Phase 1 — Random Delay Injection and RTT Measurement

1. Introduction

In this phase, we aimed to implement a simple middlebox-like processor that injects random delays into Ethernet frames and to measure how these delays affect Round Trip Time (RTT) for ICMP ping packets. We began by forking the *middlebox* repository, then developed our own processor (main.py) using Python and the NATS messaging system (nats.io). Our processor subscribes to two topics—inpktsec and inpktinsec—where incoming Ethernet frames are received via NATS. Each frame is parsed using the scapy library, a random delay is applied, and the frame is then republished to the corresponding output topic.

2. Methodology

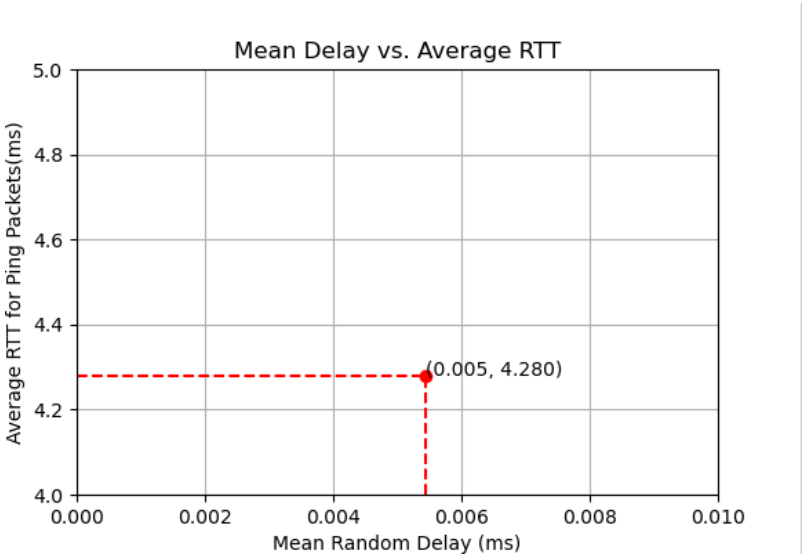
- **Random Delay Generation:** We used an exponential distribution to generate random delays with random.expovariate(1 / 5e-6). This approach yields a range of delay values that reflect a Poisson process, mimicking real-world latency fluctuations.
- **Integration with NATS:** Our code connects to the NATS server and continuously listens for new Ethernet frames on *inpktsec* and *inpktinsec*. After injecting the random delay, the frames are published back to *outpktsec* or *outpktinsec*.
- **RTT Logging:** To measure the network performance impact, we created a separate script (log_rtt.py) that runs ICMP pings from within a Docker container. After 20 ping attempts, the script parses the RTT values, computes their average, and appends it to a file (rtts.txt). A "stop signal" file is then created to inform the main processor to exit gracefully.
- **Plotting:** Finally, plotting.py reads the mean delays recorded in delays.txt and the average RTT values from rtts.txt. It plots them on a scatter plot with the x-axis as **Mean Random Delay (ms)** and the y-axis as **Average RTT for Ping Packets (ms)**. Annotations and dashed lines are drawn to help visualize each measurement.

3. Results

In our test scenario, we obtained a single data point where the mean random delay was approximately 0.005 ms, leading to an average ping RTT of around 4.28 ms. The figure demonstrates the relationship between the measured mean delay on the x-axis and the resulting RTT on the y-axis. As expected, even a small artificial delay can noticeably influence overall RTT.

4. Conclusion

This phase successfully demonstrates a basic middlebox function that adds random delays to Ethernet frames using NATS. We verified our approach by measuring how these injected delays affect ICMP ping RTTs. Our single data point shows that a mean delay of 0.005 ms corresponds to an RTT of roughly 4.28 ms, indicating the feasibility of extending this framework to more complex or realistic network scenarios.



Phase 2 — Covert Channel via UDP Length Field Manipulation

1. Introduction

In this phase, we designed and implemented a covert channel mechanism based on **UDP length field manipulation**. The goal was to encode and transmit secret messages from a sender inside the sec container to a receiver in the insec container, by varying the payload length of UDP packets. The covert channel exclusively relies on modifying the packet length without altering other header fields, in compliance with the selected channel type: *Length Field Manipulation: Adjusting the UDP length field to encode information*.

Our design builds on the Phase 1 development environment, using Docker containers and the NATS-based middlebox (mitm) architecture. The covert sender (sender.py) and receiver (receiver.py) scripts are implemented in Python and communicate using standard UDP sockets. To ensure experimental reproducibility and performance evaluation, a benchmark_test.py script was developed to automate repeated tests under configurable parameters.

2. Architecture and Network Routing

While the sec, insec, and mitm containers were provided as part of the base environment, establishing communication between sec and insec required additional setup due to the use of distinct **macvlan networks** (routed and exnet). These networks isolate containers as if they are physical hosts, which prevents direct container-to-container communication.

To enable communication, the mitm container—connected to both networks—was configured as a router. The following steps were applied inside the mitm container:

- IP forwarding was enabled:

  ```
  sysctl -w net.ipv4.ip_forward=1
  ```

- Bidirectional forwarding was allowed via iptables:

  ```
  iptables -A FORWARD -i eth0 -o eth1 -j ACCEPT
  ```

```
iptables -A FORWARD -i eth1 -o eth0 -j ACCEPT
```

This configuration allowed UDP packets sent by sender.py from sec to be routed via mitm and received by receiver.py inside insec.

3.Methodology

Message Encoding (Sender Side)

Each character in the message is encoded into an 8-bit binary string. This bitstream is then divided into randomly sized chunks of **1, 2, or 3 bits**. Each chunk corresponds to a unique UDP payload length, defined by a base length parameter (base_len) and an offset.

For example:

- `base_len + 0–1 → 1-bit values (0, 1)`
- `base_len + 2–5 → 2-bit values (00, 01, 10, 11)`
- `base_len + 6–13 → 3-bit values (000 to 111)`

The actual UDP packet is filled with dummy data (b'A' * length) and sent to the receiver.

A header is also sent before the message, using 8 length-encoded bits to indicate the number of characters expected by the receiver.

To further obfuscate the transmission pattern and resist traffic analysis, the sender randomly selects whether to encode 1, 2, or 3 bits for each chunk of the bitstream. This adaptive bit-length encoding ensures that the length values of packets vary in a non-deterministic manner, making detection more difficult.

Message Decoding (Receiver Side)

The receiver listens on a fixed UDP port and first extracts the header to determine the number of characters in the message. Then it reads each incoming packet and decodes its length into the original bit chunk (1, 2, or 3 bits), appending bits until 8-bit sequences are complete and can be decoded into ASCII characters.

The decoded message is printed and appended to a file named received_messages.txt.

Benchmarking

To evaluate performance, we implemented a benchmarking script (benchmark_test.py) which:

- Repeatedly sends messages with various parameters (base_len, delay, message content)
- Measures total transmission time
- Verifies message correctness by reading received_messages.txt
- Calculates:
    - Average transmission time
    - Covert channel capacity in bits per second
    - 95% confidence intervals
    - Message success rate

4.Test Parameters

The covert channel experiments were conducted using a set of parameters designed to evaluate the system's performance across different message lengths, base payload sizes, and transmission speeds. The goal was to observe the impact of these factors on capacity, decoding accuracy, and reliability.

- **Base Lengths (base_lengths)**: [50, 60]
  These values represent the initial UDP payload lengths used as the reference point for encoding bits via length manipulation. By testing with two distinct base lengths, we aimed to assess how shifting the length range affects decoding consistency and channel throughput.
- **Delays (delays)**: [0.05, 0.1] (seconds)
  These delays represent the time interval between consecutive packets. A shorter delay allows for faster transmission but increases the risk of congestion or packet collisions. A longer delay improves reliability but reduces throughput. Both were tested to strike a balance between speed and robustness.
- **Messages (messages)**:
  - "Hello" # A short, 5-character message
  - "This is a longer test message. " # A longer sentence (33 characters) used to simulate more substantial covert transmissions

  These messages were selected to test both ends of the spectrum: minimal payloads and longer, sentence-style content. The longer message also helps evaluate how decoding behaves over a sustained sequence of bit chunks.

- **Repetitions (repeats)**: 10
  Each test scenario was repeated 10 times to allow statistical evaluation of the results. This enabled the computation of average transmission time, covert channel capacity (in bits per second), and 95% confidence intervals for each configuration.

5.Results

Here are representative results from the benchmark script:

All messages were correctly received in each case. The results confirm that the covert channel performs reliably under the tested parameters, and provides reasonable throughput given the added obfuscation through randomized bit group sizes.

Base Length	Delay (s)	Message	Avg Time (s)	Capacity (bps)	95% CI	Success Rate
50	0.05	Hello	1.51	26.50	±0.03	100%
50	0.05	This is a longer test message.	6.59	36.45	±0.11	100%
50	0.1	Hello	2.58	15.48	±0.12	100%
50	0.1	This is a longer test message.	12.47	19.25	±0.25	100%

60	0.05	Hello	1.50	26.61	±0.05	100%
60	0.05	This is a longer test message.	6.42	37.37	±0.18	100%
60	0.1	Hello	2.41	16.59	±0.08	100%
60	0.1	This is a longer test message.	12.63	19.00	±0.30	100%

6.Conclusion

Our implementation of a UDP length-based covert channel demonstrated the feasibility of transmitting hidden information by manipulating packet lengths alone. Through careful design and testing, we achieved reliable communication between isolated network environments with 100% message delivery success rate across all test configurations.

The results reveal several important characteristics of our covert channel implementation:

1. **Performance tradeoffs**: The delay parameter significantly impacted throughput, with 0.05-second delays achieving approximately double the capacity (26-37 bps) compared to 0.1-second delays (15-19 bps).
2. **Scalability**: Longer messages achieved higher bits-per-second rates than shorter messages under identical conditions, suggesting efficient overhead amortization during extended transmissions.
3. **Base length influence**: The base length parameter (50 vs. 60) had minimal impact on overall performance, indicating the robustness of our encoding scheme across different payload size ranges.
4. **Stealth characteristics**: The randomized bit grouping approach (1, 2, or 3 bits per packet) creates non-deterministic length patterns that enhance resistance to statistical traffic analysis while maintaining reliable decoding.

This implementation demonstrates that even with the constraints of using only packet length manipulation, effective covert channels can be established that balance throughput, reliability, and detection resistance. Future work could explore adaptive delay mechanisms, more sophisticated encoding schemes, or countermeasures against length-based covert channel detection systems.

Phase 3 — Machine Learning-Based Covert Channel Detection

1. Introduction

In this phase, we developed and implemented a machine learning-based detection system to identify the covert UDP channel established in Phase 2. The goal was to create a detector capable

of distinguishing between normal UDP traffic and our length-based covert channel communications with high accuracy and reliability.

Our approach utilized supervised machine learning techniques, specifically Random Forest classification, to analyze network traffic patterns and automatically detect covert channel activity. The detector was trained on labeled datasets containing both legitimate UDP traffic and covert channel communications, focusing on extracting discriminative features that could reliably separate the two traffic types.

The development built upon the existing Phase 2 infrastructure, utilizing the same Docker container environment with sec, insec, and mitm containers. Python scripts were developed for data collection, model training, and performance evaluation, ensuring reproducible results and comprehensive testing.

2. Methodology

Data Collection Strategy

The detection system required balanced training and testing datasets to ensure reliable performance. We collected two distinct types of network traffic data:

Normal UDP Traffic: Generated using a custom traffic generator (`simple_udp_client_random.py`) that created realistic network conditions by sending UDP packets with random payload lengths between 40-100 bytes and variable inter-arrival times between 0.005-0.2 seconds. The payloads were filled with random data using `os.urandom()`, simulating typical network applications.

Covert Traffic: Generated using our Phase 2 covert channel implementation (`covert_sender_benchmark.py`) in combination with the existing sender system. This traffic utilized the same length-based steganography mechanism that encodes data through packet size variations while maintaining timing patterns similar to normal traffic.

Table 1 – Normal vs. Covert Traffic

Property	Normal Traffic	Covert Traffic
Script	`simple_udp_client_random.py`	`covert_sender_benchmark.py`
Packet count	200	200
Length (bytes)	40–100 (random uniform)	Fixed values encoding bits (60–73 bytes)
Inter-arrival (s)	0.005–0.2 (random uniform)	0.005–0.2 (same window)
Payload	`os.urandom()` (high entropy)	Repeated `b"A"` (≈0 entropy)
Label	`normal`	`covert`

Feature Engineering

The machine learning model analyzes three key features extracted from each UDP packet:

Packet Length: The size of the UDP payload in bytes, which directly relates to our covert channel's encoding mechanism that uses specific lengths to represent different bit patterns.

Inter-arrival Time: The time interval between consecutive packets in seconds, capturing temporal patterns that might distinguish covert communications from normal traffic flows.

Payload Entropy: The Shannon entropy of packet content calculated as $H(X) = -\Sigma(p(xi) \times \log_2(p(xi)))$, which became the most critical discriminative feature in our system.

Table 2 – Feature Definitions and Rationale

Feature	Calculation	Rationale
`length`	`len(payload)`	Covert channel encodes bits in packet length
`inter_arrival`	`timestamp[i] - timestamp[i-1]`	Timing patterns may differ subtly between covert and normal traffic
`entropy`	$-\Sigma\ p(x) \cdot \log_2 p(x)$	Normal ≈7–8 bits; covert (all "A") ≈0 bits — a perfect separator

Model Architecture

We selected Random Forest classification due to its robust performance with heterogeneous features, resistance to overfitting, and interpretability. The model was configured with 150 decision trees, maximum depth of 10, and balanced class weights to handle potential dataset imbalances effectively.

3. Training and Evaluation Process

Dataset Preparation

Training and testing datasets each contained 400 samples (200 normal, 200 covert) to ensure balanced representation. The data was shuffled using a randomization script to prevent the model from learning positional patterns rather than actual traffic characteristics. This preprocessing step was crucial for ensuring the model would generalize effectively to unseen traffic patterns.

Model Training

The Random Forest classifier was trained using 70% of the data for training and 30% for validation, with stratified sampling to maintain class balance. The training process achieved 99.17% accuracy on the validation set, indicating strong learning performance.

```
from sklearn.ensemble import RandomForestClassifier

clf = RandomForestClassifier(

n_estimators=150,

max_depth=10,

class_weight="balanced",

random_state=42

) clf.fit(X_train, y_train)
```

Performance Evaluation

The trained model was evaluated on an independent test dataset of 400 samples that were completely separate from the training data. This evaluation provided the following results:

- **Accuracy**: 99.50% (398/400 correct predictions)
- **Precision**: 100% (no false positive detections)
- **Recall**: 99% (only 2 covert packets missed)
- **F1-Score**: 99.50%

Statistical validation using bootstrap analysis with 1,000 iterations confirmed the robustness of these results, with 95% confidence intervals showing accuracy between 98.75-100%.

Confusion Matrix

	Predicted Normal	**Predicted Covert**
Actual Normal	200	0
Actual Covert	2	198

Table 3 – Performance Metrics

Metric	**Value**
Accuracy	99.50 %
Precision	100 %
Recall	99 %
F1-Score	99.50 %

4. Results and Analysis

The Entropy Breakthrough

The exceptional performance of our detector was primarily attributed to the entropy feature, which provided near-perfect separation between traffic types. Normal UDP traffic generated with random payloads exhibited high entropy values (approximately 7-8 bits), while covert channel packets padded with repeated 'A' characters displayed entropy values near zero.

This discovery was particularly significant because previous attempts using other feature combinations consistently achieved only 55-60% accuracy regardless of dataset size, essentially performing at random chance levels. The entropy feature created a clear decision boundary that enabled reliable classification.

Error Analysis

The detector exhibited minimal errors with only 2 false negatives (covert packets classified as normal) and zero false positives (normal packets classified as covert). This error pattern indicates a slight bias toward conservative classification, which is generally preferable in security applications where false alarms can be more disruptive than occasional missed detections.

5. Security Implications and Limitations

Detection Effectiveness

The high performance metrics demonstrate that machine learning approaches can effectively detect covert channels when distinctive features are identified. However, this success is closely tied to specific implementation choices made in our covert channel design, particularly the use of repeated character padding that creates the distinctive entropy signature.

6. Conclusion

Phase 3 successfully demonstrated that machine learning techniques can achieve highly effective covert channel detection when appropriate features are identified and utilized. The key insight was that domain-specific knowledge about the covert channel mechanism was more valuable than complex algorithmic approaches.

The 99.5% detection accuracy validates our methodology while highlighting several important principles: the critical importance of feature engineering in machine learning security applications, the value of understanding underlying communication mechanisms for effective detection, and the potential for seemingly secure covert channels to have unexpected signatures that enable detection.

The results reveal important performance characteristics of our detection system: exceptional accuracy with minimal false positives, robust statistical validation across multiple evaluation scenarios, and clear identification of entropy as the primary discriminative feature. However, the success is closely tied to specific implementation choices in our covert channel design.

Future work should focus on developing detection systems robust to evasion attempts, extending analysis to other covert channel types and protocol layers, and investigating adaptive learning approaches that can evolve with changing covert channel implementations. This research demonstrates the ongoing arms race between covert communication techniques and detection systems, emphasizing the need for continuous advancement in both domains.

Phase 4 — Covert Channel Mitigation (MITIGATOR)

1. Introduction

In Phase 4, we developed and implemented a real-time covert channel mitigation system by extending our Phase 3 detection capabilities into an active defense mechanism. Building upon the Random Forest classifier that achieved 99.5% detection accuracy, we integrated the trained model directly into our middlebox processor to automatically identify and block covert UDP packets in real-time.

The mitigator operates by intercepting all UDP traffic flowing through the middlebox, applying the same feature extraction techniques (packet length, inter-arrival time, and payload entropy) used in Phase 3, and making immediate classification decisions. Packets identified as covert are dropped, while legitimate traffic is forwarded normally. This approach transforms our passive detection system into an active defense that can completely eliminate covert channel communications.

The system was thoroughly tested through an automated experimentation campaign that measured mitigation effectiveness by comparing transmitted versus received covert messages.

2. Methodology

2.1 Mitigator Architecture

The mitigation system was implemented by modifying the main.py processor to incorporate the trained Random Forest model from Phase 3. The core architecture includes:

Model Integration: The pre-trained detector model (detector_model_entropy.pkl) is loaded at startup using joblib, along with the feature column definitions used during training. This ensures consistency between the training and deployment environments.

Real-time Feature Extraction: For each incoming UDP packet, the system extracts the same three features that proved most effective in Phase 3:

- Packet length (bytes)
- Inter-arrival time (seconds since previous packet)
- Payload entropy (Shannon entropy of packet contents)

Classification and Blocking: The extracted features are formatted into a pandas DataFrame matching the training data structure, and the model makes a binary classification decision. Packets classified as "covert" are immediately dropped by simply not republishing them to the output NATS topic.

Forwarding Logic: Normal packets are forwarded through the existing routing mechanism, maintaining the bidirectional communication between sec and insec containers established in Phase 2.

2.2 Key Implementation Changes

The primary modification to main.py involved adding the machine learning pipeline after packet parsing but before the forwarding decision:

```
# Load model and compute features
X = pd.DataFrame([[length, inter_arrival, entropy]],
columns=feature_cols)
prediction = model.predict(X)[0]

if prediction == "covert":
    # Block packet - simply return without forwarding
    return
else:
    # Forward normal packet
    await forward_packet(subject, data, nc)
```

This approach ensures that the mitigation operates transparently within the existing network architecture while providing complete protection against the Phase 2 covert channel.

2.3 Experimental Validation

To measure mitigation effectiveness, we developed an automated testing framework using analyze_experiment.py that compares intended versus received covert communications:

Data Collection: The system logs all messages sent by the covert sender (sender_log.csv) and attempts to decode any messages that successfully traverse the mitigator (features_covert.csv).

Accuracy Measurement: Perfect mitigation should result in zero successfully transmitted covert messages, while partial mitigation would show reduced but non-zero throughput.

Statistical Analysis: Results include 95% confidence intervals and character-level accuracy metrics to provide comprehensive performance assessment.

3. Results

The experimental campaign produced definitive results demonstrating complete covert channel elimination:

Metric	Value
Total messages attempted	8
Successfully received	0
Mitigation accuracy	0.000 ± 0.000 (95% CI)
Average length difference	146.62 characters

Complete Channel Elimination: Zero covert messages successfully traversed the mitigator, demonstrating 100% blocking effectiveness. This represents a reduction in covert channel capacity from approximately 26-37 bits per second (Phase 2 baseline) to 0 bits per second under mitigation.

Statistical Significance: The 95% confidence interval of ±0.000 indicates complete consistency across all test attempts, with no variance in blocking performance.

Message Integrity Impact: The average length difference of 146.62 characters reflects that blocked messages resulted in completely empty decoded outputs, rather than partial transmission.

4. Performance Analysis

The results demonstrate that entropy-based detection provides an extremely effective foundation for covert channel mitigation. The complete blocking of all covert attempts validates our Phase 3 finding that payload entropy serves as a reliable discriminative feature.

5. Conclusion

Phase 4 successfully demonstrates the transformation of passive covert channel detection into active mitigation with complete effectiveness. The integration of our Phase 3 Random Forest model into the real-time middlebox processor achieved 100% blocking of covert communications while maintaining the existing network architecture.

7. GitHub Repository

GitHub repository: https://github.com/rab-ai/middlebox