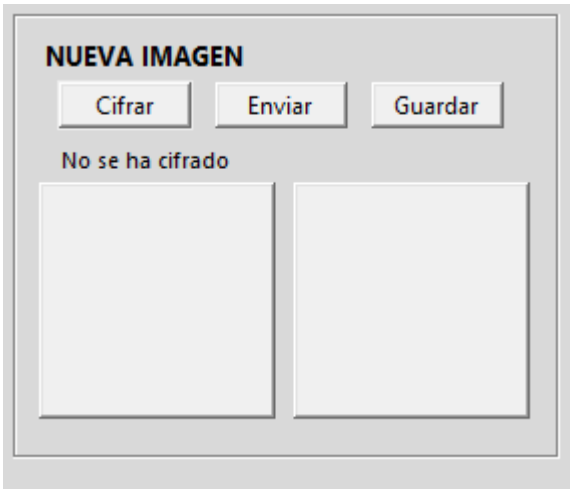


Figura 4. Módulo de Envío de Imágenes y Cifrado

6.3. ALGORITMO DE CODIFICACIÓN DE LA IMAGEN

6.3.1 ALGORITMO HASH MD5

Desde la clase Facade se obtiene la clave en texto plano ingresada por el usuario en la interfaz y se utiliza el método hashear_clave de la clase AESCifrador en algoritmos_seguridad.py.

```
# CLAVE DE SESIÓN
# Obtiene la clave de sesion de la interfaz
clave = self.gui.obtener_clave()

# Hashea la clave de sesión
self.cifrador.hashear_clave(clave)
```

```
# Obtiene el hash de la clave de sesión a partir del texto ingresado
def hashear_clave(self, txt):
    # Obtiene la sal a partir de una cadena de caracteres aleatoria del tamaño del texto
    salt = ''.join([chr(random.randint(65, 122)) for i in range(len(txt))])
    # Realiza la operación XOR entre la sal y el texto
    key = [chr(ord(a) ^ ord(b)) for a,b in zip(txt, salt)]
    # Pasa a bytes la nueva cadena de caracteres
    key = bytes(''.join(key),"utf-8")
    # Obtiene el digest de 16 bytes
    print(f"    [*] Clave con Sal: {key}")
    digest = hashlib.md5(key).digest()
    #hexdigest = hashlib.md5(txt).hexdigest() # 32 character hexadecimal
    print(f"    [*] Digest: {digest}")
    self.clave_sesion = digest
    return self.clave_sesion
```

Este método obtiene la sal a partir de una cadena de caracteres aleatoria del tamaño de la clave ingresada, y luego realiza la operación XOR entre la sal y la clave en texto plano. La cadena de caracteres resultante es del mismo tamaño de la clave ingresada, y esta se pasa a bytes para que la pueda recibir el algoritmo md5. Posteriormente se usa el método md5 de la librería hashlib para obtener el digest de 16 bytes utilizado como clave de sesión del algoritmo AES CBC.

6.3.2 CADENA ALEATORIA

En esta fase del algoritmo, se busca generar una cadena de caracteres aleatoria para concatenarla con el texto plano. Primero se define el separador entre el texto plano y la cadena aleatoria, en este caso serán cinco signos “=”.

```
plain_text += b"====="
```

Luego se obtiene la longitud que tendrá la cadena aleatoria a partir de la diferencia entre el tamaño disponible de la imagen y la longitud de la información, es decir, la longitud del texto plano, clave pública (450 bytes), clave de sesión cifrada (256 bytes) y última subclave cifrada (256 bytes).

```
length_random_str = self.n_bytes - (len(plain_text) + 512 + 450)
```

En este paso se presenta un problema, ya que el algoritmo AES siempre generará un texto cifrado de un tamaño de bytes múltiplo de 16, mayor al tamaño del texto plano original. Es decir, que si se utiliza este tamaño de cadena aleatoria, el texto cifrado resultante será mayor que la capacidad de la imagen y no funcionará el algoritmo.

Para solucionar esto, se utiliza el siguiente fragmento de código, el cual disminuirá el tamaño de la cadena aleatoria hasta que la suma del tamaño de esta cadena con el texto sea múltiplo de 16. Luego del ciclo, se le resta uno más a la cadena para que sea múltiplo de 15. De esta manera, el texto cifrado de AES tendrá el tamaño más uno del texto plano ingresado, y será de 0 a 16 bytes menor que el tamaño total de la imagen.

```
while (length_random_str+len(plain_text)) % 16 != 0:
    length_random_str -= 1
length_random_str -= 1
```

Luego se procede a generar la cadena aleatoria de caracteres del tamaño especificado:

```
# Genera cadena aleatoria de caracteres
random_str = np.zeros(length_random_str, dtype=np.uint8)
random_chain = []
for i in range(0,length_random_str):
    random_str[i] = random.randint(65, 122)
random_chain = ''.join(chr(i) for i in random_str)
```

Finalmente se concatena el texto plano con la cadena aleatoria. Este es el texto final que será la entrada al algoritmo AES CBC.

```
plain_text = plain_text + random_chain.encode("latin1")
```

Se ilustrará lo anterior mediante el siguiente ejemplo:

Se tiene una imagen de tamaño máximo de 121935 bytes y un texto plano de 23416 bytes, por lo que la cadena de caracteres aleatorios (relleno) inicial será de:

Longitud Relleno = 121935 – (23416+512+450) = 97557 bytes

Al juntar el texto plano con el relleno se tiene la siguiente cadena:

Longitud Final Texto = 97557 + 23416 = 120973 bytes

Si se usa el algoritmo AES con esta cadena resulta en un texto cifrado de 120976 bytes, ya que es el siguiente entero cuyo módulo 16 es 0 respecto al texto ingresado. Al realizar la suma de 120976+512+450 da como resultado 120938, lo cual es mayor que el tamaño de la imagen y por lo tanto no sirve. Para solucionar esto se fue restando la longitud del relleno hasta que la longitud final tenga un valor múltiplo de 16 menos 1:

Longitud Relleno = 97543 bytes

La longitud final del texto resultó la siguiente:

Longitud Final Texto = 97543 + 23416 = 120959 bytes

Si se usa el algoritmo AES con esta cadena resulta en:

Texto Cifrado: 120960 bytes % 16 = 0

Al realizar la suma de 120960+512+450 da como resultado 121922, lo cual es 13 bytes menor que el tamaño de la imagen. Se puede observar que este algoritmo siempre generará una longitud final de texto una unidad menor que la longitud del texto cifrado resultante. A su vez, se observa que el contenido a codificar en la imagen siempre será de 0 a 16 bytes menor que el tamaño total de la imagen.

6.3.3 ALGORITMO DE CIFRADO AES CBC

En esta parte del algoritmo se ejecuta el algoritmo AES CBC para cifrar, el cual se llama desde el Facade de la siguiente manera:

```
# ALGORITMO AES CBC
print("[+] Ejecutando AES CBC...")
# Se le envía al cifrador
self.cifrador.plain_text_Bytes = full_txt_relleno
# Cifra el texto con el relleno
texto_cifrado, iv = self.cifrador.cifrar_archivo()
```

El método del algoritmo utiliza la clave de sesión y el texto final que hace referencia al texto y el relleno; y retorna el texto cifrado y la última subclave.

```
def cifrar_archivo(self):
    cipher = AES.new(self.clave_sesion, AES.MODE_CBC)
    ct_bytes = cipher.encrypt(pad(self.plain_text_Bytes, AES.block_size))
    self.iv = cipher.iv
    self.texto_cifrado = ct_bytes
    return ct_bytes, cipher.iv
```

En la figura 5 se muestra el diagrama con las entradas y salidas de este algoritmo.

Figura 5. Entradas y salidas del algoritmo AES CBC

6.3.4 CIFRADO DE CLAVE DE SESIÓN Y ÚLTIMA SUBCLAVE

En esta parte del algoritmo se cifra la clave de sesión y la última subclave a partir del algoritmo RSA, el cual se llama desde el Facade de la siguiente manera:

```
# CIFRA INFORMACIÓN CON LA CLAVE PÚBLICA RSA
# Cifra la clave de sesión cs con la clave pública del destinatario
cs_c = self.apprsa.cifrar_clave(self.cifrador.clave_sesion)
# Cifra la última subclave iv con la clave pública del destinatario
iv_c = self.apprsa.cifrar_clave(iv)
```

Este algoritmo utiliza la clave pública del destinatario para cifrar la clave de sesión y la última subclave. De esta manera sólo es posible descifrar la información a partir de la clave privada del destinatario.

```
def cifrar_clave(self, clave):
    # Obtiene la clave RSA publica del destinatario
    recipient_key = RSA.import_key(self.otra_clave_publica)

    # Cifra la clave ingresada con la clave RSA publica del otro
    cipher_rsa = PKCS1_OAEP.new(recipient_key)
    clave_cifrada = cipher_rsa.encrypt(clave)
    return clave_cifrada
```

6.3.5 ALGORITMO DE ESTEGANOGRAFÍA (CIFRADO)

En esta sección se realiza el proceso de Esteganografía, es decir, codificar la información en la imagen ingresada. La siguiente línea de código en la clase Facade especifica el contenido que se guardará en la imagen:

```
full_content = self.apprsa.otra_clave_publica + cs_c + iv_c + texto_cifrado
```

Este contenido tiene el siguiente formato:

Clave de Sesión Cifrada **256 bytes**	Última Subclave Cifrada **256 bytes**	Texto Cifrado = AES(Texto+Relleno)

El llamado al método de codificación mediante Esteganografía se realiza en la siguiente línea de código:

```
encoded_image = self.encoder.encode(full_content)
```

El algoritmo utilizado es el LSB, el cual almacenará cada bit de la información secreta en el bit menos significativo de cada canal en cada pixel de la imagen. Primero se convierte el contenido a codificar a binario y almacena el tamaño de los datos a codificar:

```
# Convierte los datos a binario
binary_secret_data = self.to_bin(secret_data)

# Tamaño de los datos a ocultar
data_len = len(binary_secret_data)
```

Luego, se recorre cada fila, pixel y canal de la imagen, quitando el último bit del binario y agregando el de los datos:

```
for row in self.image:
    for pixel in row:
        for ch in range(len(pixel)):
            if data_index < data_len:
                pixel[ch] = int(self.to_bin(pixel[ch])[:-1]+binary_secret_data[data_index],2)
                data_index+=1
            else:
                break
return self.image
```

En el fragmento anterior, se realiza la suma del binario del valor del pixel en el canal sin su bit menos significativo, con el bit actual de los datos secretos. Este valor se le asigna al nuevo pixel:

```
pixel[ch] = int(self.to_bin(pixel[ch])[:-1]+binary_secret_data[data_index],2)
```

A su vez, se lleva un contador que indica el bit actual de los datos secretos.

```
data_index+=1
```

Este método retorna la nueva imagen generada, la cual contiene la información secreta en los bits menos significativos de sus pixeles. Esta nueva imagen presenta la característica que es imposible saber si tiene un mensaje secreto, ya que el análisis de la entropía de sus bits menos significativos no evidencia ningún patrón, ya que los datos están codificados mediante AES.

6.4. ENVÍO DE LA NUEVA IMAGEN

Para el envío de la imagen entre los dos Peers conectados, se hace uso de los métodos del archivo appsocket.py Primero, en la clase Facade se convierte la imagen en memoria, obtenida mediante le proceso de Esteganografía, en un string mediante línea de código:

```
img_str = cv2.imencode('.png', self.encoder.newimage)[1].tostring()
```

Luego, se hace el llamado al método enviar_mensaje de la clase AppSocket.

```
self.app.enviar_mensaje(img_str)
```

```
def enviar_mensaje(self, objeto):
    # Utiliza pickle para serializar el mensaje
    msg = pickle.dumps(objeto)
    # Agrega encabezado con el tamaño del mensaje
    msg = bytes(f'{len(msg):<{HEADERSIZE}}',"utf-8")+msg
    # Envía el mensaje al otro peer
    self.othersk.send(msg)
```

En este método se utiliza la librería pickle para serializar el mensaje, y luego se agrega el encabezado con el tamaño del mensaje. Este encabezado tiene un tamaño HEADERSIZE de 10 bytes. Finalmente se envía el mensaje al otro socket. Para recibir la imagen, desde el Facade siempre se está llamando al método escuchar_mensajes de la clase AppSocket. En este método se recibe la información de manera dinámica, en fragmentos de 16 bytes cada iteración.

Luego, se obtiene la longitud del mensaje a partir del encabezado. Cada iteración se va a agregar los 16 bytes actuales al total del mensaje. Finalmente cuando se llega al final del mensaje, se remueve el encabezado y se obtiene el objeto original mediante la librería pickle. Este objeto se retorna a la clase Facade.

Desde el Facade se convierte la string recibida a un arreglo de numpy:

```
nparr = np.fromstring(msg_bytes, np.uint8)
```

Luego, se decodifica el arreglo de numpy como una imagen cv2:

```
img = cv2.imdecode(nparr, cv2.IMREAD_COLOR)
```

Esta es la imagen que se utilizará en el proceso de descifrado, ya cual quedó cargada en memoria correctamente como un objeto de tipo cv2.

7. MÓDULO DE RECEPCIÓN Y DESCIFRADO

En la figura 6 se muestra la sección que le permite al usuario recibir la imagen, descifrar el archivo y guardarlo.

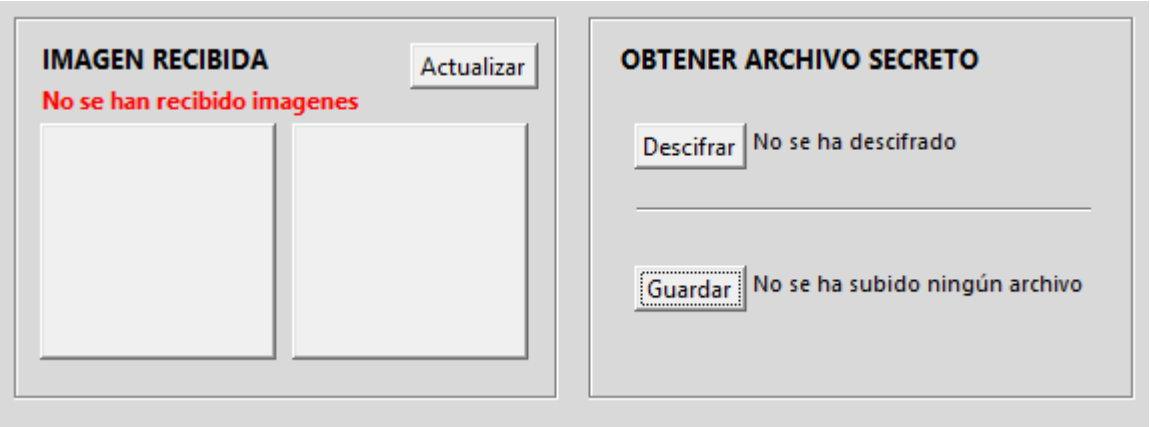


Figura 6. Módulo de recepción de imagen y descifrado

7.2. ALGORITMO DE DECODIFICACIÓN DE LA IMAGEN

7.2.1 ALGORITMO DE ESTEGANOGRAFÍA (DESCIFRADO)

```
# Valor para obtener el limite del texto cifrado original.
sum_mod = len(decoded_data_bytes)%16-2
# Obtiene texto cifrado
texto_cifrado = decoded_data_bytes[962:len(decoded_data_bytes)-sum_mod]
```

El fragmento anterior hace uso de un valor para obtener el límite del texto cifrado original a partir de la longitud de todo el paquete recibido. Esto es debido a que el algoritmo AES siempre resulta en un valor múltiplo de 16 y siempre sobraban de 0 a 16 espacios en el tamaño de la imagen original.

7.2.2 DESCIFRADO CON RSA

Se realiza el descifrado de la clave de sesión cifrada y la última subclave cifrada a partir de la clave privada local del destinatario.

```
# DESCIFRA INFORMACIÓN CON LA CLAVE PRIVADA RSA
# Descifra la clave de sesión cifrada
cs_d = self.apprsa.descifrar_clave(cs_c)
# Descifra la última subclave cifrada
iv_d = self.apprsa.descifrar_clave(iv_c)
```

```
def descifrar_clave(self, clave_cifrada):
    # Descifra la clave de sesión con la clave RSA privada
    p_key = RSA.import_key(self.clave_privada)
    cipher_rsa = PKCS1_OAEP.new(p_key)
    clave = cipher_rsa.decrypt(clave_cifrada)
    return clave
```

7.2.3 ALGORITMO DE DESCIFRADO AES CBC

Se procede a descifrar la información recibida y obtener el texto secreto a partir del algoritmo AES CBC, el cual recibe la clave de sesión y la última subclave descifradas anteriormente, como también el texto cifrado recibido.

```
    # Ejecuta el algoritmo de descifrado AES CBC y obtiene el texto plano original
    texto_descifrado = self.descifrador.descifrar_archivo()

def descifrar_archivo(self):
    cipher = AES.new(self.clave_sesion_descifrada, AES.MODE_CBC, self.iv_descifrado)
    self.texto_descifrado = unpad(cipher.decrypt(self.texto_cifrado), AES.block_size)

    return self.texto_descifrado
```

Al finalizar, se debe remover el relleno del texto plano a partir del separador establecido:

```
    # Remueve la cadena de texto aleatoria del texto plano
    self.descifrador.texto_descifrado = texto_descifrado.split(b"====")[0]
```

10. TIEMPO DE RESPUESTA DEL ALGORITMO

Finalmente, se prueba el tiempo de respuesta del algoritmo con una imagen que tiene una capacidad de almacenar 121935 bytes de información. Como el algoritmo utiliza el mecanismo para agregar la cadena aleatoria de texto, el algoritmo LSB almacena información en todos los bits menos significativos de la imagen. En la siguiente tabla se observa la comparación de tiempos de los principales procesos realizados en el algoritmo para esta prueba específica.

Proceso	Cifrado	Descifrado
Generación Claves RSA	0.16246 s	-
Hash md5	0.01567 s	-
AES CBC	0.01201 s	0.00077 s
Esteganografía LSB	1.84691 s	1.29945 s

El proceso de la Esteganografía es el que se demora más tiempo, el cual varía respecto a la capacidad de información que puede almacenar la imagen los bits menos significativos de sus pixeles en sus tres canales. En el proceso de la generación del hash, este tuvo un corto tiempo de ejecución debido a que sólo se hizo una vez el proceso de agregar la sal, pero en un entorno de producción, esta sal debe generarse en miles de iteraciones, por lo que su tiempo aumentaría.