# HWK5 – Cifrado César sobre TCP con ESP32 y Linux

Microprocesadores

Jaime Emmanuel Valenzuela Valdivia

14 de octubre de 2025

## 1. Objetivo

Implementar un flujo cliente–servidor por TCP que envía un mensaje cifrado con César (rotación de letras y dígitos), validarlo con `Wireshark` y un sniffer propio en Python, usando:

- ESP32 como cliente y servidor (según el paso).
- Binarios cliente/servidor en Linux.

## 2. Descripción del protocolo

El cliente envía un buffer con el siguiente formato:

`[1 byte shift][cadena_cifrada...]`

donde `shift` es el desplazamiento César. El servidor invierte la rotación para letras (módulo 26) y dígitos (módulo 10) y reconstruye el texto llano.

## 3. Entorno y comandos

Los comandos usados para compilar/ejecutar cada escenario fueron:

Listing 1: Resumen de comandos usados (ESP32 y Linux).

```
# -- ESP32 cliente / LINUX servidor --
. $HOME/esp/esp-idf/export.sh
cd ~/Documentos/CifradoCesar/esp/client
idf.py set-target esp32
idf.py menuconfig
idf.py build
idf.py -p /dev/ttyUSB0 flash monitor

# -- LINUX cliente / ESP32 servidor --
. $HOME/esp/esp-idf/export.sh
cd ~/Documentos/CifradoCesar/esp/server
idf.py set-target esp32
idf.py build
idf.py -p /dev/ttyUSB0 flash monitor

# -- Cliente Linux contra ESP (reemplazar IP_DEL_ESP) --
cd ~/Documentos/CifradoCesar/linux/build
./client <IP_DEL_ESP> 3333 4 Bren_123

# -- Wireshark --
tcp.port == 3333

```

```
# -- IP de la maquina --
hostname -I

# -- Sniffer Python (Scapy) --
cd ~/Documentos/CifradoCesar/python
sudo python3 sniff_caesar.py wlo1 3333
```

(Estos comandos están recopilados de la guía de trabajo que se siguió y los apuntes propios del laboratorio.[1])

# 4. Paso 1: ESP32 cliente → Linux servidor

## Configuración y compilación

En `Kconfig.projbuild` del cliente se definieron los parámetros:

- SSID/Password de la red Wi-Fi.
- IP y puerto del servidor TCP (en Linux).
- Texto y shift del cifrado.

Se realizó `idf.py build & flash`. Al iniciar, el ESP mostró su IP y la conexión al servidor.

## Evidencias

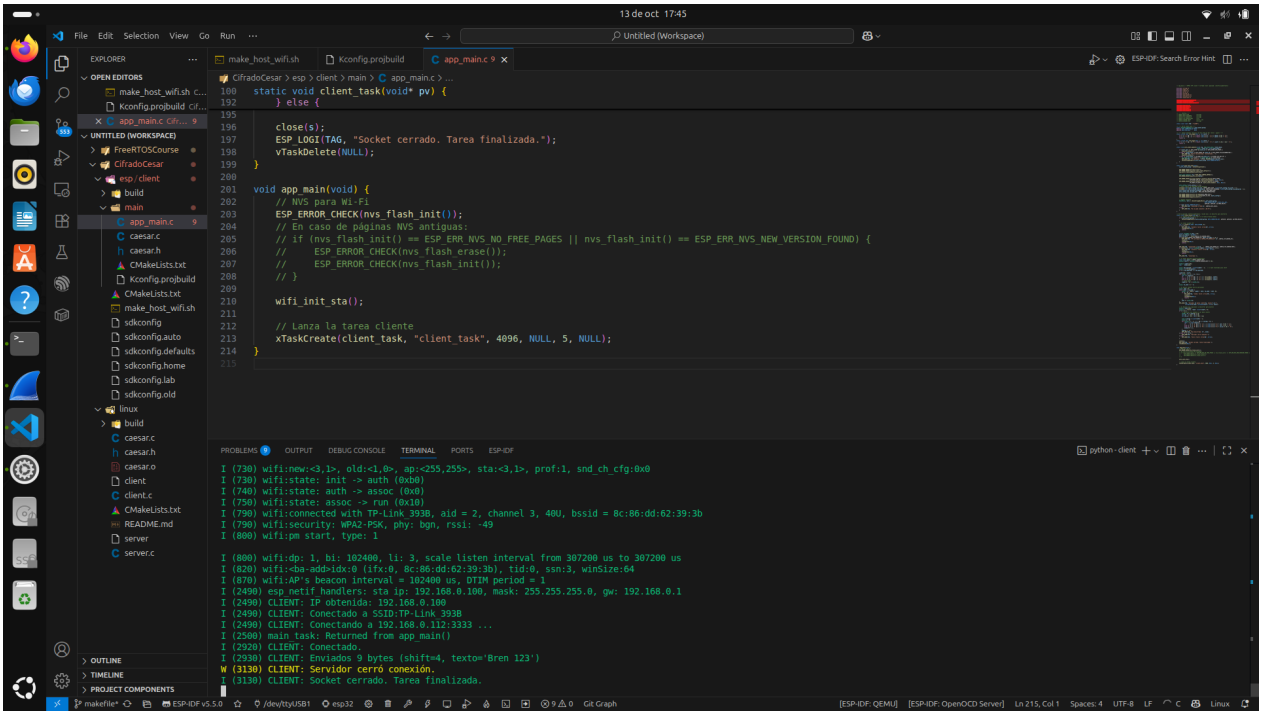


Figura 1: ESP32 **cliente**: conexión Wi-Fi, conexión TCP y envío de *"Bren_123"*/*"Jaime_123"*.

[1] Ver lista de comandos del alumno en *ESP32 cliente / LINUX server*. :contentReference[oaicite:0]index=0

```
jaime@JaimeMSI:~/Documentos/CifradoCesar/linux/build$ ./server 0.0.0.0 3333
[server] Escuchando en 0.0.0.0:3333 ...
[server] shift=4, descifrado="Bren 123"
jaime@JaimeMSI:~/Documentos/CifradoCesar/linux/build$ ./server 0.0.0.0 3333
[server] Escuchando en 0.0.0.0:3333 ...
[server] shift=4, descifrado="Bren 123"
jaime@JaimeMSI:~/Documentos/CifradoCesar/linux/build$ ./server 0.0.0.0 3333
[server] Escuchando en 0.0.0.0:3333 ...
[server] shift=4, descifrado="Bren 123"
jaime@JaimeMSI:~/Documentos/CifradoCesar/linux/build$
```

Figura 2: Servidor **Linux** escuchando en `0.0.0.0:3333`, mostrando `shift=4` y texto descifrado.

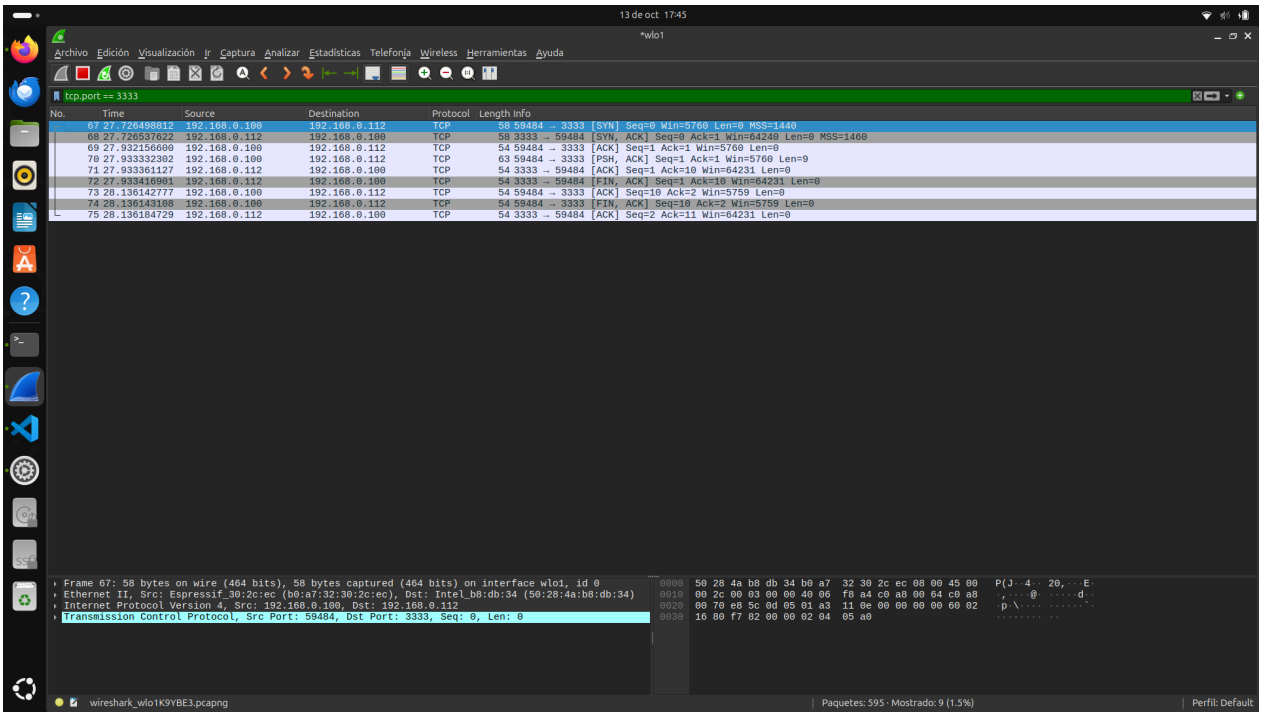


Figura 3: Captura Wireshark (`tcp.port == 3333`) para el flujo ESP→Linux.

# 5. Paso 2: Linux cliente → ESP32 servidor

## Configuración y compilación

Se construyó el proyecto servidor para ESP32 y se ejecutó el cliente Linux:

```
cd ~/Documentos/CifradoCesar/linux/build
./server 0.0.0.0 3333        # (para la prueba inversa con ESP cliente)
./client 192.168.0.100 3333 4 Bren_123
```

## Evidencias

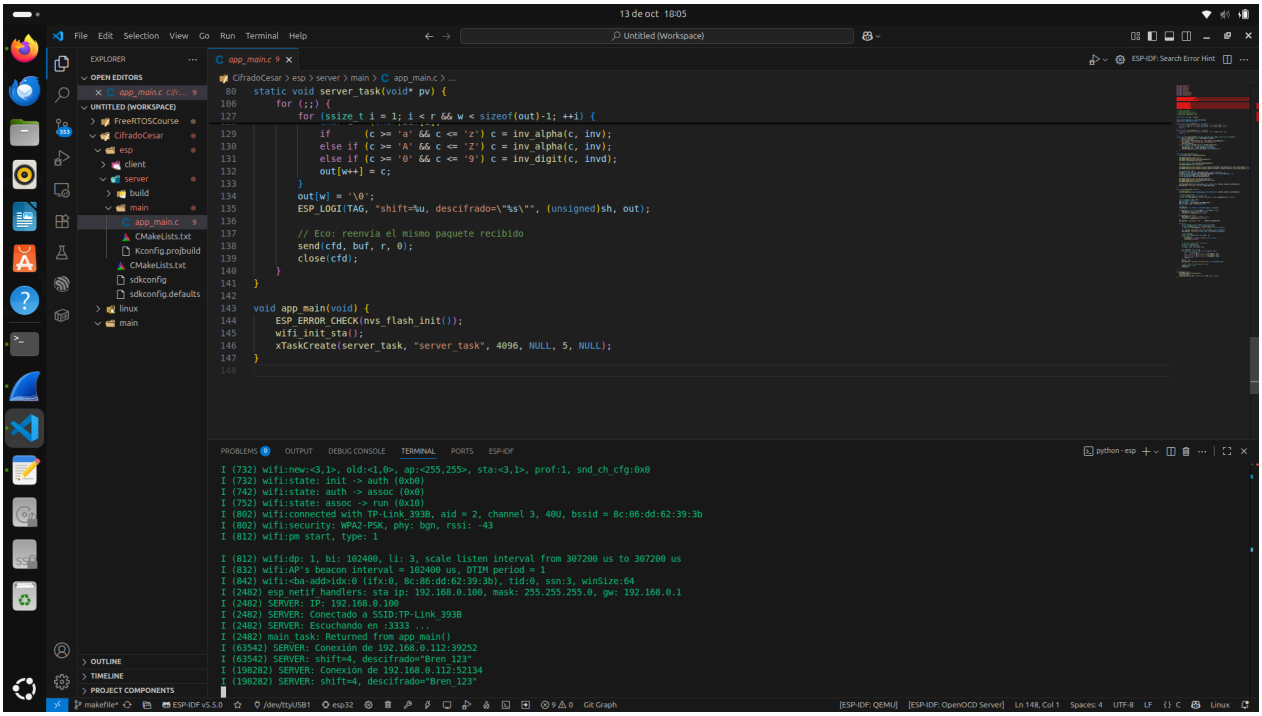


Figura 4: ESP32 **servidor**: IP adquirida y socket escuchando en `:3333`. Muestra conexiones entrantes y texto llano.


Figura 5: Cliente **Linux**: envío de 9 bytes (1 de `shift` + 8–9 de payload) al ESP servidor.

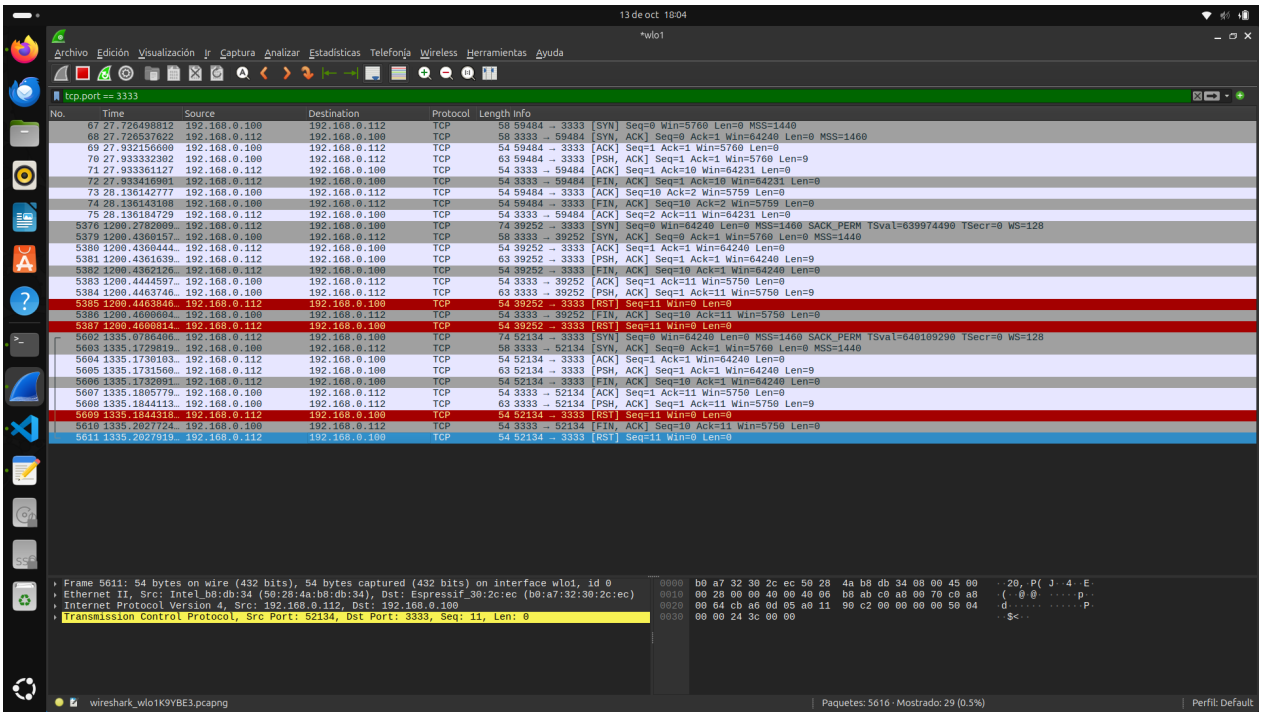


Figura 6: Wireshark del flujo Linux→ESP (handshake, `PSH,ACK` con `Len=9/10`, y cierre).

## 6. Sniffer en Python (Scapy)

Para validar la capa de aplicación, se implementó un sniffer que:

1. Captura `TCP` en el puerto 3333.
2. Interpreta el primer byte como `shift`.
3. Descifra el resto combinando rotación de letras (mód. 26) y dígitos (mód. 10).

Listing 2: sniff_caesar.py (Scapy).

```
from scapy.all import sniff, TCP, Raw, IP
import sys

# Descifrado Cesar: letras rotan 26, digitos rotan 10
def dec_caesar(shift, data: bytes) -> str:
    s = shift % 26
    sd = (shift % 10)
    inv = (26 - s) % 26
    invd = (10 - sd) % 10
    out = []
    for b in data:
        c = chr(b)
        if 'a' <= c <= 'z':
            out.append(chr(((ord(c)-97 + inv) % 26) + 97))
        elif 'A' <= c <= 'Z':
            out.append(chr(((ord(c)-65 + inv) % 26) + 65))
        elif '0' <= c <= '9':
            out.append(chr(((ord(c)-48 + invd) % 10) + 48))
        else:
            out.append(c)
    return ''.join(out)

def handle(pkt):
```

```
    if pkt.haslayer(TCP) and pkt.haslayer(Raw) and pkt.haslayer(IP):
        payload = bytes(pkt[Raw].load)
        if len(payload) >= 2:  # esperamos [shift][cipher...]
            shift = payload[0]
            plain = dec_caesar(shift, payload[1:])
            print(f"{pkt[IP].src}:{pkt[TCP].sport} -> {pkt[IP].dst}:{pkt
                [TCP].dport} | "
                  f"len={len(payload)} | shift={shift} | plaintext='{
                      plain}'")

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Uso: sudo python3 sniff_caesar.py <iface> [puerto]")
        sys.exit(1)
    iface = sys.argv[1]
    port = sys.argv[2] if len(sys.argv) > 2 else "3333"
    bpf = f"tcp port {port}"
    print(f"Sniffeando en iface={iface} filtro='{bpf}' ... Ctrl-C para
        salir")
    sniff(iface=iface, filter=bpf, prn=handle, store=False)
```

(Código base del sniffer según el archivo del alumno. :contentReference[oaicite:1]index=1)

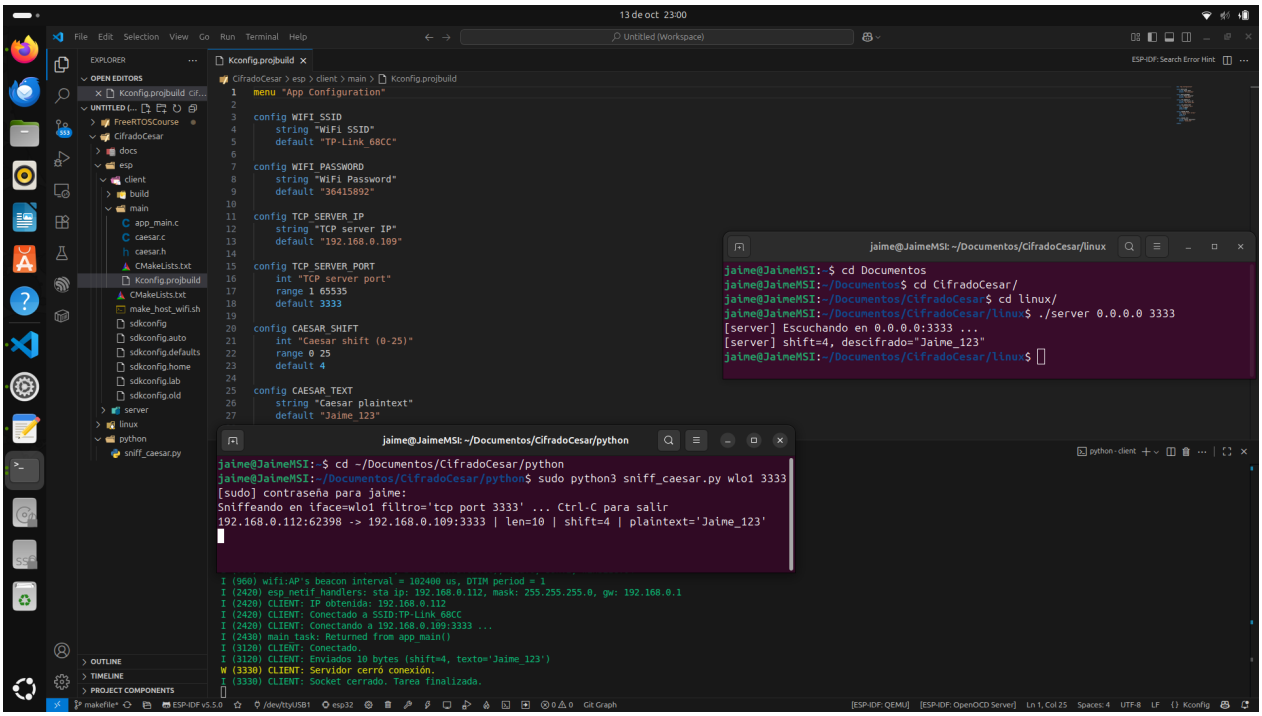


Figura 7: Sniffer Python mostrando `shift=4` y `plaintext='Jaime_123'`.

## 7. Resultados

- En ambos escenarios se observa en consola del receptor el texto descifrado correctamente.
- Las capturas en Wireshark validan el `3-way handshake`, el segmento de datos con `Len=9/10` y el cierre de conexión.
- El sniffer externo (Scapy) reconstruye el *plaintext* a partir del payload de aplicación, confirmando el formato del mensaje.

## 8. Conclusiones

Se implementó y probó con éxito un protocolo de *eco-descifrado* usando César sobre TCP. Las pruebas cruzadas (ESP↔Linux), más la verificación con Wireshark y Scapy, demuestran:

1. Correcta operación de la pila TCP/IP en ambos extremos.
2. Formato simple y verificable en la capa de aplicación.
3. Integración fluida entre herramientas embebidas (ESP-IDF) y utilerías de escritorio.