NETWORK SECURITY TERM PROJECT PHASE 2

Yusuf Şahin

April 13, 2025

Contents

1 Introduction

1.1 Covert Channel

Covert channels are communication mechanisms that operate outside the boundaries of normal network protocols, enabling data transmission in a manner that avoids detection by unauthorized observers. In this project, a covert channel was implemented based on the technique described by Giffin et al. [1] in "Covert Messaging through TCP Timestamps".

The core idea behind the selected covert channel is to encode information into the least significant bit (LSB) of TCP timestamp values, which are typically perceived as random on slow connections. By modifying these LSBs in a controlled manner—using cryptographic techniques and delayed packet scheduling—it becomes possible to transmit a hidden message across seemingly normal TCP traffic. This method satisfies the properties of plausibility and undetectability which are fundamental to an effective covert channel.

The implementation consists of two core components:

A sender, running inside the sec container, that modifies TCP timestamps according to a keyed hash-based encoding protocol.

A receiver, running inside the insec container, that reconstructs the message by passively observing the LSBs of the timestamp values and performing inverse decoding.

The channel is tested in a controlled container-based environment with real packet transmission. This report presents the experimental setup, parameter variations (such as drop rate and occupation number), and evaluates the performance of the covert channel in terms of packet overhead, delivery success, and capacity.

2 Methodology

2.1 Implementation Details

The covert channel is implemented by embedding encoded message bits into the least significant bit of TCP timestamp values. This section outlines the design and logic of both the sender and receiver components.

2.2 Sender

The sender operates within the sec container and is responsible for encoding message bits into TCP timestamps. It is implemented as a man in the middle, i.e it captures tcp packets from *inkptsec* and encodes them with the cipher bits. For each outgoing TCP packet, the sender computes a keyed HMAC-SHA256 hash over the TCP header using a shared secret. Differing from the paper's SHA1 usage I implemented SHA256, a

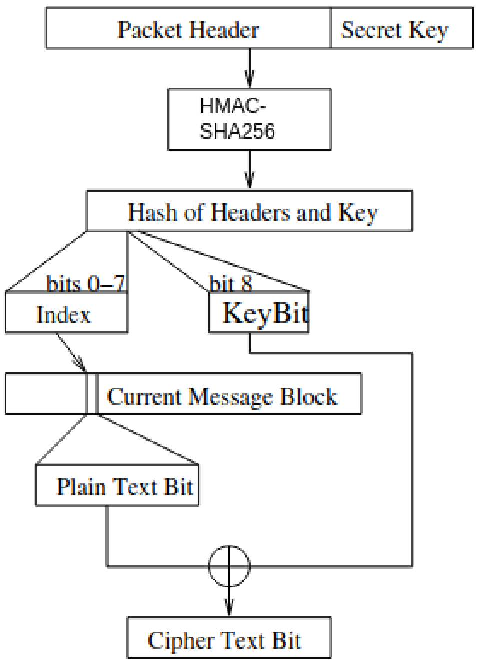


Figure 1: Sender, taken from:[1]

modern replacement, which also reduces collisions on calculation of the digest from the nonce(TCP header). From the resulting digest, it extracts:

1. A bit index to determine which bit of the message block to transmit.

2. A key bit used to key bit via XOR, producing the cipher text bit.

The timestamp value in the TCP header is accessed and, if necessary, incremented until its least significant bit matches the cipher text bit. If incrementing the timestamp changes the higher-order bits and thereby affects the nonce, the encoding process is recursively retried to maintain consistency.

Each bit is transmitted multiple times based on a fixed occupation number, and the sender tracks the number of transmissions per bit. When every bit in the current message block reaches the required occupation count, the block is considered transmitted, and the sender moves on to the next block.

In addition, the determined size of block in bits is 256. Before sending the block, blocks last 32 bits are the crc32 checksum of the message we are going to send.

Lastly, each bit in the block is sent atleast *occupation_number* of times. The sender stops encoding packets if a transmission count of a bit is larger than this number. What this means is that, since we calculate which bit to send from the digest our bit index selection is random. As a result we will send many bits more than *occupation_number*. So, this provides some level of reliability to the channel by introducing redundancy. In the

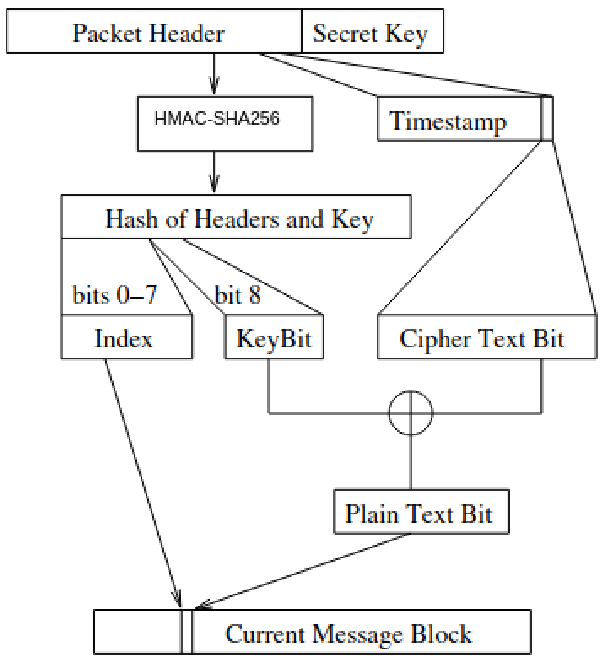


Figure 2: Receiver, taken from [1]

original paper it is claimed that sending 3000 packets provides 99.6% chance on sending every bit at least once.

In my implementation I run experiments with single block mode. There were some synchronization issues with my multi-blocked implementation(there is another branch called multi-block in my repo: [2]). The issue was that since communicating parties can't agree on which block of the message they are receiving, the received could not be checksum verified due to erroneous bits introduced from old or new blocks. I presume this was due to testing with high packet sending speeds as this was designed to work lower connection speeds.

2.3 Receiver

The receiver runs inside the insec container and passively captures TCP packets using the libpcap interface. Only incoming packets to insec network are sniffed as we encoded only outgoing packets from sec network. For each received packet, receiver extracts the TCP header and computes an HMAC-SHA256 digest using the shared secret key. Similar to the sender the receiver determines bit index and key bit from the digest2. At every received bit, receiver calculates crc32 checksum of the the first $256 - 32 = 224$ bits and compares it with the last 32 bits of the block. Upon verification it prints received message and then moves on to receive next block.

3 Experiments

For simulating an application running on top of TCP, a simple tcp echo server is implemented in the insec network. In secure network 30 applications opened sockets to insecure network and pinged each other. These are the outgoing packets that are encoded by man in the middle (called c-processor in the code). Bit Error Rate is calculated over comparison of sent bits and received message, i.e it is ratio of incorrect bits over total length. Also I could not provide confidence intervals since experiments took so much time to run.

I tested my implementation with sender delays and drop rates, and also varying occupation number. I will present and discuss results below. Also provide instructions here on how to run the code.

First, go into insec container. We will compile our receiver here.

```
make clean
make
./receiver
```

Make sure running "make clean". In the terminal screen you will see the message being formed bit by bit.

Then, go to c-processor

```
make clean
make
./processor <drop_rate_in_percent> <occupation_number>
```

I suggest running with low drop rates, and occupation number 2 for seeing results faster.

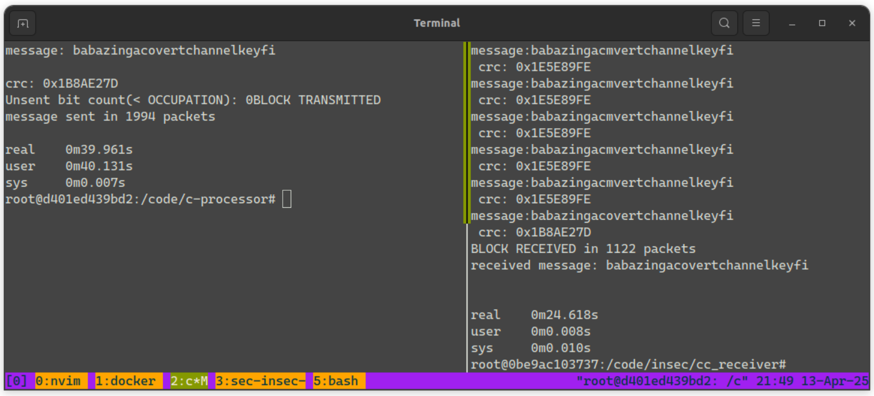
Now we will start our dummy application codes. Go into insec container.

```
cd ..
python3 receiver.py
```

Go to the sec container.

```
./run.sh <number_of_instances> <mean_delay_in_seconds>
```

this will spawn *number_of_instances* of process with the specified mean delay for sending each subsequent packet. From this point watch the terminal where we started ./receiver. Both sender and receiver should look like this 3:



```
message: babazingacovertchannelkeyfi

crc: 0x1B8AE27D
Unsent bit count(< OCCUPATION): 0BLOCK TRANSMITTED
message sent in 1994 packets

real    0m39.961s
user    0m40.131s
sys     0m0.007s
root@d401ed439bd2:/code/c-processor# 
```

```
message:babazingacmvertchannelkeyfi
 crc: 0x1E5E89FE
message:babazingacmvertchannelkeyfi
 crc: 0x1E5E89FE
message:babazingacmvertchannelkeyfi
 crc: 0x1E5E89FE
message:babazingacmvertchannelkeyfi
 crc: 0x1E5E89FE
message:babazingacmvertchannelkeyfi
 crc: 0x1E5E89FE
message:babazingacovertchannelkeyfi
 crc: 0x1B8AE27D
BLOCK RECEIVED in 1122 packets
received message: babazingacovertchannelkeyfi


real    0m24.618s
user    0m0.008s
sys     0m0.010s
root@0be9ac103737:/code/insec/cc_receiver#
```

```
[0] 0:nvim  1:docker  2:c*M 3:sec-insec- 5:bash        "root@d401ed439bd2: /c" 21:49 13-Apr-25
```

Figure 3: Sender and receiver

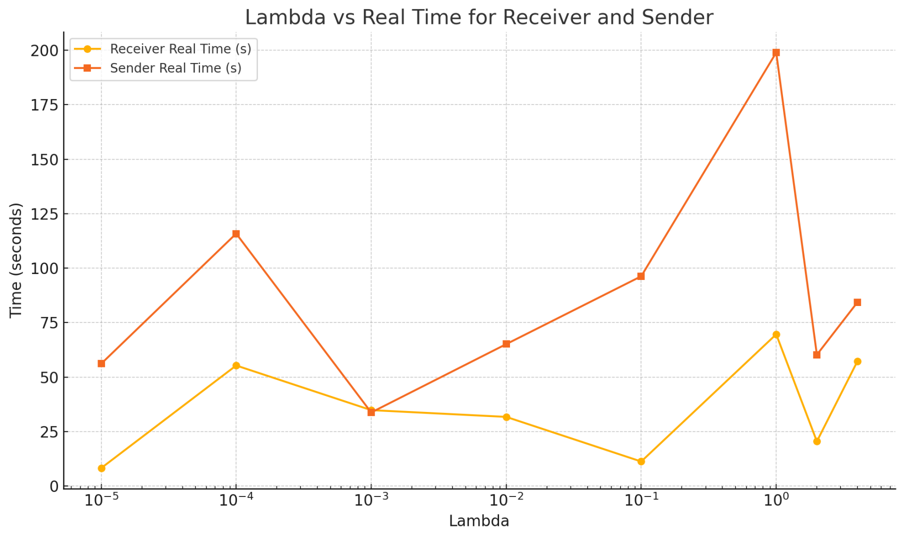


Figure 4: Mean value for sender vs reception times in the receiver

3.1 Sender delay effect on message reception times and generated packet counts

In this setting of experimentation, the sender application introduced with some delay according to exponential distribution 4. We can see that there is no clear correlation with increasing delays and reception times. This is due to fact that calculation bit index to send from digest. Selecting which bit to send this way introduces delay to reception, however sometimes by chance receiver can receive all the bits correctly to validate checksum hence receiving the block.

3.2 Drop rate vs Bit error rate

In the c-processor (encoder of the packets), drop rate introduced to encoded packets with some chance Figure 5. This simulates packet drops for encoded packets destined to insec. In this run for experiment $occupation\_number = 3$ is utilized. We can see that even for

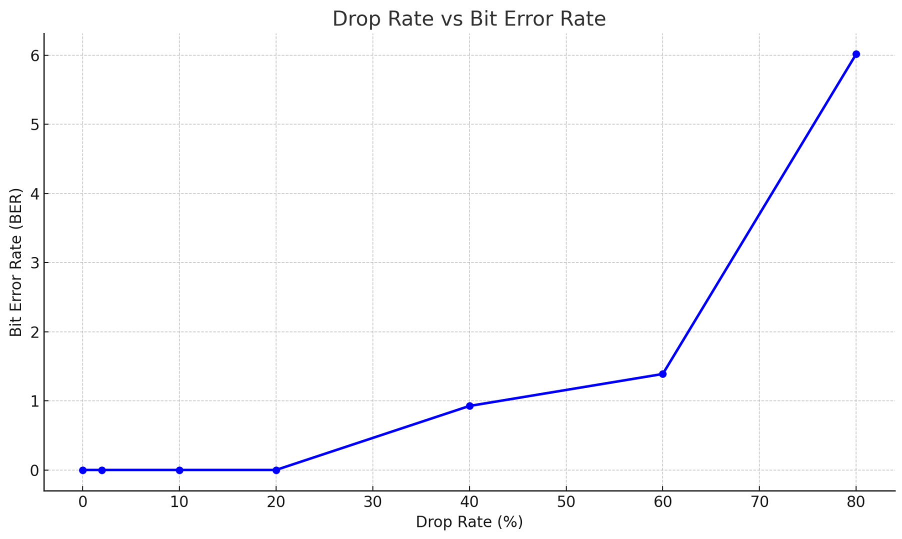


Figure 5: Drop rate vs Bit error rate (in percentages)

80% drop rate we have 6% bit rate error. Proving the reliability of the protocol.

3.3 Occupation number vs Bit error rate

To show effectiveness of increasing occupation number, 3 experiments conducted. For each drop rate %20, %40, %60 occupation level is increased. At a 20% drop rate Figure 7, the receiver was unable to reconstruct the message correctly at occupation levels 1 through 3 due to high bit error rates (e.g., 1.39 at level 1). Successful message recovery was achieved at occupation level 4 and beyond, where the BER dropped to zero. Under a 40% drop rate Figure 8, the channel demonstrated greater robustness: although the message could not be recovered at levels 1 and 2 (with BERs of 1.85 and 0.46), successful reception began at occupation level 3. For the 60% drop rate Figure 9 condition, the channel faced the most significant degradation, with very high BER observed at lower occupation levels (10.18 at level 1), and successful decoding occurring only at levels 4 and 5. These results confirm that increasing the occupation level improves reliability, compensating for higher drop rates by repeating each bit more frequently to ensure delivery.

It is also worth noting that increasing occupation number increases the outgoing packets from the sender. Figure6.

3.4 Packet re-ordering and bit error introduction

I didn't run experiments for packet reordering since as I have explained, we select index from digest. This means that sent bit selection can be called a random process. So the protocol is robust against reordering. As for introducing bit errors, i could not have enough time to include them. However, i think that the results won't differ from the experiments where drop rate is introduced. Since we send each bit many times

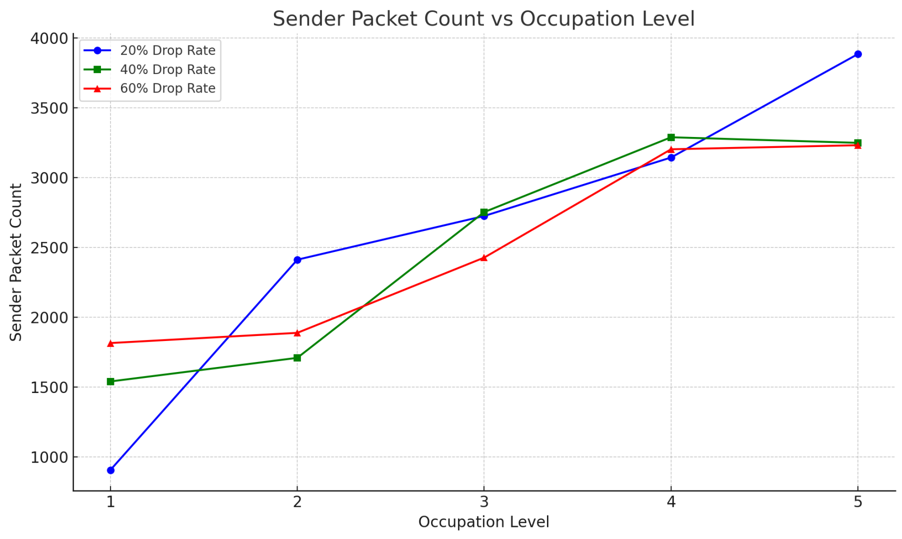


Figure 6: Sender packet count vs occupation number

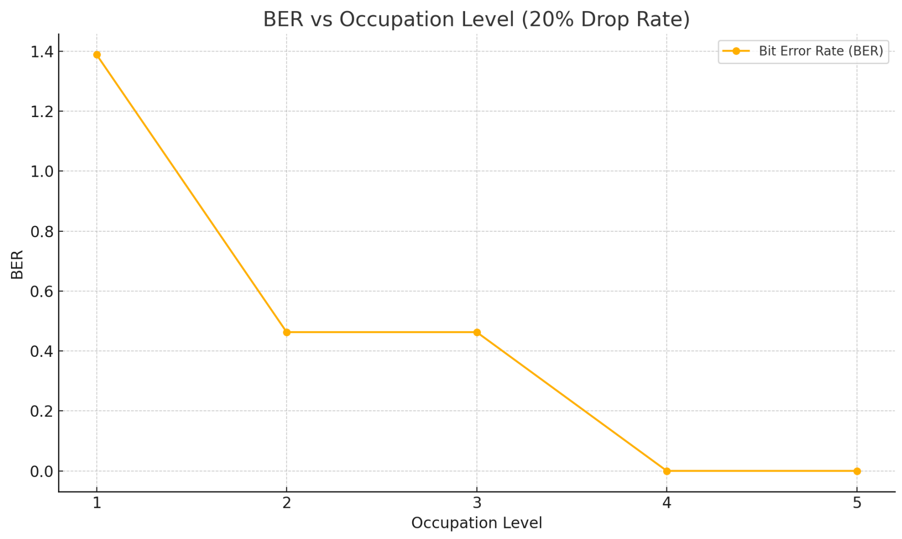


Figure 7: 20 % drop rate, BER vs occupation

redundantly.

3.5 Channel capacity

Channel capacity was estimated by dividing the number of received message bits by the total receiver transmission time in seconds. Specifically, the capacity was calculated as $C = \frac{l}{T}$, where l is the message length in bits (224 in our case), and T is the receiver time in seconds. The table below summarizes the capacity results under different drop rates and occupation numbers.1

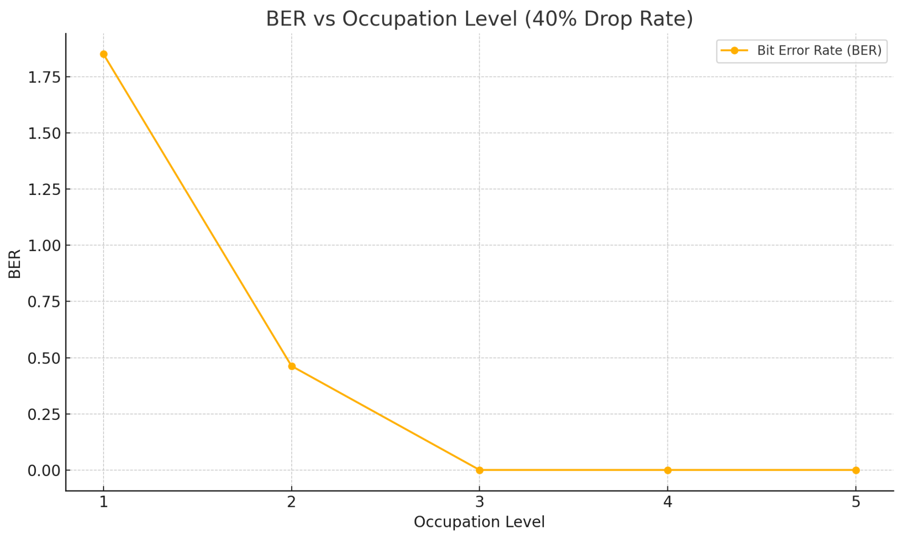


Figure 8: 40 % drop rate, BER vs occupation

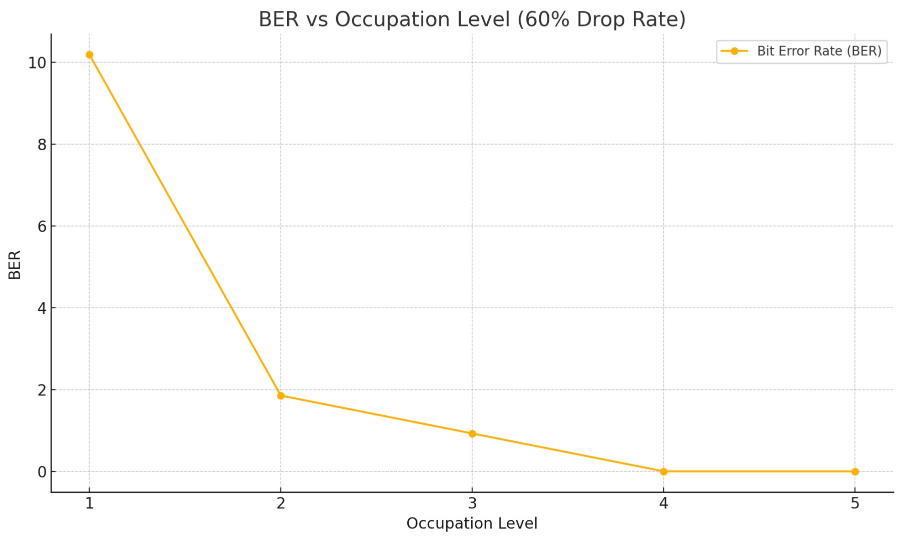


Figure 9: 60 % drop rate, BER vs occupation

4 Conclusion

In this project, a covert channel was implemented by modulating the least significant bit of TCP timestamp values, as described in Giffin et al.'s protocol. The system was evaluated under various conditions including adjustable sender delay, packet drop rates, and occupation numbers. Experimental results confirmed that the protocol is robust against packet loss, with reliable message delivery achieved by increasing the occupation number. Even under high drop rates (up to 60%), successful decoding was possible with sufficient redundancy. While sender-side delays did not show a consistent effect on reception time, the reliability of the channel remained high.

Drop Rate	**Occupation**	**Receiver Time (s)**	**Capacity (bits/s)**
0 %	3	10.243	21.87
20%	4	21.52	10.41
20%	5	42.00	5.33
40%	3	73.77	3.04
40%	4	62.51	3.58
40%	5	66.60	3.36
60%	4	162.82	1.38
60%	5	165.89	1.35

Table 1: Estimated channel capacity based on receiver times across varying drop rates and occupation levels

References

[1] Giffin, J., Greenstadt, R., Litwack, P., & Tibbetts, R. (2003). Covert Messaging through TCP Timestamps. In Lecture notes in computer science (pp. 194–208). https://doi.org/10.1007/3-540-36467-6_15

[2] https://github.com/yufusuf/netsec-project/tree/multi-blocks