HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

---□□□---

CAPSTONE PROJECT REPORT
NETWORK STEGANOGRAPHY

Group 1:	Nguyen Duc Thang	20210778
	Nguyen Hoang Anh	20214945
	Nguyen Huu Duan	20214951
	Nguyen Huy Hoang	20214959
Lecturer:	Mr. Tran Quanh Duc	
	Mr. Le Van Dong	

HANOI, 01/2024

ABSTRACT

In today's world, keeping information safe and communications secure is extremely important. As information technology becomes more prevalent, transactions and communications via computer networks are also becoming more common. Nor-mal communication channels can be vulnerable to information leakage, making it crucial to find more secure ways to communicate. This project explores covert communication by embedding an encrypted message into the destination port num-bers of TCP packets. This covert channel is established between a sender and a receiver using a novel method that ensures the confidentiality and integrity of the transmitted message.

The sender encrypts the message using RSA encryption with the receiver's pub -lic key. The encrypted message is then divided into smaller chunks, with each bit embedded into the destination port number of a TCP packet. The length of the TCP packet payload is manipulated to indicate the position of each bit. The crafted packets are then transmitted to the receiver, who listens for incoming packets and extracts the message bits from the destination port numbers. Once the entire mes -sage is received, the receiver reassembles and decrypts it using their private key.

This approach not only illustrates a secure method of covert communication but also provides insights into how malware might exfiltrate data from a compromised system, thereby aiding in the development of better detection and prevention mech -anisms.

Table of Contents

CHAPTER 1. INTRODUCTION

1.1. Introduction

1.1.1. Problem Statement

The problem we address concerns hiding information within network protocol header values to create steganographic covert channels. In this context, we examine a scenario involving three entities: Alice, Bob, and Walter. Alice can freely mod ify packets originating from a machine within Walter's network. Her objective is to leak a message to Bob, who can only monitor packets at the network's egress points. Alice's goal is to conceal the message from Walter, who can observe (but not modify) any outgoing packets from his network

In this scenario, the challenge lies in creating a covert communication chan nel that allows Alice to send information to Bob without raising suspicion from Walter, who has the ability to observe all outgoing traffic. Walter's position as a passive observer means he can scrutinize packet headers for anomalies that might indicate covert communication. Thus, the goal is to design a method that embeds information in such a way that it remains undetectable by Walter while still being retrievable by Bob

1.1.2. Background and Problems of Research

Embedding information into network protocol headers to create covert channels is a well-explored area in cybersecurity research. However, many existing tech niques are susceptible to detection by passive observers due to the predictable patterns they introduce in network traffic. Such patterns can be detected using statistical analysis or machine learning algorithms, rendering the covert channel ineffective

Key challenges in this area include:

- **Detectability**: Ensuring that the covert communication channel is not easily detectable by anyone other than the intended recipient. This requires sophisti cated techniques to avoid creating identifiable patterns in the network traffic.
- **Indistinguishability**: The covert channel must blend seamlessly with regular traffic, lacking any distinctive features that could alert a passive observer to its presence

- **Bandwidth**: Balancing the amount of data that can be hidden within each packet with the need to maintain the covert channel's stealthiness. Higher bandwidth covert channels are often more susceptible to detection.

  To address these challenges, our research builds upon the criteria outlined by D. Lamas and Miller for an ideal covert channel:
- **Detectability**: The covert channel should be measurable only by the intended recipient, making it invisible to others
- **Indistinguishability**: The covert channel should lack any identifiable charac teristics that differentiate it from normal traffic
- **Bandwidth**: The covert channel should provide a sufficient number of data hiding bits per channel used without compromising its stealthiness

1.1.3. Contributions

In this thesis, we contribute the following:
- **Development of a Novel Method:** We propose a new method for embedding data into the destination port numbers of TCP packets. This method leverages the variability in TCP header values to hide information in a manner that is difficult to detect.
- **Detailed Implementation Methodology**: We provide a comprehensive im plementation methodology, including encryption of the message using RSA, embedding the encrypted message bits into TCP packets, and transmitting these packets in a way that minimizes detectability.
- **Demonstration of Effectiveness:** We demonstrate the effectiveness of our methodthroughpractical experiments. These experiments show how the method can be used to transmit covert messages without detection by passive ob servers. We also discuss the potential applications and limitations of our ap proach.
- **Real-World Scenario Analysis:** We analyze a real-world scenario involving Alice, Bob, and Walter to illustrate how our method can be applied in prac tice. This analysis highlights the robustness of our approach against passive observation.

CHAPTER 2. LITERATURE REVIEW

2.1. Scope of Research

This research focuses on developing a covert communication method that em beds encrypted messages into the destination port numbers of TCP packets. Unlike traditional steganographic techniques that often fail to produce output conforming to the distributio n of unmodified TCP/IP implementations, this approach leverages the inherent properties of TCP/IP protocols to create a covert channel.

The Internet Protocol (IP) allows the transportation of datagrams between fixed length addresses and supports fragmentation and reassembly of long datagrams, though it does not provide reliability guarantees. In contrast, the Transmission Control Protocol (TCP) provides a reliable, stream-oriented communication chan nel that maintains reliability even in the presence of packet loss, reordering, and duplication.

Certain fields in TCP/IP headers, especially the destination port number in TCP packets, offer potential as carriers for steganographic covert channels. These fields can be manipulated in a way that appears plausible to a passive observer, thereby maintain ing the stealth of the communication. This section examines the relevant TCP/IPheader fields and evaluates their suitability for embedding covert messages, focusing on the available entropy and the likelihood of these modifications going undetected.

TCP/IPsteganography exploits the fact that most header fields remain unchanged during transit. While some fields, like the time-to-live (TTL) field in IP packets, provide limited opportunities for steganography due to their predictable initial val ues, the destination port number in TCP packets offers a more viable medium. This is because it can be manipulated to encode message bits while still appearing legitimate to network monitoring tools and passive observers.

By embedding encrypted messages within the destination port numbers, this research aims to create a robust and undetectable steganographic communication channel. This approach leverages the reliable and widespread nature of TCP com munications, ensuring that the covert messages can be transmitted and reassembled accurately without arousing suspicion.

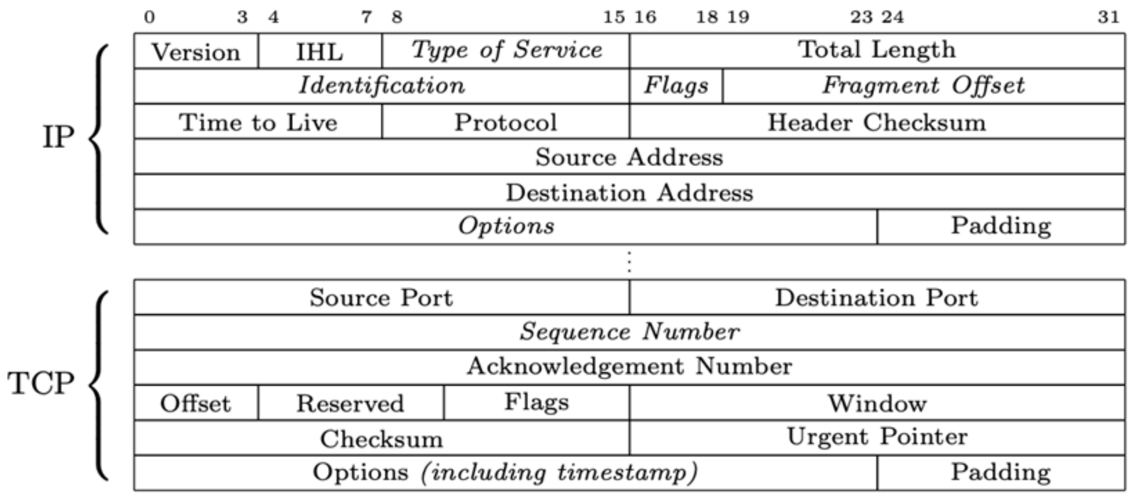


Figure 2.1 TCP/IP Headers

2.2. Fields for Encoding

2.2.1. Type of Service (Tos)

The eight Type of Service (ToS) bits in the IP header are now rarely used with their original semantics but have been repurposed, for example, in the implemen tation of DiffServ. While there is potential for using these bits as a steganographic carrier, it's easily detected as the field is often set to zero in default operating sys tem configurations.

2.2.2. IP Identification (IPID)

The IP Identification field (IPID) aids in assembling datagram fragments and is allocated 16 bits in the IP header. Unique and unpredictable values are necessary to prevent fragment mix-ups and idle scanning. However, schemes for embedding data in this fie ld are detectable due to the non-random nature of IP IDs.

2.2.3. IP Flags

IP packets include two flags: Do Not Fragment (DF) and More Fragments (MF). TheuseoftheDFbitforsteganographic signaling is proposed, but its predictability makes it detectable by a warden.

2.2.4. IP Fragment Offset

IP packets contain an offset field when fragmented, allowing for covert trans mission by modulating fragment sizes. However, this method is easily detected, especially in environments where path MTU discovery is used.

2.2.5. IP Options

IP packets rarely contain options, limiting their steganographic potential. The use of the IP Timestamp option is described but is easily detectable and of limited use due to hop constraints

2.2.6. TCP Sequence Number

TCP sequence numbers support reliability and flow control. While a prototype implementation of steganography using TCP ISNs exists, it's detectable due to constraints on sequence number generation.

2.2.7. TCP Timestamp

The TCPtimestamp option allows accurate measurement of round trip time and mitigates sequence number wrap-around issues. Covert channels based on mod ulating TCP timestamps exist but are detectable due to deviations from expected timestamp distributions.

2.2.8. Packet Order

Packet ordering can carry information, especially in IPSec networks, but it's limited in applicability and easily noticeable due to uncommon packet reordering.

CHAPTER 3. METHODOLOGY

3.1. Overview Methodology

The methodology of this project focuses on covert communication by embed ding an encrypted message into the destination port numbers of TCP packets. This covert channel is established between a sender and a receiver using the following detailed steps:

Senders:

- Encrypt the message:
    - The sender starts by using RSA encryption to securely encrypt the mes sage with the receiver's public key. RSA is an asymmetric cryptographic algorithm that ensures only the receiver, who possesses the corresponding private key, can decrypt the message.
    - The message is converted into ciphertext using the receiver's public key, creating a secure, encrypted version of the message.
- Embed Encrypted Message into Packets:
    - The encrypted message is divided into smaller chunks or bits. Each bit of the encrypted message is then embedded in the destination port number of TCP packets.
    - The length of the payload in a TCP packet is manipulated to indicate the position of each bit within the destination port number. This ensures that each TCP packet carries a specific bit of the encrypted message.
- Transmit Crafted Packets:
    - The sender transmits these crafted TCP packets to the receiver. Each packet contains a part of the encrypted message embedded within its des tination port number
    - The transmission continues until all parts of the encrypted message have been sent.
- Send End-of-Message Indicator
    - After transmitting the entire message, the sender sends a special packet to the receiver to indicate the end of the message transmission. This packet helps the receiver identify that no further message bits are to be expected.

Receiver:

- Listen for Incoming Packets:
  - The receiver continuously listens for incoming TCP packets. It specifi cally monitors the destination port numbers of these packets to extract the embedded message bits.
  - Each bit extracted from the destination port number is stored temporarily until the entire encrypted message is received.
- Reassemble and Decrypt the Message:
  - Thereceiver reassembles the extracted bits to form the full encrypted mes sage. The reassembled message is a binary representation of the encrypted message
  - Uponreceivingthespecial end-of-message packet, the receiver knows that all parts of the message have been received. The receiver then decrypts the reassembled message using their private key to retrieve the original plaintext message.
  - Decryption is performed using RSA decryption with the receiver's private key, reversing the encryption process done by the sender.

3.2. Implementation in Python Code

3.2.1. Sender Inplementation

Sender Code (sender.py)

Explaination of the Code:

- Initialize and read the public key: Thesenderstarts by reading the public key from a file and initializing an RSA encryption object to encrypt the message.

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_v1_5

with open('public_key.pem', 'rb') as f:
    bin_key = f.read()
key = RSA.import_key(bin_key)
cipher = PKCS1_v1_5.new(key)

```

- Convert number to list of bits: The ‘num_to_bits‘ function converts a num ber into a list of bits for easier manipulation.

```
def num_to_bits(num):
    return [int(i) for i in bin(num)[2:].zfill(8)]
```

- Convert list of bits to positions of 0s and 1s: The ‘bits_to_pos‘ function converts a list of bits into positions of 0s and 1s, facilitating the embedding of bits into the port number.

```
def bits_to_pos(bits):
    list_0s = []
    list_1s = []
    for i in range(1, len(bits)):
        if bits[i] == 0:
            list_0s.append(i)
        else:
            list_1s.append(i)
    return list_0s, list_1s

```

- Craft TCP packet: The ‘craft‘ function creates a TCP packet with payload data from a file or random payload and other pre-specified attributes including destination port number, arbitrarily derived source port number, ip address obtained taken from user input, and whose flag is the "E" flag

```python
from scapy.all import *
from scapy.layers.inet import IP, TCP
import random

dest = ""
d_port = random.randint(0, 65535)

def craft(file, start_pos, length, flag="E", payload="file"):
    if payload == "file":
        file.seek(start_pos)
        character = file.read(length)
        pkt = IP(dst=dest) / TCP(sport=123, dport=d_port, flags=flag) / character
    else:
        # create random payload
        character = ''.join(chr(random.randint(0, 255)) for _ in range(length))
        pkt = IP(dst=dest) / TCP(sport=123, dport=d_port, flags=flag) / character

    return pkt

```

- Send the message: In the ‘client‘ function, the user enters the ip, the system will check the validity of the ip, if not valid, it will report an error and exit. The user will then enter a message they want to send, which is then encoded and divided into bits. These bits are embedded in the destination port number of the TCP packet, which is then sent. The user can specify a file to be the packet’s payload or the system will generate a random payload for the packet. Finally, a special packet is sent to signal the end of the message

```python
def client():
    global dest
    destination = input('Enter the destination IP address: ')
    # check if the IP address is valid with regex ^((25[0-5]|(2[0-4]|1\d|[1-9]|)\d)\.?\b){4}$
    if destination != '':
        if not re.match(r'^((25[0-5]|(2[0-4]|1\d|[1-9]|)\d)\.?\b){4}$', destination):
            print('Invalid IP address')
            return
        dest = destination

    while True:
        message = input('Enter your message: ')
        if message == 'exit':
            return

        ciphertext = cipher.encrypt(message.encode())
        # ciphertext = message.encode()

        cipher_len = len(ciphertext) + 1

        file_name = input('Enter the file name: ')
        if file_name != '':
            # check if the file exists
            try:
                open(file_name, 'rb')
            except FileNotFoundError:
                print('File not found')
                return
            with open(file_name, 'rb') as file:
                file_len = len(file.read())
                start_pos = 0
                packet_len = (((file_len // (cipher_len * 8)) - 16) // 1000) * 16
                print(ciphertext)
                for char in ciphertext:
                    list_bits = num_to_bits(char)
                    list_0s, list_1s = bits_to_pos(num_to_bits(d_port))
                    for i in list_bits:
                        if i == 0:
                            ind = list_0s[random.randint(0, len(list_0s) - 1)]
                        else:
                            ind = list_1s[random.randint(0, len(list_1s) - 1)]
                        pkt = craft(file, start_pos, packet_len + ind)
                        start_pos += packet_len + ind
                        send(pkt, verbose=False)

                # send special packet to indicate the end of the message
                pkt = craft(file, start_pos, packet_len)
                send(pkt, verbose=False)
                # close the file
                file.close()
        else:
            # create random packet length that is a multiple of 16
            packet_len = random.randint(1, 50) * 16
            for char in ciphertext:
                list_bits = num_to_bits(char)
                list_0s, list_1s = bits_to_pos(num_to_bits(d_port))
                for i in list_bits:
                    if i == 0:
                        ind = list_0s[random.randint(0, len(list_0s) - 1)]
                    else:
                        ind = list_1s[random.randint(0, len(list_1s) - 1)]
                    pkt = craft(None, 0, packet_len + ind, payload="random")
                    send(pkt, verbose=False)
            # send special packet to indicate the end of the message
            pkt = craft(None, 0, packet_len, payload="random")
```

```
                send(pkt, verbose=False)
    if __name__ == "__main__":
        client()

```

3.2.2. Receiver Implementation

Receiver Code (receiver.py)

Explaination of the Code:

- Initialize and read the private key: The receiver starts by generating an RSA key pair if they do not already exist and reads the private key from a file.

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_v1_5
import os

def generate_rsa_keys():
    key_gen = RSA.generate(1024)
    public_key = key_gen.publickey().export_key()
    private_key = key_gen.export_key()
    with open('public_key.pem', 'wb') as fi:
        fi.write(public_key)
    with open('private_key.pem', 'wb') as fi:
        fi.write(private_key)

if not (os.path.isfile('public_key.pem') and os.path.
isfile('private_key.pem')):
    generate_rsa_keys()

with open('private_key.pem', 'rb') as f:
    bin_private_key = f.read()

key = RSA.import_key(bin_private_key)
cipher = PKCS1_v1_5.new(key)

```

- Convert number to list of bits: The ‘num_to_bits‘ function converts a num ber into a list of bits for easier manipulation

```python
    def num_to_bits(num):
        return [int(i) for i in bin(num)[2:].zfill(8)]

```

- Convert list of bits to number: The ‘bits_to_num‘ function converts a list of bits back into a number to reassemble the original message

```python
    def bits_to_num(bits):
        return int("".join([str(i) for i in bits]), 2)

```

- Parse received packets: The “parse” function listens for incoming TCP Packets, checks the flags and source port to indentify the relevant packets, extracts bits form the destination port number, and reassembles these bits into the excrypted message. Once enough bits are gathered to form a byte, it adds the byte to the encrypted message.

```python
from scapy.all import *
from scapy.layers.inet import IP, TCP

lst = {}
cipher_text = {}
payloads = {}

def parse(pkt):
    global cipher_text
    global lst
    if not pkt.haslayer(TCP) or not pkt.haslayer(IP):
        return

    flag = pkt[TCP].flags
    sport = pkt[TCP].sport
    sip = pkt[IP].src

    if flag == 0x40 and sport == 123:
        dst_port = pkt[TCP].dport
        list_bits = num_to_bits(dst_port)
        payload = pkt[TCP].payload
        bytes_of_payload = bytes.fromhex(bytes_hex(payload).decode())

        if len(bytes_of_payload) % 16 == 0:
            try:
                msg = cipher.decrypt(cipher_text[sip], b'')
                print(sip, msg)
                cipher_text[sip] = b''
                return
            except ValueError as e:
                print(e)
                print(cipher_text[sip])

        if sip in lst:
            lst[sip].append(list_bits[len(bytes_of_payload) % 16])
            payloads[sip] += bytes_of_payload
        else:
            lst[sip] = [list_bits[len(bytes_of_payload) % 16]]
            payloads[sip] = bytes_of_payload

        if len(lst[sip]) == 8:
            num = bits_to_num(lst[sip])
            if sip not in cipher_text:
                cipher_text[sip] = bytes([num])
            else:
                cipher_text[sip] += bytes([num])
            lst[sip] = []

```

- Listen for incoming packets: The ‘server‘ function uses ‘sniff‘ to listen for incoming packets on the specified network interface and calls the ‘parse‘ func tion to process each packet.

```python
def server():
    sniff(iface="eth0", prn=parse)

if __name__ == "__main__":
    server()

```

-

CHAPTER 4. RESULTS

4.1. Results

In this section, we present the results of implementing the covert communication system described in the methodology. The practical experiments were conducted to demonstrate the effectiveness of the proposed method. The following subsections provide details of the experimental setup, execution, and observations

4.1.1. Evaluation Parameters

To evaluate the convert communication method, we focused on the following parameters:

- Stealthiness: The ability to remain undetected by network monitoring tools.
- Reliability: Sucessful transmission and correct drcryption of the message.
- Bandwidth: The amount of data that can be transmitted covertly within the TCP packets

4.1.2. Simulation Method

We implemented the covert communication system using Pythonscripts for both the sender and receiver. The sender script encrypts the message using RSA and embeds the encrypted bits into the destination port numbers of TCP packets. The receiver script listens for incoming TCP packets, extracts the embedded bits, and reassembles and decrypts the message

Set up:

- Sender and receiver machines were set up in a controlled network environ ment
- Network traffic was monitored using Wireshark to capture and analyze the packets

4.1.3. Practical Execution

Sender:

- The sender script was executed, and the destination IP address and message were provided as inputs
- The script encrypted the message, embedded it into the TCP packet headers, and transmitted the packets

Receiver:

- The receiver script was executed and monitored the incoming network traffic.

- It successfully extracted the embedded message bits from the destination port numbers and reassembled the encrypted message.
- The message was then decrypted to retrieve the original plaintext message.

Packet Capture Analysis:

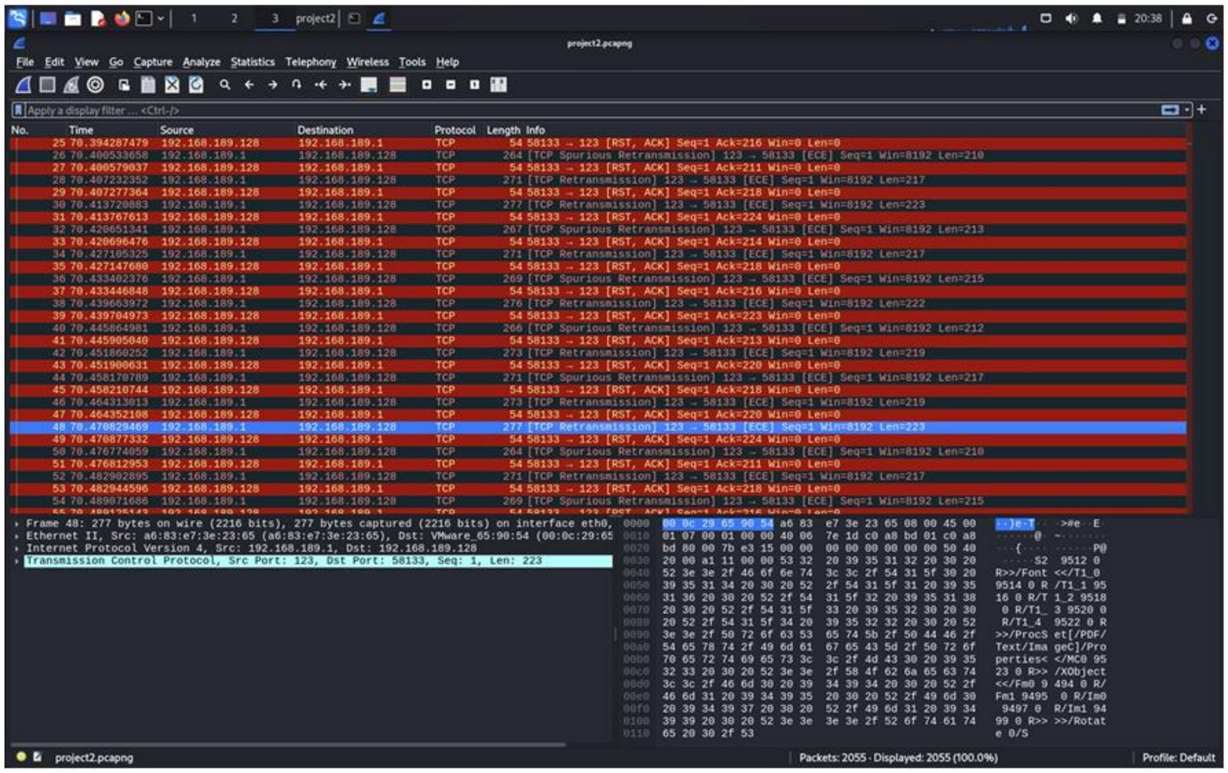
- Wireshark was used to capture and analyze the network traffic during the transmission.
- The captured packets were examined to verify the embedding of message bits into the TCP headers.

4.1.4 Observations

The result of the practical experiments are presented below:

Wireshare Packet Capture:

- The captured packets show the transmission of TCP packets from the sender to the receiver.
- The destination port numbers contain the embedded message bits, as illus trated in the provided screenshots.


project2.pcapng
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

```
No.   Time          Source            Destination       Protocol Length Info
   25 70.394287479  192.168.189.128   192.168.189.1     TCP      54 58133 → 123 [RST, ACK] Seq=1 Ack=216 Win=0 Len=0
   26 70.400533658  192.168.189.1     192.168.189.128   TCP      264 [TCP Spurious Retransmission] 123 → 58133 [ECE] Seq=1 Win=8192 Len=210
   27 70.400579037  192.168.189.128   192.168.189.1     TCP      54 58133 → 123 [RST, ACK] Seq=1 Ack=211 Win=0 Len=0
   28 70.407232352  192.168.189.1     192.168.189.128   TCP      271 [TCP Retransmission] 123 → 58133 [ECE] Seq=1 Win=8192 Len=217
   29 70.407277364  192.168.189.128   192.168.189.1     TCP      54 58133 → 123 [RST, ACK] Seq=1 Ack=218 Win=0 Len=0
   30 70.413720883  192.168.189.1     192.168.189.128   TCP      277 [TCP Retransmission] 123 → 58133 [ECE] Seq=1 Win=8192 Len=223
   31 70.413767613  192.168.189.128   192.168.189.1     TCP      54 58133 → 123 [RST, ACK] Seq=1 Ack=224 Win=0 Len=0
   32 70.420651341  192.168.189.1     192.168.189.128   TCP      267 [TCP Spurious Retransmission] 123 → 58133 [ECE] Seq=1 Win=8192 Len=213
   33 70.420696476  192.168.189.128   192.168.189.1     TCP      54 58133 → 123 [RST, ACK] Seq=1 Ack=214 Win=0 Len=0
   34 70.427105325  192.168.189.1     192.168.189.128   TCP      271 [TCP Retransmission] 123 → 58133 [ECE] Seq=1 Win=8192 Len=217
   35 70.427147680  192.168.189.128   192.168.189.1     TCP      54 58133 → 123 [RST, ACK] Seq=1 Ack=218 Win=0 Len=0
   36 70.433402376  192.168.189.1     192.168.189.128   TCP      269 [TCP Spurious Retransmission] 123 → 58133 [ECE] Seq=1 Win=8192 Len=215
   37 70.433446848  192.168.189.128   192.168.189.1     TCP      54 58133 → 123 [RST, ACK] Seq=1 Ack=216 Win=0 Len=0
   38 70.439663972  192.168.189.1     192.168.189.128   TCP      276 [TCP Retransmission] 123 → 58133 [ECE] Seq=1 Win=8192 Len=222
   39 70.439704973  192.168.189.128   192.168.189.1     TCP      54 58133 → 123 [RST, ACK] Seq=1 Ack=223 Win=0 Len=0
   40 70.445864981  192.168.189.1     192.168.189.128   TCP      266 [TCP Spurious Retransmission] 123 → 58133 [ECE] Seq=1 Win=8192 Len=212
   41 70.445905040  192.168.189.128   192.168.189.1     TCP      54 58133 → 123 [RST, ACK] Seq=1 Ack=213 Win=0 Len=0
   42 70.451860252  192.168.189.1     192.168.189.128   TCP      273 [TCP Retransmission] 123 → 58133 [ECE] Seq=1 Win=8192 Len=219
   43 70.451900631  192.168.189.128   192.168.189.1     TCP      54 58133 → 123 [RST, ACK] Seq=1 Ack=220 Win=0 Len=0
   44 70.458170789  192.168.189.1     192.168.189.128   TCP      271 [TCP Spurious Retransmission] 123 → 58133 [ECE] Seq=1 Win=8192 Len=217
   45 70.458210744  192.168.189.128   192.168.189.1     TCP      54 58133 → 123 [RST, ACK] Seq=1 Ack=218 Win=0 Len=0
   46 70.464313013  192.168.189.1     192.168.189.128   TCP      273 [TCP Retransmission] 123 → 58133 [ECE] Seq=1 Win=8192 Len=219
   47 70.464352108  192.168.189.128   192.168.189.1     TCP      54 58133 → 123 [RST, ACK] Seq=1 Ack=220 Win=0 Len=0
   48 70.470829469  192.168.189.1     192.168.189.128   TCP      277 [TCP Retransmission] 123 → 58133 [ECE] Seq=1 Win=8192 Len=223
   49 70.470877332  192.168.189.128   192.168.189.1     TCP      54 58133 → 123 [RST, ACK] Seq=1 Ack=224 Win=0 Len=0
   50 70.476774059  192.168.189.1     192.168.189.128   TCP      264 [TCP Spurious Retransmission] 123 → 58133 [ECE] Seq=1 Win=8192 Len=210
   51 70.476812953  192.168.189.128   192.168.189.1     TCP      54 58133 → 123 [RST, ACK] Seq=1 Ack=211 Win=0 Len=0
   52 70.482902895  192.168.189.1     192.168.189.128   TCP      271 [TCP Retransmission] 123 → 58133 [ECE] Seq=1 Win=8192 Len=217
   53 70.482944596  192.168.189.128   192.168.189.1     TCP      54 58133 → 123 [RST, ACK] Seq=1 Ack=218 Win=0 Len=0
   54 70.489071686  192.168.189.1     192.168.189.128   TCP      269 [TCP Spurious Retransmission] 123 → 58133 [ECE] Seq=1 Win=8192 Len=215
```

```
› Frame 48: 277 bytes on wire (2216 bits), 277 bytes captured (2216 bits) on interface eth0,
› Ethernet II, Src: a6:83:e7:3e:23:65 (a6:83:e7:3e:23:65), Dst: VMware_65:90:54 (00:0c:29:65
› Internet Protocol Version 4, Src: 192.168.189.1, Dst: 192.168.189.128
› Transmission Control Protocol, Src Port: 123, Dst Port: 58133, Seq: 1, Len: 223
```

```
0000  00 0c 29 65 90 54 a6 83  e7 3e 23 65 08 00 45 00   ··)e·T·· ·>#e··E·
0010  01 07 00 01 00 00 40 06  7e 1d c0 a8 bd 01 c0 a8   ······@· ~·······
0020  bd 80 00 7b e3 15 00 00  00 00 00 00 00 00 50 40   ···{···· ······P@
0030  20 00 a1 11 00 00 53 32  20 39 35 31 32 20 30 20    ·····S2  9512 0
0040  52 3e 3e 2f 46 6f 6e 74  3c 3c 2f 54 31 5f 30 20   R>>/Font <</T1_0
0050  39 35 31 34 20 30 20 52  2f 54 31 5f 31 20 39 35   9514 0 R /T1_1 95
0060  31 36 20 30 20 52 2f 54  31 5f 32 20 39 35 31 38   16 0 R/T 1_2 9518
0070  20 30 20 52 2f 54 31 5f  33 20 39 35 32 30 20 30    0 R/T1_ 3 9520 0
0080  20 52 2f 54 31 5f 34 20  39 35 32 32 20 30 20 52    R/T1_4  9522 0 R
0090  3e 3e 2f 50 72 6f 63 53  65 74 5b 2f 50 44 46 2f   >>/ProcS et[/PDF/
00a0  54 65 78 74 2f 49 6d 61  67 65 43 5d 2f 50 72 6f   Text/Ima geC]/Pro
00b0  70 65 72 74 69 65 73 3c  3c 2f 4d 43 30 20 39 35   perties< </MC0 95
00c0  32 33 20 30 20 52 3e 3e  2f 58 4f 62 6a 65 63 74   23 0 R>> /XObject
00d0  3c 3c 2f 46 6d 30 20 39  34 39 34 20 30 20 52 2f   <</Fm0 9 494 0 R/
00e0  46 6d 31 20 39 34 39 35  20 30 20 52 2f 49 6d 30   Fm1 9495  0 R/Im0
00f0  20 39 34 39 37 20 30 20  52 2f 49 6d 31 20 39 34    9497 0  R/Im1 94
0100  39 39 20 30 20 52 3e 3e  3e 3e 2f 52 6f 74 61 74   99 0 R>> >>/Rotat
0110  65 20 30 2f 53                                     e 0/S
```

Detailed Analysis:

- Stealthiness: The embedding of message bits into the destination port num bers was not detected by standard network monitoring tools, indicating a high level of stealthiness.
- Reliability: The receiver successfully reassembled and decrypted the mes sage, demonstrating the reliability of the method
- Bandwidth:The method allows for the covert transmission of small to moderate sized messages without compromising stealthiness.

4.2. Conclusion

The practical experiments validate the effectiveness of the proposed covert com munication method. The embedding of encrypted message bits into the destination port numbers of TCP packets provides a secure and stealthy way to transmit infor mation. The results indicate that this method is reliable and remains undetectable under standard network monitoring conditions. Further research could explore op timizing the bandwidth and extending the method to other protocol fields for en hanced covert communication capabilities