CENG519 - Phase 2 Report

Cover Channel Implementation

Student Name: [**Nevzat Altay Yücetaş**]
Student ID: [**2540367**]

Department of Computer Engineering
Middle East Technical University

April 13, 2025

Contents

1 Objective **2**

2 System Components **3**
2.1 Sender (`sender.py`) . . . 3
2.2 Receiver (`receiver.py`) . . . 3

3 Implementation Notes **4**
3.1 Use of TCP Option Field . . . 4
3.2 One-Way Covert Channel Design . . . 4
3.3 XOR-Based Message Encryption . . . 4
3.4 Timing and Logging Strategy . . . 5
3.5 Parameterization and Reusability . . . 5

4 Experiment Setup **5**
4.1 Test Environment . . . 5
4.2 Message Variants . . . 5
4.3 Repetition and Timing . . . 5
4.4 Log and Output Files . . . 6

5 Results **6**

6 Observations **6**

7 Conclusion **7**

1 Objective

The objective of this phase is to design, implement, and evaluate a one-way covert communication channel using TCP header options in a controlled container-based network environment. The primary goal is to encode a secret message into TCP packets in a way that is difficult to detect by third-party observers, without modifying the payload or standard header fields.

In the implemented setup, the sender process runs inside the `sec` container and covertly transmits data to a receiver running in the `insec` container. The covert channel is realized by embedding characters of a plaintext message into a rarely used and unrecognized TCP option field (specifically, option kind 76). Before transmission, the message is encrypted using a lightweight XOR cipher, which operates character by character against a repeated symmetric key. This encrypted message is then sent one character at a time using separate TCP SYN packets, each carrying a single encrypted byte encoded within the TCP option field.

On the receiver side, TCP packets arriving on a known port are sniffed using Scapy. The receiver parses the TCP header options, extracts the encrypted content, and applies the same XOR key to reconstruct the original plaintext message. Once the complete message is received, the receiver sends back a fixed acknowledgment using a standard UDP packet to a predefined port on the sender. This acknowledgment is not part of the covert channel itself but is used solely for measuring performance metrics such as round-trip time (RTT).

The project focuses on analyzing the performance characteristics of this covert channel by experimenting with different message lengths. For each transmission, the sender records the time taken to receive a UDP response after dispatching the full message. This RTT is logged and later processed to extract statistical measures including average latency, standard deviation, 95% confidence intervals, and an estimation of the covert channel capacity (in bits per second).

All components were implemented in Python using the Scapy library for low-level packet crafting and inspection, alongside native socket APIs for standard UDP communication. The structure of the implementation is modular, with configurable parameters such as message content, encryption key, and communication ports, allowing for reproducible experiments under varying conditions.

In summary, this phase successfully demonstrates a functioning covert communication mechanism based on TCP option manipulation. The system provides both a working prototype and quantifiable performance metrics, showing how such a covert channel can operate within an existing network setup without relying on conventional data payloads or visible application-layer transmissions.

2 System Components

The implementation is composed of two core Python modules: the sender and the receiver. These scripts establish and coordinate a one-way covert channel over TCP by embedding data in TCP option fields, and provide performance metrics via UDP-based RTT measurements.

2.1 Sender (`sender.py`)

The sender script is responsible for constructing TCP SYN packets that carry covert data in the TCP option field (kind 76), dispatching them one-by-one, and measuring the round-trip time based on a UDP acknowledgment from the receiver.

Its functionality includes:

- Fetching the receiver's IP address from the environment variable `INSECURENET_HOST_IP`.
- Encrypting the original plaintext message using a simple XOR cipher with a user-defined symmetric key.
- Sending a TCP packet with a header field indicating the message length (e.g., `LEN:12`) using TCP option kind 76.
- Iteratively sending each XOR-encrypted character in a separate TCP SYN packet using the same TCP option field.
- Listening for a UDP acknowledgment from the receiver on a predefined port (`udp_port = 9999` by default).
- Measuring and logging the round-trip time (RTT) for each complete message exchange, starting from the first TCP packet sent and ending at the reception of the UDP response.
- Saving all RTT measurements to a text file (`rtt_results.txt`) for later statistical analysis.

The sender logic is encapsulated in a parameterized function named `tcp_sender()`, which is invoked with the message content, XOR key, TCP destination port, and UDP reply port. These values can be modified in the main block of the script for repeated testing under different conditions.

2.2 Receiver (`receiver.py`)

The receiver script listens for incoming TCP packets on a predefined port and extracts the covert data from TCP option fields. Once the full message is reconstructed and decrypted, it sends back a UDP response to the sender.

Its workflow consists of:

- Sniffing TCP packets on port 8888 using Scapy's `sniff()` function with a custom packet handler.
- Parsing the TCP option list to identify entries with kind 76, and extracting their values as raw bytes.
- Detecting the message length from a specially formatted header packet (e.g., `LEN:9`).
- Collecting encrypted characters until the total expected message length is reached.
- Decrypting the collected characters using XOR with the same key used by the sender.

- Displaying the reconstructed plaintext message on the console.
- Sending a fixed UDP reply (e.g., `Ack!`) to the source IP and a predefined UDP port to signal the end of reception.
- Resetting internal state to prepare for the next incoming message.

The receiver behavior is encapsulated in the function `start_udp_listener()`, which accepts the XOR key, UDP reply message, and TCP/UDP port numbers as parameters. The script is designed to run indefinitely, listening and responding to new messages as they arrive.

3 Implementation Notes

This section outlines several important technical decisions and practical considerations encountered during the implementation of the covert channel system.

3.1 Use of TCP Option Field

The covert message is embedded in TCP option kind 76, an unused and undefined option code according to the official IANA registry. This choice was intentional to avoid interference with standard TCP behavior and to ensure that the data does not appear in any conventional header or payload field. TCP options are generally ignored by end systems and middleboxes if the option kind is unrecognized, which makes them ideal for stealthy data embedding.

Each packet sent by the sender contains a single encrypted character in the option value field. The structure of TCP allows multiple options in a single packet, but the system uses a fixed format for simplicity and clarity in parsing. Both the sender and receiver expect and handle only option kind 76.

3.2 One-Way Covert Channel Design

This system implements a unidirectional covert channel from sender to receiver. All covert information (i.e., the encrypted message) is transmitted from the sender to the receiver using TCP packets. No covert data is ever sent from the receiver back to the sender.

Instead, the receiver replies using a standard UDP packet containing a fixed acknowledgment message. This UDP reply is not part of the covert channel; it serves a purely operational role in performance measurement, allowing the sender to calculate round-trip time (RTT) for each message transmission. The use of UDP ensures that the response is lightweight, decoupled from the TCP stream, and easy to detect at the sender.

3.3 XOR-Based Message Encryption

The encryption algorithm used is a simple character-wise XOR operation between each byte of the message and a repeating symmetric key. This method was selected for its ease of implementation and to provide a basic level of obfuscation without introducing cryptographic complexity. The same key must be configured on both the sender and the receiver, and the key is hardcoded in both scripts to maintain consistency.

While XOR encryption does not provide security against advanced adversaries, it serves its purpose in this context by making the payload non-trivial to interpret by passive observers who may inspect TCP options.

3.4 Timing and Logging Strategy

Round-trip time is measured by the sender from the moment the first packet is dispatched (header packet) until the corresponding UDP acknowledgment is received. These measurements are recorded to a text file named `rtt_results.txt`, which is then processed by a separate analysis script.

The timing logic is implemented directly in the sender's main loop, and the results are exported with high precision for statistical processing, including confidence interval calculation and throughput estimation.

3.5 Parameterization and Reusability

Both the sender and receiver functions are implemented in a parameterized and reusable way. Instead of relying on command-line arguments, key parameters such as the message, encryption key, TCP destination port, and UDP port are passed directly to the main function within the script. This makes it easier to repeatedly test the same code under different configurations without changing the internal logic.

This design also simplifies experimentation, enabling rapid switching between test messages and configurations in order to gather performance data across a variety of scenarios.

4 Experiment Setup

To evaluate the performance of the covert channel implementation, a series of controlled experiments were conducted. The primary goal was to observe how message length impacts round-trip time (RTT) and the effective data capacity of the covert channel.

4.1 Test Environment

All experiments were carried out within the container-based development environment provided during Phase 1. The sender script was executed inside the `sec` container, and the receiver ran in the `insec` container. These containers were connected through a simulated middlebox system, which mirrors a real-world networking scenario with the ability to observe and potentially interfere with packet-level traffic.

4.2 Message Variants

Three distinct messages were used in the experiments to assess the behavior of the system under different message lengths:

- **Short message:** `Hi!`
- **Medium message:** `Hello covert!`
- **Long message:** `Long covert message is sent!`

Each of these messages was XOR-encrypted and transmitted using the sender logic described in Section 2. The same symmetric key (`altay`) was used in all cases.

4.3 Repetition and Timing

For each message, the sender script was executed with the corresponding message hardcoded in the `msg` variable. The `tcp_sender()` function was configured to transmit the full message 30 times consecutively, each time recording the RTT.

The RTT was measured as the elapsed time between the initial transmission of the header packet (indicating message length) and the reception of the UDP acknowledgment from the receiver. Each RTT value was stored in a file named `rtt_results.txt` in plain text format.

4.4 Log and Output Files

After completing each test, the contents of `rtt_results.txt` were archived under a different name corresponding to the message size:

- `rtt_results_small.txt`
- `rtt_results_medium.txt`
- `rtt_results_long.txt`

Each RTT log was then processed using the `analyze_results.py` script, which produced a corresponding output summary file:

- `analysis_output_small.txt`
- `analysis_output_medium.txt`
- `analysis_output_long.txt`

These files include statistical metrics such as average RTT, standard deviation, 95% confidence intervals, and the calculated channel capacity in bits per second.

5 Results

After running the experiment for each of the three message sizes, the recorded RTT data was analyzed to extract statistical metrics. The results include average round-trip time, 95% confidence intervals computed using the t-distribution, and the estimated covert channel capacity in bits per second.

Table 1 summarizes the outcome of each test run.

Message	**Average RTT (ms)**	**95% CI (ms)**	**Capacity (bps)**
`Hi!` (3 bytes)	625.98	[600.93, 651.03]	38.34
`Hello covert!` (13 bytes)	2276.03	[2217.93, 2334.13]	45.69
`Long covert message is sent!` (28 bytes)	4602.24	[4507.34, 4697.14]	48.67

Table 1: Summary of RTT measurements and channel capacity for different message sizes.

As seen in the results, the average RTT increases with the message length. This is expected because each character is transmitted as a separate TCP packet, introducing more processing and scheduling time. Despite this increase, the estimated capacity (in bits per second) also slightly increases, as the fixed overhead is amortized over longer messages.

6 Observations

The results demonstrate a consistent increase in round-trip time (RTT) as the length of the covert message increases. Since each character is transmitted in a separate TCP packet, longer messages naturally result in more transmissions and longer overall durations.

Despite higher RTTs, the estimated channel capacity (in bits per second) slightly improves with message length. This is due to the fixed overhead being spread across more bits, making the transmission more efficient as message size increases.

The 95% confidence intervals remained narrow, indicating stable and repeatable performance within the controlled environment. Additionally, the use of an unrecognized TCP option (kind 76) enabled effective covert transmission without affecting standard protocol behavior, supporting the stealthiness of the method.

However, the design remains unidirectional: covert data is sent only from sender to receiver. The reverse path uses a standard UDP response, which simplifies RTT measurement but limits covert interaction to one direction.

7 Conclusion

This phase demonstrated the successful implementation of a unidirectional covert channel using TCP option fields. By embedding XOR-encrypted data into TCP option kind 76, the sender was able to transmit covert messages that were accurately received and decrypted by the receiver.

UDP acknowledgments were used to measure round-trip time (RTT), allowing for performance analysis. Results showed that longer messages lead to higher RTT but also slightly better capacity in bits per second, due to improved efficiency over time.

The system proved stable and effective within the provided container environment. While limited to one-way covert transmission, the implementation serves as a solid foundation for more advanced covert communication techniques in future work.