CENG519 - Project Phase 3 Report

1. Introduction

This phase 3 report presents an implementation of covert channel detection system specifically designed to identify TCP flags-based covert communications. The detection framework employs a multi-module heuristic approach combining TCP flags analysis, pattern detection, behavioral analysis, and statistical modeling in identifying.

2. Detector Architecture

The architecture consists of the following core components:

- Packet Processing Pipeline: Handles packet capture, parsing, and forwarding
- Multi-Module Detection Engine: Implements parallel analysis using specialized detectors
- Alert Management System: Processes and logs detection results
- Configuration Management: Provides tunable detection parameters
- Statistical Analysis Framework: Enables performance evaluation and optimization

3. Detection Methodology

3.1 Heuristic Detection Approach

The system employs a weighted heuristic scoring mechanism that combines outputs from four specialized analysis modules. Each module contributes a normalized score (0.0-1.0) representing the likelihood of covert channel activity:

Combined Score = (TCP_Flags × 0.4) + (Pattern × 0.3) + (Behavior × 0.2) + (Statistical × 0.1)

This weighted approach prioritizes TCP flags analysis as the primary indicator while incorporating contextual information from other analysis dimensions.

3.2 Detection Threshold Configuration

The system uses configurable detection thresholds defined in detection_config.py:

- Combined Score Threshold: 0.60 (primary alert trigger)
- TCP Flags Threshold: 0.70 (individual module threshold)
- Pattern Detection Threshold: 0.80
- Behavior Analysis Threshold: 0.60
- Statistical Analysis Threshold: 0.50

4. Detection Modules

4.1 TCP Flags Analyzer (tcp_flags_analyzer.py)

The TCP Flags Analyzer serves as the primary detection component, specifically designed to identify the flag combinations used by the target covert channel protocol.

Implementation Details:
- Tracks flag combinations across recent packet history (100-packet sliding window)
- Maintains frequency statistics for unusual flag patterns
- Implements signature-based detection for known covert patterns

Detection Logic:

```
def _is_covert_flag_combo(self, flag_combo):
    covert_combinations = ['AS', 'AU', 'AP', 'AF']  # SA, UA, PA, FA patterns
    return flag_combo in covert_combinations
```

Scoring Mechanism:
- Covert flag detection: +0.8 score
- Suspicious port usage (31337): +0.3 score
- Unusual flag combinations: +0.5 score

4.2 Pattern Detector (pattern_detector.py)

The Pattern Detector analyzes packet sequences to identify covert communication sessions and data transmission patterns.

Key Features:
- Per-connection sequence tracking with 100-packet history
- Session state management for covert channel protocols
- Temporal pattern analysis for identifying structured communications

Detection Signatures:
- Session Start: SA flag combination (+0.6 score)
- Data Transmission: UA (bit 1) and PA (bit 0) patterns (+0.7 score)
- Session End: FA flag combination (+0.5 score)

Implementation Approach:
The module maintains connection-specific packet sequences and applies state machine logic to identify covert channel session lifecycle events.

4.3 Behavior Analyzer (behavior_analyzer.py)

The Behavior Analyzer evaluates protocol-level anomalies and unusual network behaviors that may indicate covert communications.

Analysis Dimensions:
- Port Usage Patterns: Identifies non-standard port usage (e.g., port 31337)
- Flag Usage Frequency: Detects unusual utilization of rarely-used TCP flags
- Connection Characteristics: Analyzes connection establishment and teardown patterns

Scoring Logic:
- Suspicious port detection: +0.4 score
- URG flag usage (rarely used in normal traffic): +0.3 score

4.4 Statistical Engine (statistical_engine.py)

The Statistical Engine performs frequency analysis and timing-based anomaly detection to identify statistical signatures of covert communications.

Statistical Measures:
- Flag combination frequency analysis
- Temporal pattern analysis using timing statistics
- Anomaly detection based on historical baselines

Detection Algorithm:

```
def analyze(self, packet_info):
    flag_combo = ''.join(sorted(flags))
    if flag_combo in ['AU', 'AP']:  # Covert flags
        score += 0.4
    return min(score, 1.0)
```

5. Alert Management System

5.1 Alert Generation (alert_manager.py)

The Alert Manager processes detection results and generates structured alerts for security monitoring and analysis.

Alert Structure:

```
{
    "alert_id": 1,
    "timestamp": "2024-XX-XX...",
    "packet_info": {
        "src": "IP:Port",
        "dst": "IP:Port",
        "flags": ["A", "U"],
        "direction": "sec_to_insec"
    },
    "detection_scores": {
        "tcp_flags": 0.8,
        "pattern": 0.7,
        "behavior": 0.4,
        "statistical": 0.4
    },
    "combined_score": 0.72,
    "severity": "HIGH"
}
```

Alert Persistence:
- JSON Lines format for efficient processing
- File-based storage in alerts.jsonl
- Real-time logging for immediate response

5.2 Severity Classification

Alerts are classified based on combined detection scores:
- HIGH: Combined score > 0.8
- MEDIUM: Combined score 0.6-0.8
- LOW: Combined score < 0.6 (not typically alerted)

6. Delay Processing Integration

The system incorporates random delay processing capabilities from Phase 2 development, adding 0-100ms delays to each forwarded packet to simulate realistic network conditions and provide timing analysis capabilities.

Implementation:

```
random_delay = random.uniform(DELAY_RANGE[0], DELAY_RANGE[1])  # 0-0.1 seconds
delay_category = self.get_delay_category(random_delay)
await asyncio.sleep(random_delay)
```

7. Test Framework and Methodology

7.1 Automated Testing Framework (test_framework.py)

The testing framework implements a comprehensive statistical evaluation methodology designed to assess detection performance across multiple dimensions.

Test Campaign Structure:
1. Standard covert test
2. Shorter covert message (might be missed)
3. Covert with noise (TCP traffic mixed in)
4. Covert with random delay
5. Normal Traffic Tests

7.2 Detection Accuracy Testing

Methodology:
- Sample Size: 100 test runs per scenario. (can be changed)
- Test Types: Various covert channel detection vs. normal traffic classification
- Metrics Calculation: Confusion matrix analysis
- Use of %95 confidence interval

Performance Metrics:

```
True Positives (TP): 40
True Negatives (TN): 42
False Positives (FP): 8
False Negatives (FN): 10

Accuracy: 0.820 ± 0.075
Precision: 0.833
Recall: 0.800
F1-Score: 0.816
F2-Score: 0.806
```

8. Detection Parameters (detection_config.py)

The system provides comprehensive configuration management through centralized parameter files:

```
DETECTION_THRESHOLDS = {
    'tcp_flags_anomaly': 0.7,
    'pattern_detection': 0.8,
    'behavior_analysis': 0.6,
    'statistical_analysis': 0.5,
    'combined_score': 0.60
}
```

9. Running Test Campaigns

9.1 Automated Testing Execution

Test Execution:

```
# Navigate to project directory
```

```
cd /path/to/middlebox/

# Run comprehensive test campaign
python3 test_framework.py

# Monitor test progress
tail -f phase3_results.json
```

9.2 Manual Validation Testing

Individual Component Testing:

```
# Test covert channel detection
docker exec -d insec python3 /code/insec/covert/receiver.py
docker exec sec python3 /code/sec/covert/sender.py "TESTMSG"

# Generate normal traffic for baseline
docker exec sec ping -c 10 insec

# Check detection results
docker exec detector cat /results/alerts.jsonl
docker logs detector | grep "ALERT"
```

10. Results Analysis and Interpretation

10.1 Statistical Significance

The test framework implements proper statistical methodologies:
- Confidence Intervals: 95% CI
- Sample Sizes: Minimum 100 runs for statistical significance
- Standard Deviations: Reported for all continuous metrics

10.2 Confusion Matrix Analysis

Expected Results Format:

```
Confusion Matrix:

                Predicted
```

```
              Covert  Normal
Actual Covert   40    10
      Normal    8     42
```

13. Conclusion

This implementation is a heuristic approach to TCP flags covert channel detection. The modular architecture enables easy extension, while the testing framework ensures performance validation.

The weighted heuristic approach, combining specialized detection modules with configurable thresholds, provides a solution that achieves high detection accuracy while remaining adaptable to different network environments and threat scenarios. The integration of delay processing capabilities further enhances the system's utility for both detection and network performance analysis.