Middle East Technical University
Department of Computer Engineering
Wireless Systems, Networks and Cybersecurity (WINS) Laboratory

Term Project Report - Phase 2

CENG519 Network Security
2024-2025 Spring
Term Project Report - Phase 2

Prepared by
Yılmaz Yiğitcan Uçan
Student ID: e2310555
yigitcan.ucan@metu.edu.tr
Computer Engineering
13 April 2025

1 Setup

1.1 Build

Run the following command in /code/sec and /code/insec directories to build go code.

```
GOOS=linux GOARCH=arm64 go build
```

1.2 Configuration

Edit configure-sec.sh and configure-insec.sh files as you want. The type parameter should be same for both of them.

1.2.1 Sender parameters:

- **Covert channel type:** cname/typed

```
# configure-sec.sh
./sender typed message.txt 10
```

1.2.2 Receiver parameters:

- **Covert channel type:** cname/typed
- **Filename:** message.txt or small.txt
- **Wait Between:** Integer (Delay between DNS requests in ms)

```
# configure-insec.sh
./receiver typed
```

1.3 GitHub Fork

I have forked the project in: github.com/ucanyiit/middlebox

The related phase 2 code is under this PR: Phase 2 PR

2 Covert Channels

There are two covert channel implementations presented in this paper. Both of the implementations follow the general flow below.

2.0.1 Sender (Sec node)

- **Message Preparation:** Takes a message (read from a file).
- **Query Generation:** Message is encoded and DNS queries are generated using the implemented function. This part depends on the given covert channel parameter and implementation.

- **Transmission:** The sender sends these DNS queries one by one (with an optional delay) over UDP to the receiver's IP address on port 53 (the standard DNS port).
- **End Signal:** After sending all data chunks, the sender sends a final query with a special marker. (The marker is different in the implementations)

2.0.2 Receiver (Insec Node)

- **Listening:** A DNS server listens on UDP port 53 for incoming requests.
- **Request Handling:** When a DNS request arrives, it's handled depending on the implementation. Here the received encoded data is saved to a map.
- **Data Extraction:**
  - Encoded Data: The receiver extracts data using the implemented method and saves it to reassemble the message using the parts.
  - **End Signal Check:** If the received query is an end signal, this means receiver can reassemble the message fully with the received DNS queries.
- **Reassembly:**
  - Condition Check: It checks if the `"end"` signal has been received and all data chunks are received.
  - Message Reconstruction: If both conditions are true, it triggers the function to reassamble the message (in a separate goroutine).
    - Concatenation: This function iterates over all received data, retrieves the corresponding chunks, and concatenates them.
    - Output: The final decoded message is printed.
    - Cleanup: The storage `(receivedChunks)` is then cleared.

2.1 Using CNAME DNS Queries

This code implements a covert channel that hides data within the domain names being requested in standard DNS queries. Instead of using DNS to legitimately resolve an IP address, the sender crafts special domain names that contain encoded chunks of the secret message, and the receiver extracts this data from the incoming queries.

2.1.1 Sender Side (sec node)

- **Chunking:** Splits the message into smaller pieces. The size of these chunks is limited by the maximum length of a DNS label (part of a domain name between dots).
- **Encoding:** Each chunk is encoded into hexadecimal format (e.g., "Hello" becomes "48656c6c6f"). This makes the data safe to use in domain names.
- **Domain Name Construction:** For each encoded chunk, the sender constructs a unique domain name. Based on the handleCNAMEDNSQuestion function, the format is: `[hex_encoded_chunk].[sequence_number].[BASE_DOMAIN]`
  - `hex_encoded_chunk`: The hex representation of the data piece.
  - `sequence_number`: A number (0, 1, 2...) indicating the order of this chunk in the original message.
  - `BASE_DOMAIN`: A common base domain (e.g., example.com) to make the queries look somewhat structured and potentially target a specific controlled server.

Example: 48656c6c6f.0.example.com (for the first chunk "Hello" with sequence number 0)

- **Query Generation:** The sender creates DNS queries. Crucially, the cname implementation uses the CNAME query type for these crafted domain names.
- **End Signal Marker:** end.`[total_number_of_chunks]`.`[BASE_DOMAIN]` Example: end.5.example.com (if there were 5 chunks in total, indexed 0 to 4).

2.1.2 Receiver Side (Insec node)

- **Request Handling:** When a DNS request arrives:
  - Query Type Check: It checks if the query type is CNAME. Other types are handled differently or ignored.
  - Domain Name Extraction: It extracts the requested domain name: *e.g.*, `48656c6c6f.0.example.com.`.
  - Domain Stripping: It strips the `BASE_DOMAIN` and the trailing dot.
  - Splitting: It splits the remaining part *e.g.*, `48656c6c6f.0` by the dot.
- **Data Extraction:**
  - Part Separation: It expects two parts: the hex chunk and the sequence number string.
  - Sequence Number Conversion: It converts the sequence number string to an integer.
  - **End Signal Check:** It checks if the first part is the special `"end"` signal.
    - If it's `"end"`, it records that the end signal was received and stores the total number of expected chunks `(lastSequenceNumber)`.
    - If it's not `"end"`, it assumes it's a hex-encoded data chunk.
- **Decoding & Storage:**
  - Decoding: The hex chunk is decoded back into its original binary form `(handleHexChunk)`.
  - Storage: The decoded chunk is stored in a map `(receivedChunks)` keyed by its `sequenceNumber (storeChunk)`. A mutex `(mapMutex)` protects this map.

2.2 Using DNS Query Types

This implementation encodes covert data into the sequence of DNS query *types* requested, rather than embedding data within the domain name itself. It utilizes a predefined mapping between small integer values (representing data bits) and specific DNS query types.

2.2.1 Sender Side (sec node)

- **Encoding**: The message is converted into a byte array. Each byte (8 bits) is processed in 2-bit chunks. The sender iterates through each byte, extracting the least significant 2 bits four times. Each 2-bit value (0, 1, 2, or 3) is mapped to a specific DNS query type using a predefined map (`DNS_TYPE_MAP`):
  - 0 → `layers.DNSTypeCNAME` (Standard code: 5)
  - 1 → `layers.DNSTypeA` (Standard code: 1)
  - 2 → `layers.DNSTypeAAAA` (Standard code: 28)
  - 3 → `layers.DNSTypeMX` (Standard code: 15)

This results in a sequence of DNS types representing the original message, where each original byte corresponds to four DNS queries.

- **Query Generation**: For each generated `DNSType` in the sequence, a DNS query packet is created (`generateDNSQuery`). Crucially, all these queries use the *same*, fixed base domain (`BASE_DOMAIN`, e.g., `example.com`). The data is solely encoded in the `QTYPE` field of the DNS query header. The sequence of the data is maintained by the order in which the queries are sent.
- **Transmission**: Queries are sent sequentially over UDP to the receiver's port 53, with a delay (`waitBetween`) between packets. The delay is added so that sequence would not be mixed.
- **End Signal**: After all data-carrying queries are sent, a final query is sent using a distinct, reserved query type: `layers.DNSTypeMD` (Standard code: 99). This signals the end of the transmission to the receiver.

2.2.2 Receiver Side (Insec node)

- **Request Handling**: When a request arrives, the handler focuses on the query type (`q.Qtype`). The requested domain name is ignored in this scheme.
- **Data Extraction & Decoding**: The handler (`handleTypedDNSQuestion`) processes each query based on its type:
  - **End Signal Check**: If `q.Qtype` matches the end signal type (`layers.DNSTypeMD`), the receiver notes that the transmission is complete (`handleEndSignal`). It records the total number of data-carrying queries received so far (`currentSequenceNumber`) as the expected final count (`lastSequenceNumber`).
  - **Data Decoding**: If `q.Qtype` is one of the data-carrying types (CNAME, A, AAAA, MX), the receiver performs the reverse mapping: it determines the 2-bit value (0-3) corresponding to the received type.
  - **Buffering & Byte Reconstruction**: The receiver needs to collect four consecutive 2-bit values (from four distinct incoming queries) to reconstruct one original byte of the message. As each query arrives, its corresponding 2-bit value is temporarily stored or processed. Once four 2-bit values are gathered, they are combined (e.g., byte $= (\text{bits}_1 \ll 6)|(\text{bits}_2 \ll 4)|(\text{bits}_3 \ll 2)|\text{bits}_4$) to form the original 8-bit byte.
- **Storage**: The reconstructed byte is stored in the `receivedChunks` map (`storeChunk`), keyed by its sequence number (representing the byte's position in the original message, e.g., 0 for the first byte, 1 for the second, etc.). Access to the map is protected by `mapMutex`. The sequence number for storage is derived from the overall count of received data queries (e.g., byte n is formed by queries $4n$ to $4n + 3$).
- **Reassembly**: Reassembly (`reassembleAndPrintMessage`) is triggered after the end signal (`DNSTypeMD`) has been received *and* all expected byte chunks (from 0 to `lastSequenceNumber`/4 - 1) are present in the `receivedChunks` map (`allSequencesReceived`). The function concatenates the stored bytes in the correct order and prints the final message. The state (`receivedChunks`, `endSignalReceived`, etc.) is then cleared.

3 Results

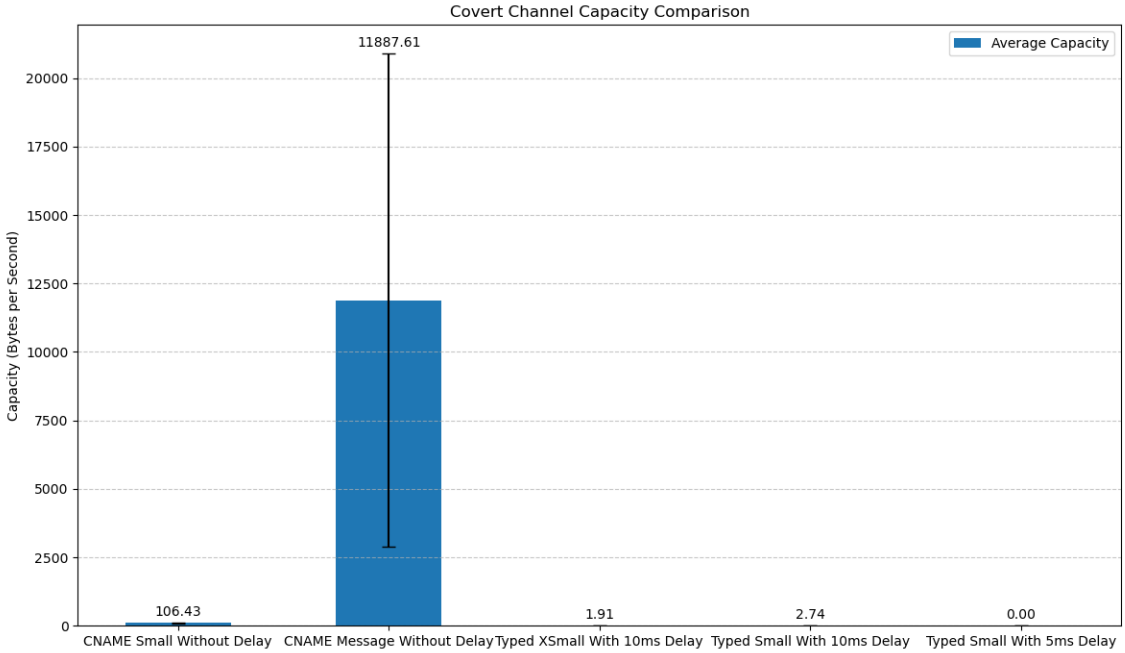

Figure 1 . Data Transfer Rate Comparison

This graph compares the data transfer rate (Bytes/Second) across different covert channel methods. The results clearly show that the CNAME method without delay achieves significantly higher capacity (around 12 kB/s) by embedding data in domain names, whereas the Typed method, encoding data via query types, has very low capacity < 3 B/s). When the delay is decreased to 5ms from 10ms, it fails to get the sequences numbers correctly and fails to transfer any data.

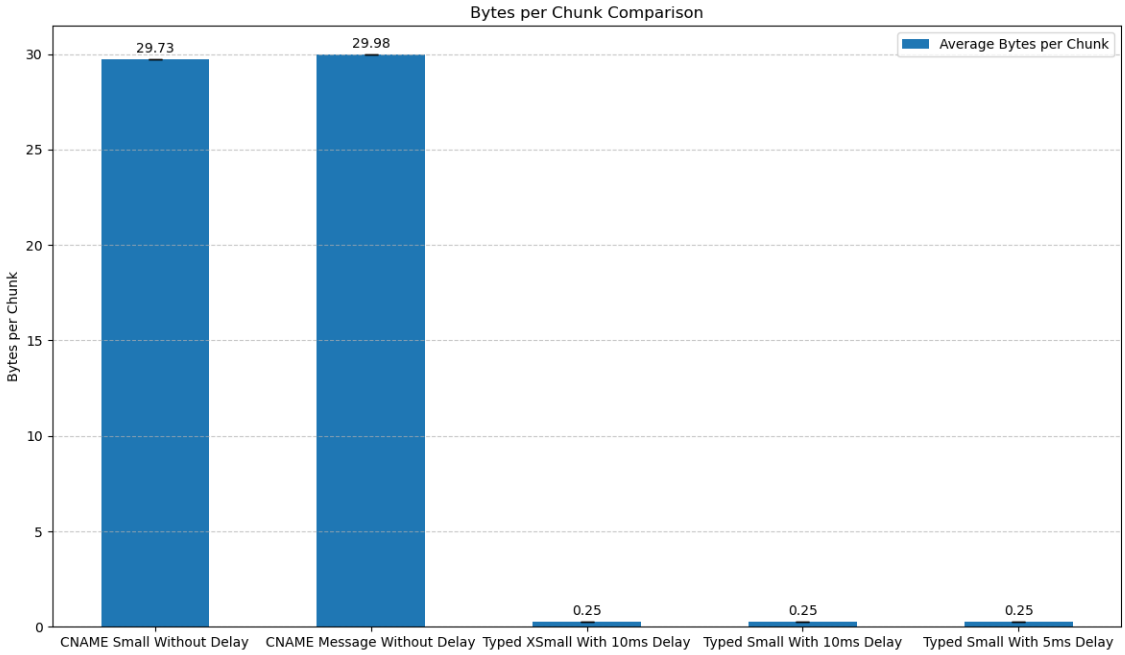

Figure 2 . Data Density per DNS Query

This chart illustrates the data density per DNS query for each method. The CNAME approach packs nearly 30 bytes into each query's domain name, while the Typed method encodes a minimal, fixed 0.25 bytes (2 bits) per query using only the query type, explaining the vast difference in overall throughput.

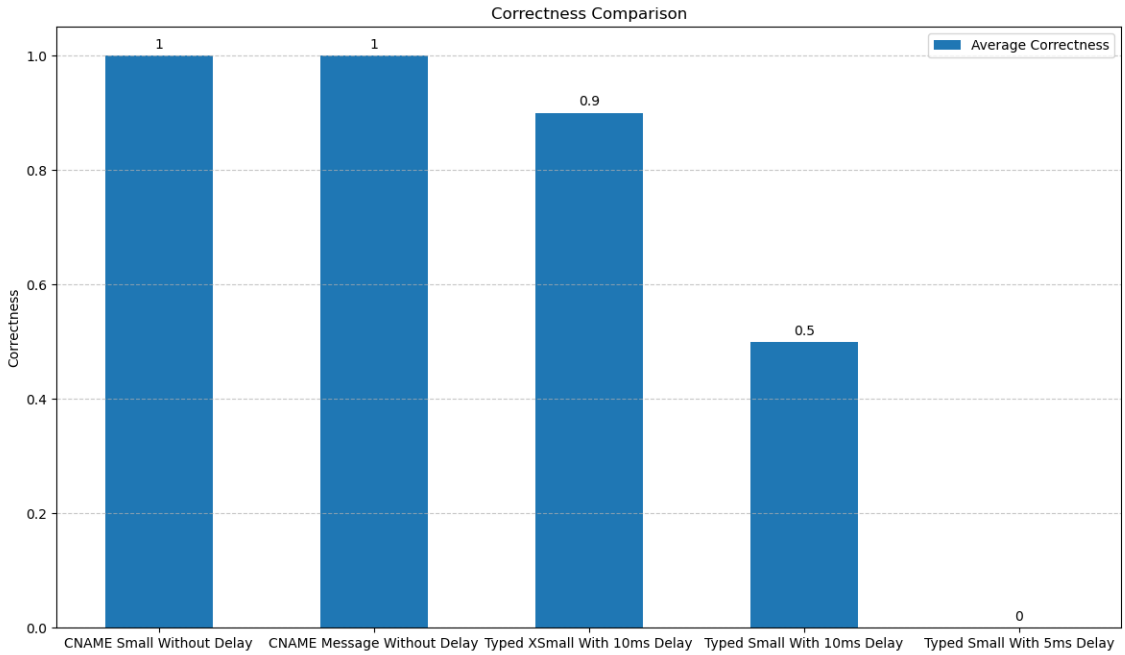

Figure 3 . Accuracy of Message Reconstruction

This chart measures the accuracy of message reconstruction at the receiver. The CNAME method demonstrated perfect reliability (1.0 correctness) in tests, while the Typed method's correctness was sensitive to timing, achieving high accuracy (0.9) only with a very small file and 10ms delay, degrading significantly (0.5) with a slightly larger file, and failing entirely (0.0) with a 5ms delay.

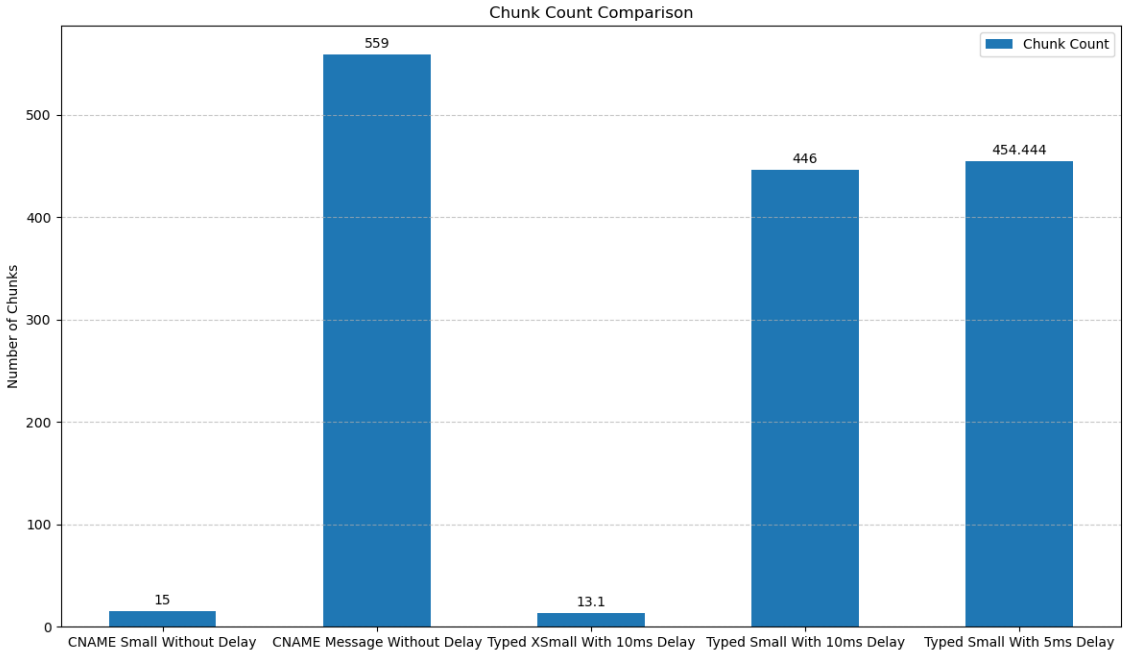

Figure 4 . Total Number of DNS Queries

This graph shows the total number of DNS queries (chunks) needed to transmit the message. Due to its higher data density, the CNAME method requires far fewer chunks (e.g., 15 for "Small") compared to the Typed method (e.g., 450 for "Small"), which needs many more queries because it encodes less data per query; chunk counts scale directly with message size for both methods.

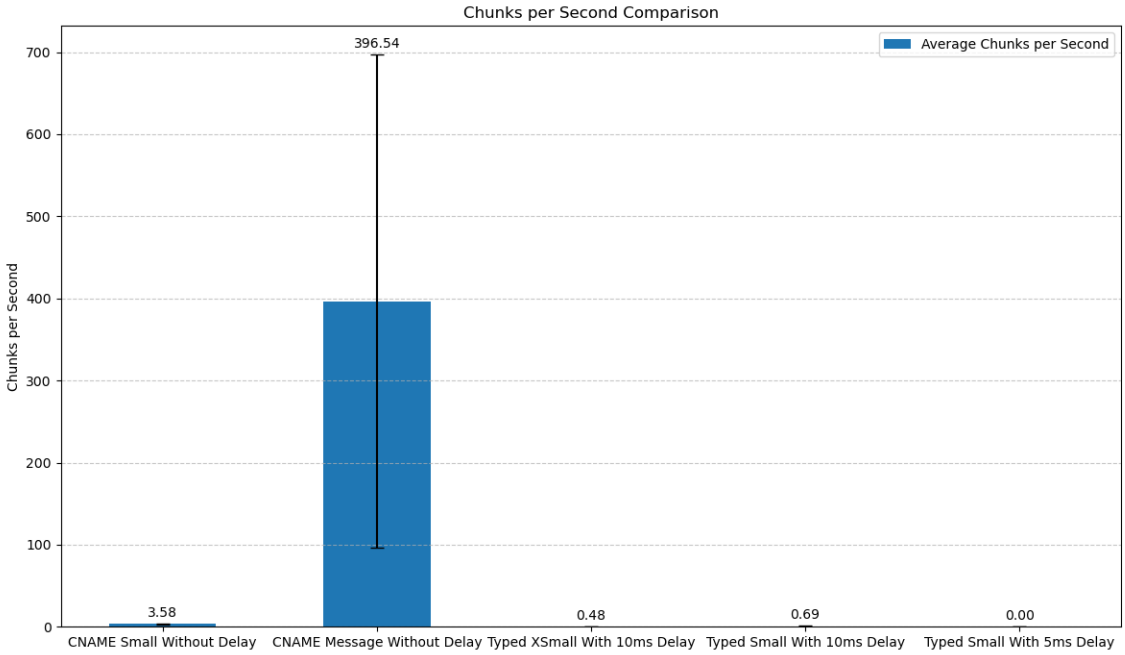

Figure 5 . Rate of DNS Query Transmission and Processing

This graph displays the rate of DNS query transmission and processing. The CNAME method without delay achieved a high rate ($\sim$ 400 chunks/sec), correlating with its high capacity, while the Typed methods were limited to very low rates $<$ 1 chunks/sec) by the imposed delays, with the 5ms delay scenario showing a zero rate, indicating transmission or processing failure.