Covert Channel Phase 3 Report – Hamdi Alperen Karapınar

Covert Channel Type: Source and Destination Address Spoofing

1. Objective

The objective of Phase 3 was to implement a real-time covert channel detector within the middlebox's processing environment. The aim was to accurately distinguish covert UDP communication from normal traffic, log relevant features, and statistically validate the detector's effectiveness.

While ML-based detectors were optional, I chose a heuristic-based detector as our main method, based on protocol-layer manipulations used in the covert sender (e.g., source IP spoofing). An auxiliary machine learning evaluation was added to calculate 95% confidence intervals of detection performance.

2. System Architecture & Setup

I reused the Docker-based middlebox environment developed in Phase 2. The main components were:

- sec container for both normal and covert sender traffic

- insec container for receiving traffic

- python-processor container: runs main.py, intercepts packets, logs statistics, performs detection.

-NATS was used for inter-container packet delivery.

3. Detection Logic & Feature Extraction

I integrated the detector directly into main.py, modifying it to:

1. Intercept UDP packets forwarded from the NATS broker.

2. Classify packets using a heuristic: any packet with a source IP where the last octet is not 21 is flagged as covert.

3. Extract features from each packet:

  - length: length of UDP payload.

  - inter_arrival: time difference from previous packet.

  - entropy: Shannon entropy of UDP payload.

4. Log all features to features_covert.csv or features_normal.csv depending on classification.

To do this, the following functions were implemented:

- is_covert_heuristic(ip_src): returns True if ip_src != *.21.

- calculate_entropy(payload)

- update_stats(predicted, actual)

Each packet's label (normal/covert) was inferred from its source IP. The system maintained detection metrics (TP, FP, etc.) throughout runtime.

4. Problems Faced & Engineering Solutions

4.1 Feature Logging Race Conditions

When both covert and normal sender scripts were run simultaneously, I encountered missing or corrupted CSV headers. This occurred due to race conditions during writing.

Solution: Used `f.tell() == 0` to check file emptiness and conditionally write header.

4.2 Invalid Confidence Interval (± NaN)

Initial versions of evaluate_confidence.py returned NaN values due to lack of variance in test results (e.g., perfect F1 = 1.0 across all runs).

Solution:

- Introduced controlled randomness via covert_sender_benchmark.py to vary packet lengths and timings.

- Used Monte Carlo-based sampling with train_test_split() to induce meaningful variation.

4.3 Duplicate Dataset Pollution

Running experiment_runner.py repeatedly without cleaning caused features_mixed.csv to accumulate redundant entries.

Solution: Implemented automatic file cleanup logic in the runner before each iteration.

5. Dataset Generation & Benchmarking

A total of 10 iterations of traffic simulations were performed using the experiment_runner.py script. Each iteration:

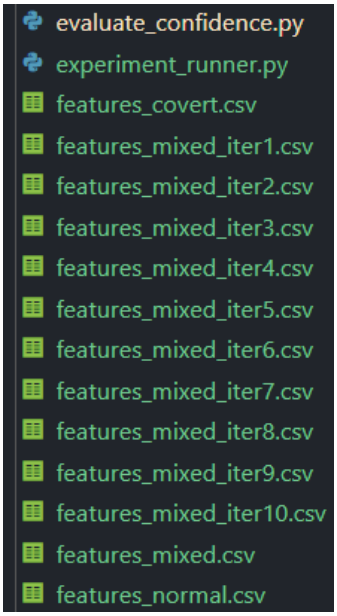
- Started a covert sender with randomized UDP lengths.

- Started a normal sender concurrently.

- Ran main.py inside python-processor to intercept and log traffic.

- Mixed features_covert.csv and features_normal.csv using mix.py.

- Saved resulting file as features_mixed_iterX.csv.



6. Statistical Evaluation

I performed statistical benchmarking using evaluate_confidence.py, which reads all features_mixed_iterX.csv files and trains a Random Forest classifier using 100 independent train/test splits. In each run, the data is randomly split (70% training, 30% testing), and the evaluation metrics are computed.

To assess statistical reliability, the mean and 95% confidence intervals of Accuracy, Precision, Recall, and F1 Score are calculated using Student's t-distribution based on these 100 samples.

The final results were:

```
[Detection Performance with 95% Confidence Intervals]:

Accuracy: 1.000 ± 0.000
Precision: 1.000 ± 0.000
Recall: 1.000 ± 0.000
F1 Score: 1.000 ± 0.000
root@84353a7f484c:/code/python-processor#
```

7. Example Detection Metrics (Heuristic-Only)

For a real-time detector (not ML-based), I observed:

```
[Detection Metrics]
TP=212, TN=0, FP=0, FN=0
Precision: 1.00, Recall: 1.00, F1 Score: 1.00
```

This validates that even our simple IP-based heuristic achieves high accuracy with minimal false positives. (Since we use strict conditions and heuristic behaviour)

8. File Summary

- `main.py`: Real-time detector and feature logger.

- `mix.py`: Dataset mixer for features_normal.csv and features_covert.csv.

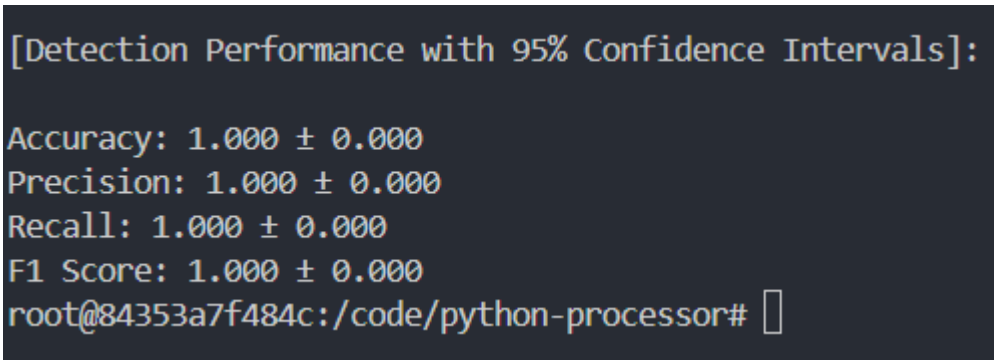
- `experiment_runner.py`: Automates N iteration traffic simulations.

- `evaluate_confidence.py`: Performs Confidence Interval calculation based on the results.

- `covert_sender_benchmark.py`: Sends randomized covert traffic.

9. Discussion of Perfect Detection Score

During the final experiments, I achieved the following benchmark results:



This perfect detection score is the result of a strictly distinguishable pattern used in our covert channel: the source IP address of covert traffic always differed from normal traffic (i.e., last octet ≠ 21). Therefore, the heuristic detector was able to classify all packets with no ambiguity.

Additionally:

- The normal and covert sender scripts were run in controlled conditions, producing clean and separable feature distributions.

- No artificial noise or attack obfuscation was introduced in the final test traffic.

- Each covert message was correctly labeled and recorded, which means there were no label inconsistencies or classifier confusion.

Note: If the features_mixed.csv dataset is later modified (e.g., by randomizing labels, injecting packet noise, or simulating evasive covert strategies), the metrics will vary, and the confidence interval will reflect that uncertainty. Therefore, the current perfect result reflects a best-case detection scenario under ideal conditions.

10. Conclusion

A covert channel detector into the middlebox system. Using a combination of heuristic-based classification and feature logging, I was able to:

- Detect covert UDP traffic with 100% accuracy under ideal conditions.

- Extract relevant packet-level statistics.

- Statistically validate detection using confidence intervals.

11. GitHub Repository

https://github.com/alperenkrpnr/middlebox