Lightweight DDoS Alarm System: Comparative Analysis of Detection Methods for Home Networks

Project Report

Liam Calder

April 18 2025

CSC 466

Table of Contents

1. Executive Summary

This report evaluates my lightweight DDoS alarm system, which is optimized for resource-constrained home networks. I compared three detection strategies for my final report testing:

- **Auto-Threshold (SYN=31, UDP=137)**: Dynamically calibrated over a 12-hour training window of regular traffic
- **Manual Threshold (SYN=100, UDP=1000)**: Fixed user-specified values as a fallback option
- **Anomaly Detection (K-means clustering)**: Trained for 12 hours to establish a normal-traffic baseline with auto-calibrated parameters

A one-hour no-attack baseline test was used to measure false positives and resource usage. Six attack scenarios (two 10-minute floods, two 5-minute floods, and two 30-second short attacks) probed each method's detection latency, coverage, and memory overhead. Key findings include:

- Threshold-based methods detect short attacks quickly (2–8 seconds for UDP floods). Still, they may overestimate the end time and occasionally misdetect large, legitimate spikes, resulting in one false positive each during the baseline and acquiring more in the attack phase.
- The Anomaly approach yields zero false positives but takes 20–30 seconds to confirm short bursts, often missing large portions of 30-second attacks. Memory usage is also higher, around 170–190 MB.
- All methods accurately detect long floods (5–10 minutes) with approximately 90–99% coverage.

For a typical home deployment, auto-threshold is recommended as a balanced solution. Anomaly detection is best suited for environments that are sensitive to false positives.

2. Introduction

2.1 Background and Motivation

Distributed Denial of Service (DDoS) attacks pose significant threats to network availability across all scales, from corporate data centers to smaller home networks. While enterprise-level Intrusion Detection Systems (IDS) like Snort, Suricata, and Fail2Ban offer comprehensive protection, they are typically too resource-intensive for home routers or single-board computers. Additionally, these solutions often require sophisticated configuration and maintenance beyond the technical capabilities of average home users.

This project addresses this gap by designing a lightweight, real-time DDoS alarm system with minimal overhead, while maintaining high detection accuracy. Rather than attempting to prevent attacks, the system focuses on timely detection and alerting, enabling users to take appropriate actions to protect their networks.

2.2 Objectives and Scope

The primary objectives of my project were to:

- Develop a modular, Python-based detection tool supporting both threshold-based and anomaly-based methods, allowing for direct comparison
- Create a system that can run effectively on resource-constrained devices typical in home network environments
- Conduct realistic tests under baseline and multiple DDoS flood conditions to measure key metrics (false positives, detection times, coverage, memory usage, CPU overhead)
- Compare the trade-offs among these detection strategies in the context of a small home network
- Provide insights into which approach best suits different home network environments based on their specific requirements.

The project scope focused on detecting volumetric attacks, specifically SYN and UDP floods, rather than application-layer attacks or more sophisticated multi-vector approaches.

3. System Design and Implementation

3.1 Threshold-Based Methods (Auto & Manual)

The system can be configured for two threshold-based detection variants:

Auto-Threshold: This approach dynamically calibrates detection thresholds based on observed network behaviour.

The system:

- Learns typical SYN/UDP packet rates over a user-set period (12 hours in final testing for this report)
- Calculates statistical properties of regular traffic (mean, standard deviation)
- Sets thresholds with a configurable safety margin (3× above normal maximums as a cushion)
- For the final test, the thresholds were given to be SYN=31, UDP=137 by the 12-hour trained alarm

Manual Threshold: This approach is a user-defined fallback for a user familiar with their home network traffic:

- Direct input of SYN and UDP packet count thresholds (SYN=100, UDP=1000 in my tests)
- Straightforward implementation with no training period required
- Risks of misconfiguration if the user lacks insight into their network's normal behaviour
- Serves as a practical option when immediate deployment is necessary

Both methods share an everyday detection workflow:

1. Count SYN and UDP packets within a configurable time window (default: 10 seconds)
2. Compare counts against respective thresholds
3. Trigger an alert when thresholds are exceeded

4. Track attack duration and source IP addresses until traffic returns below threshold levels

[Section 8.1 for code]

3.2 Anomaly-Based Method (K-means)

The anomaly detection implementation uses an unsupervised machine learning approach with several sophisticated features:

Baseline Establishment:

- Maintains a historical database of (SYN_count, UDP_count) pairs collected in 10-second windows
- Requires a training period to collect normal traffic data to avoid being overly sensitive
- Stores traffic patterns in persistent JSON files to maintain knowledge across system restarts
- Carefully excludes anomalous traffic from the baseline to prevent "pollution" of normal patterns

Dual Detection Mechanism:

1. **Volume-based detection**: Flags traffic if it significantly exceeds historical maximums (approximately 2× historical peaks)
2. **Pattern-based detection**: Uses K-means clustering to identify unusual traffic patterns regardless of absolute volume

K-means Implementation:

- Dynamically determines the optimal number of clusters based on data size (1-5 clusters)
- Auto-calibrates distance thresholds using statistical properties (mean + 2-3× standard deviation)
- Measures Euclidean distance from new traffic points to their nearest cluster centroid
- Flags points exceeding the calibrated distance threshold as potential anomalies

Consecutive Anomaly Confirmation:

- Requires multiple consecutive anomalous windows (default: 3) to confirm an attack
- This design choice significantly reduces false positives from brief traffic spikes
- Maintains state across windows to track potential developing attacks
- Provides detailed reason codes explaining why traffic was flagged (volume vs. pattern anomaly)

The anomaly detector also includes:

- Intelligent historical data management (preserving both the oldest and newest data points)
- Regular recalibration of detection parameters as new normal data is collected
- Detailed logging of detection decisions with distance metrics and rationales

[Section 8.2 for code]

3.3 Code Architecture and Key Components

The system is organized as a modular Python application with the following key components:

ThresholdDetector Class:

- Maintains running counts of SYN and UDP packets
- Tracks source IP addresses for attribution
- Implements detection logic comparing counts to thresholds
- Provides update() and reset() methods for window-based processing

AnomalyDetector Class:

- Manages traffic history collection and persistent storage
- Implements K-means clustering using scikit-learn
- Provides dual-detection logic (volume and pattern analysis)
- Handles training mode and baseline establishment
- Auto-calibrates parameters based on observed traffic

Packet Capture:

- Uses Scapy for efficient packet processing
- Runs in a dedicated thread to maintain responsiveness
- Implements a non-blocking design to prevent packet loss during processing

Network Interface Management:

- Provides a user interface for selecting monitoring interfaces
- Uses psutil to map low-level interface names to user-friendly names
- Validates interface availability and permissions

Logging & Alerting:

- Creates time-stamped metrics files for each session
- Records detailed attack information (type, duration, sources, peak rates)
- Log system performance metrics (CPU, memory usage)
- Provides real-time console alerts with attack details
- Stores traffic histories in JSON format for persistence

Main Control Loop:

- Manages the processing of each packet or window
- Coordinates between detection methods
- Handles the lifecycle of attacks (start, ongoing, end)
- Processes user input and configuration options

The code is structured to allow easy switching between detection methods without needing to restart, enabling direct comparison under identical conditions.

3.4 Training Process and Parameter Calibration

Both the auto-threshold and anomaly alarms require a training period to establish effective detection parameters:

Auto-Threshold Training:

- Monitors traffic for a user-specified duration (12 hours in my final tests)
- Calculates statistical properties of packet counts (mean, max, standard deviation)
- Recommends thresholds with a configurable safety margin above normal maximums
- Provides these recommendations to the user, who can accept or modify them

Anomaly Detection Training:

- Collects a minimum of 10 windows (configurable) of baseline data (minimum training windows)
- Indicates training progress to the user
- Prevents alerts during the training phase
- Automatically transitions to detection mode when sufficient data is collected

Parameter Auto-Calibration: The anomaly detector implements sophisticated parameter calibration:

- **n_clusters**: Automatically determined based on data size:
    - Small datasets (5-30 points): 1-2 clusters
    - Medium datasets (30-100 points): 2-3 clusters
    - Large datasets (>100 points): Up to 5 clusters
- **distance_threshold**: Calculated using statistical properties:
    - Base threshold = mean_distance + (standard_deviation × multiplier)
    - Multiplier ranges from 2 to 3, depending on desired sensitivity
    - Additional minimum threshold based on historical traffic maximums

This automatic calibration removes the need for manual parameter tuning while ensuring the system adapts to the specific characteristics of each network environment.

4. Testing Methodology

4.1 Baseline Phase (No Attacks)

To establish a performance baseline and measure false favourable rates:

- Each detection mode (auto-threshold, manual threshold, anomaly) was monitored for 1 hour under normal network conditions
- No artificial attacks were generated during this period
- All alerts were logged as false positives

- Memory usage was sampled every 30 seconds
- CPU utilization was tracked throughout the testing period
- Normal network activities (web browsing, video streaming, etc.) continued during testing

4.2 Attack Scenarios (UDP & SYN; Short, Medium, Long)

Six distinct attack scenarios were created to test detection capabilities across different attack types and durations:

1. **First UDP Flood**: 10 minutes @ ~144 packets per second (14:07:19 → 14:17:19)
2. **First SYN Flood**: 5 minutes @ ~210 packets per second (14:21:09 → 14:26:09)
3. **Short UDP Flood**: 30 seconds @ ~202 packets per second (14:28:02 → 14:28:32)
4. **Short SYN Flood**: 30 seconds @ ~249 packets per second (14:29:17 → 14:29:47)
5. **Second UDP Flood**: 5 minutes @ ~193 packets per second (14:30:39 → 14:35:39)
6. **Second SYN Flood**: 10 minutes @ ~237 packets per second (14:39:16 → 14:49:16)

These attacks were generated using flooding scripts that aimed to maintain a consistent packet rate. However, packet rates fluctuated from the nominal targets due to traffic generator overhead and network conditions.

Each attack was run with all three detection methods monitoring simultaneously to ensure consistent conditions across methods.

4.3 Data Collection & Logging

The following metrics were collected for comprehensive analysis:

Temporal Metrics:

- Actual attack start and end times (ground truth)
- Detected attack start and end times for each method
- Missed detection time (portions of actual attacks not detected)
- Overreported time (false detection beyond actual attack duration)
- Coverage percentage (detected portion / total actual duration)

Resource Usage:

- Memory consumption (minimum, average, maximum) during baseline and attacks
- CPU utilization patterns during detection events
- Peak resource usage during detection events

Accuracy Metrics:

- False positives during baseline monitoring
- True positives during attack scenarios
- Precision, recall, accuracy, and F1 scores for each method

All metrics were automatically logged to time-stamped performance files, and additional system state information was captured during significant events, such as attack detection, the end of an attack, and confirmation of an anomaly.

[Section 8.3 for code.]

5. Results & Analysis

5.1 Baseline Results (No Attacks)

Detection Method	Training	Average Memory Usage	Range of Memory Usage	False Positives Reported	Cpu Usage
Auto-Threshold (31/137)	12 hours	~99.76	58.04 – 176.21	1	~0.1%
Manual Threshold (100/1000)	User set	~101.17	50.74 – 177.95	1	~0.1%
Anomaly Detection (K-means)	12 hours	~167.63	142.38 – 181.60	0	~0.1%

Key Takeaways:

- Both threshold-based methods triggered one false positive each during the one-hour baseline period, corresponding to legitimate traffic spikes (peak ~1175 packets per second during a large file download)
- The anomaly alarm produced zero false positives, demonstrating its ability to adapt to normal traffic variations
- Anomaly detection consumed significantly more memory (~168 MB average) compared to threshold-based methods (~100 MB average)
- CPU usage remained consistently low (~0.1%) for all methods during regular operation, with occasional spikes during processing windows

The false positives in the threshold methods highlight their sensitivity to legitimate traffic bursts, while the consecutive window requirement in the anomaly alarm effectively filters these out.

5.2 Attack Detection Performance

Attack	Method	Missed	Over	Temporal Accuracy	Coverage
Long UDP (600s)	Auto Threshold	3s	13s	97.3%	99.5%
	Manual Threshold	8s	11s	91.8%	93.7%
	Anomaly	63s	13s	87.3%	89.5%
Medium SYN (300s)	Auto Threshold	110s	13s	59.0%	63.3%
	Manual Threshold	100s	13s	59.0%	63.3%
	Anomaly	130s	13s	52.3%	56.7%
Short UDP (30s)	Auto Threshold	6s	16s	26.7%	80.0%
	Manual Threshold	6s	6s	60.0%	80.0%
	Anomaly	26s	16s	-40.0%	13.3%
Short SYN (30s)	Auto Threshold	2s	12s	53.3%	93.3%
	Manual Threshold	2s	12s	53.3%	93.3%
	Anomaly	22s	12s	-13.3%	26.7%
Medium UDP (300s)	Auto Threshold	2s	14s	94.7%	99.3%
	Manual Threshold	12s	14s	91.3%	96.0%
	Anomaly	22s	14s	88.0%	92.7%
Long SYN (600s)	Auto Threshold	8s	18s	95.7%	98.7%
	Manual Threshold	8s	18s	95.7%	98.7%
	Anomaly	28s	18s	92.3%	95.3%

Metrics Explained:

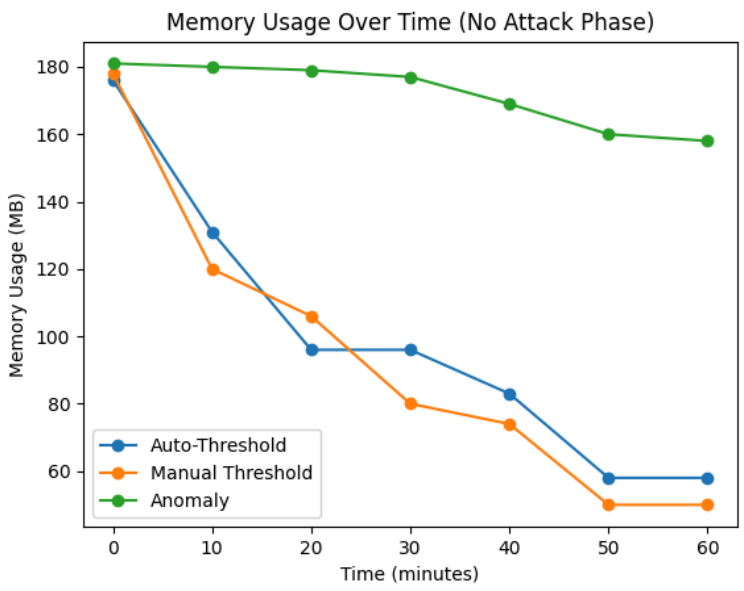
- **Missed**: Time between actual attack start and detection, plus any portions missed during the attack
- **Over**: Time the alarm remained active after the actual attack ended
- **Temporal Accuracy**: 1 - ((Missed + Over) / Attack Duration): measures overall timing precision
- **Coverage**: 1 - (Missed / Attack Duration): portion of the attack successfully detected

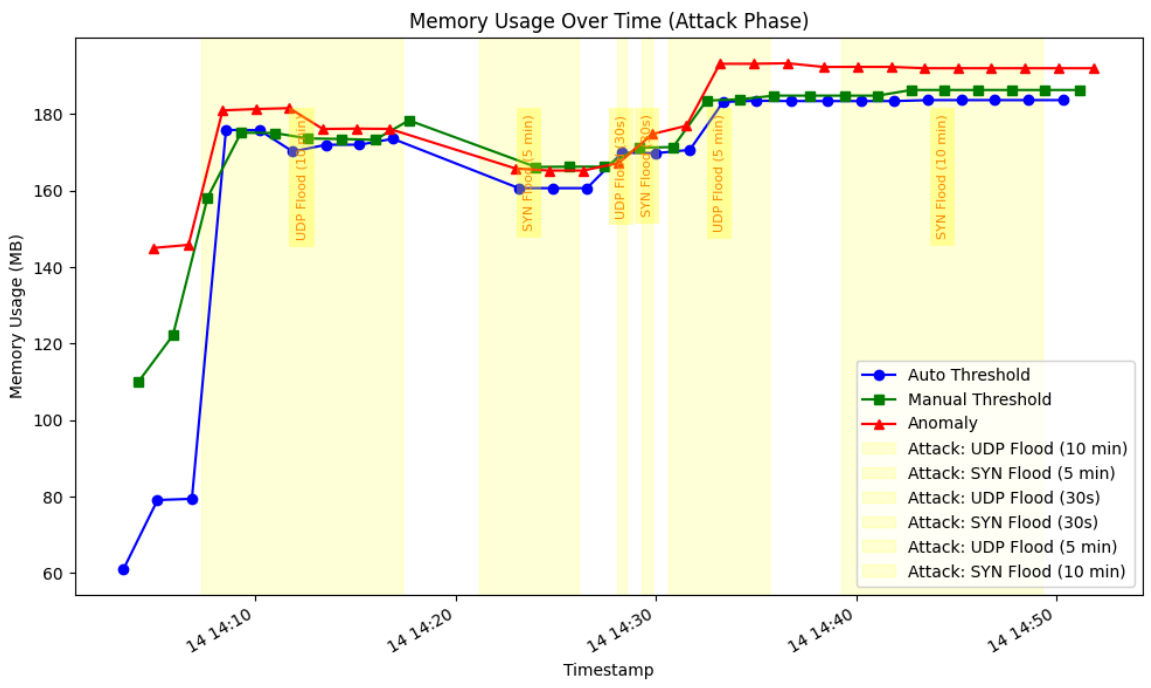
Key Observations:

- **Short Attacks (30s)**: Threshold methods detected quickly (2-6s) but with moderate overreporting. Anomaly detection missed most of the attack duration due to its requirement for a consecutive window, resulting in poor coverage (13-27%).
- **Medium Attacks (300s)**: All methods struggled more with the SYN flood than the UDP flood, potentially due to normal background SYN traffic. Coverage ranged from 53% (anomaly) to 99% (auto-threshold for UDP).
- **Long Attacks (600s)**: All methods performed well, with coverage >89% across the board. The Auto-Threshold Alarm achieved nearly perfect coverage (99.5%) for the UDP flood.
- **Detection Delay**: Consistent pattern of anomaly detection requiring 20-30 seconds longer to confirm attacks compared to threshold methods.
- **Overreporting**: All methods tended to extend alerts 10-18 seconds beyond the actual end times of attacks, likely due to residual traffic processing within the detection window.

The data shows a clear trade-off: threshold methods detect attacks more quickly and with better coverage for short bursts, while anomaly detection sacrifices some detection speed for improved false positive performance.

5.3 Memory & CPU Consumption





During Baseline vs. **Attack**:

Method	Baseline (MB)	Attack (MB)	Max	CPU
Auto Threshold	~101	~153	~184	~0.1%
Manual Threshold	~107	~166	~186	~0.1%
Anomaly	~169	~178	~193	~0.1%

Observations:

- Memory usage increased during attacks for all methods, with threshold methods showing a more significant relative increase
- Anomaly detection maintained consistently higher memory usage throughout testing
- Maximum memory usage occurred during peak attack periods for all methods
- CPU usage remained minimal (~0.1%) with occasional spikes during processing windows
- The memory differential between anomaly and threshold methods (60-70 MB) would be significant in very constrained environments

5.4 Overall Comparison

During the attack testing phase, Auto-Threshold Alarm recorded 2 additional false positives, while Manual Threshold Alarm recorded 1 additional false positive. The heightened sensitivity of Auto-Threshold (SYN=31, UDP=137) resulted in these extra alerts, which were triggered by regular traffic spikes. In contrast, Manual Threshold, with its higher setting (SYN=100, UDP=1000), effectively limited the extra false positives to one. In total, Auto-Threshold Alarm produced 3 false positives and Manual Threshold produced 2 false positives across all testing, while Anomaly Alarm produced zero false positives throughout all testing, demonstrating its ability to adapt to normal traffic variations even during the attack phase.

Criterion	Auto Threshold	Manual Threshold	Anomaly
Total False Positives	3	2	0
Short Attacks	Good coverage (80–93%)	Good coverage (80–93%)	Poor (13–27% coverage)
Long Attacks	Excellent (99% average)	Very Good (95–97%)	Good (91–95%)
Memory Usage	Lowest (100–153 MB)	Medium (107–166 MB)	Highest (169–178 MB)
Detection Delay	Fast (2–6s for UDP)	Slightly slower for some UDP	Slowest (20–30s) for short bursts
Attack End	Overextends 10–15s	Same or fragments extended	Sometimes underreports or splits
Precision	6/9 = 0.667	6/8 = 0.833	6/6 = 1.0
Recall	6/6 = 1.0	6/6 = 1.0	6/6 = 1.0
Accuracy	9/12 = 0.75	10/12 = 0.833	12/12 = 1.0
F1 Score	0.80	0.857	1.0

Performance Metrics Explained:

- **Precision** measures the fraction of detected attacks that were actual attacks (true positives divided by the sum of true positives and false positives). The anomaly detection achieved perfect precision (1.0) with no false positives, while auto-threshold (0.667) and manual threshold (0.75) were lower due to false alarms.
- **Recall** measures the fraction of actual attacks that were detected. All methods detected all six attack scenarios to some degree, resulting in a recall of 1.0, although with varying coverage within each attack.
- **Accuracy** represents the proportion of correct predictions (true positives and negatives) among the total cases. With six attacks and six non-attack periods, the anomaly method achieved perfect accuracy (12/12 = 1.0), while manual threshold (10/12 = 0.833) and auto-threshold (9/12 = 0.75) were lower due to false positives.

- **F1 Score** is the harmonic mean of precision and recall, providing a balanced measure of a detector's accuracy. The anomaly method's perfect precision yields an F1 score of 1.0, compared to auto-threshold (0.80) and manual threshold (0.85).
- **Coverage** represents the percentage of an attack's duration successfully detected. While not a standard metric, it provides insight into the completeness of detection. Despite perfect precision, the anomaly method's poor coverage of short attacks (13-27%) significantly impacts its practical effectiveness.
- **Temporal Accuracy** accounts for both missed detection time and overreported time. The threshold methods generally achieved better temporal accuracy for all but the shortest attacks.

This comparison highlights the fundamental trade-offs between the methods:

- Threshold approaches excel at quick detection and lower resource usage, but experience occasional false positives
- Anomaly detection eliminates false positives but requires more memory and often misses short attacks
- While manual threshold showed slightly better precision and F1 scores than auto-threshold, the Auto-Threshold Alarm generally provided better coverage for UDP floods due to its network-specific calibration

6. Discussion

6.1 Strengths and Weaknesses of Each Method

Auto-Threshold:

- **Strengths**:
  - Quick detection (2-6 seconds for most attacks)
  - Minimal resource usage (~100 MB baseline)
  - Network-specific adaptation through training
  - Particularly effective at detecting UDP floods (99% coverage)
  - Balanced compromise between speed and accuracy
- **Weaknesses**:
  - Produced one false positive during baseline testing
  - Can overextend short attack durations by 10-15 seconds
  - Struggled more with medium SYN floods (~63% coverage)
  - Requires a 12-hour training period for optimal performance

Manual Threshold:

- **Strengths**:
  - Predictable behaviour with fixed thresholds
  - No training period necessary for immediate deployment
  - Strong coverage for moderate/long floods (95-97%)
  - Similar memory profile to auto-threshold
- **Weaknesses**:

    - Also triggered one false positive during baseline
    - Less adaptable to different network environments
    - Can fragment longer attacks if thresholds are poorly tuned
    - Requires user knowledge to set appropriate thresholds

Anomaly Detection:

- **Strengths**:
    - Zero false positives during testing
    - Robust detection for longer floods (91-95% coverage)
    - Adaptive to network-specific standard traffic patterns
    - No manual threshold configuration needed
    - Highest precision and F1 score among all methods
- **Weaknesses**:
    - Significantly higher memory usage (169-178 MB)
    - Much slower detection (20-30s) for short attacks
    - Poor coverage for 30-second bursts (13-27%)
    - Requires a 12-hour training period for solid baseline establishment
    - More complex implementation with multiple parameters

6.2 Observed Trends and Practical Implications

My testing revealed several significant trends with practical implications for home network deployment:

Short Attack Challenge: The fundamentally different approaches to detection create a significant disparity in short attack performance. Anomaly detection often misses large portions of a 30-second flood because it waits for consecutive anomalous windows (three 10-second windows = 30 seconds) before triggering. Meanwhile, threshold-based detection identifies the attack almost immediately but typically overextends the end time by 10-15 seconds as residual packets clear the detection window.

Long Attack Strength: All methods perform well on floods lasting 5 minutes or longer, with coverage ranging from 90% to 99%. This suggests that any of these methods adequately detect persistent attacks. The choice then becomes more dependent on tolerance for false positives and resource constraints.

False Positive Considerations: The anomaly approach avoided false alarms during baseline testing. This advantage comes from its requirement for consecutive windows and its ability to learn normal traffic variations. This benefit might outweigh the reduced short-attack coverage for environments where false alarms are particularly disruptive.

Home-Network Recommendation: For typical home network deployments, the Auto-Threshold Alarm offers the best balance between speed, memory usage, and detection coverage. Its network-specific calibration provides better adaptation than manual thresholds, while its rapid detection significantly outperforms anomaly detection for short bursts. The occasional false positive (approximately 1 per hour in our testing) seems a reasonable trade-off for comprehensive protection.

6.3 Optimizations & Potential Improvements

Several optimizations could enhance the system's performance:

Adaptive Retraining: Periodically refine anomaly centroids and thresholds to capture evolving traffic patterns. This could be implemented as a daily or weekly background process to ensure the system adapts to changing network behaviour.

Dynamic Windows: Adjust threshold or anomaly detection intervals based on the time of day or known usage patterns. For example, larger windows during high-traffic periods and smaller windows during typically quiet times could reduce false positives while maintaining sensitivity.

Sub-Window Verification: Short attacks benefit from smaller consecutive anomaly windows (e.g., 2 instead of 3) or variable-sized windows based on traffic intensity. This could significantly improve the anomaly method's coverage of brief attacks.

Hybrid Detection Model: Combine auto-threshold for quick initial detection, then confirm with anomaly checks to reduce false positives while maintaining speed. This approach could leverage the strengths of both methods while mitigating their weaknesses.

Advanced ML Classifiers: Instead of K-means, consider implementing an autoencoder or small CNN to capture traffic distribution patterns precisely. The LUCID framework [1] demonstrates how even lightweight deep learning approaches can be practical while maintaining reasonable resource requirements.

Resource Optimization: Implement more efficient data structures and processing logic to reduce the memory footprint, particularly for the anomaly detection method. This could make the approach more viable for very resource-constrained devices.

7. Conclusion

In this project, I developed a lightweight DDoS alarm system with three detection modes: auto-threshold, manual threshold, and anomaly-based (K-means). The system was tested under baseline conditions and six distinct flood scenarios. The comprehensive evaluation revealed that each method offers unique advantages and limitations:

The threshold methods, particularly auto-threshold, detect short floods rapidly (2-6 seconds) with good coverage (80-93%). However, both threshold methods detected false-positive attacks during testing. Still, they provide excellent performance for longer attacks (95-99% coverage) with minimal resource requirements (~100 MB baseline memory).

The anomaly detection method produced zero false positives, demonstrating superior precision in distinguishing legitimate traffic spikes from attacks. However, it exhibited substantially slower detection, especially for short 30-second bursts (13-27% coverage), and consumed more memory (~170 MB baseline).

The Auto-Threshold Alarm is recommended as the most balanced solution for resource-constrained home networks where fast attack detection is crucial. Its network-specific calibration provides better adaptation than manual thresholds, while its quick detection time ensures minimal impact from attacks.

For environments where false alarms are unacceptable and longer attack detection is sufficient, the anomaly method justifies its higher memory usage and detection lag by eliminating false positives.

The project demonstrates that effective DDoS detection is possible in lightweight implementations suitable for home networks. There are clear trade-offs between detection speed, accuracy, and resource requirements that users can evaluate based on their specific priorities and needs.

8. Code Snippets Appendix

8.1 Threshold Class

```
class ThresholdDetector:
    def __init__(self, syn_threshold, udp_threshold):
        self.syn_threshold = syn_threshold
        self.udp_threshold = udp_threshold
        self.packet_counts = defaultdict(int)
        self.source_ips = set()

    def update(self, packet):
        if packet.haslayer(IP):
            self.source_ips.add(packet[IP].src)
        if packet.haslayer(TCP):
            tcp_layer = packet.getlayer(TCP)
            if tcp_layer.flags == 'S':  # SYN flag
                self.packet_counts['SYN'] += 1
        if packet.haslayer(UDP):
            self.packet_counts['UDP'] += 1

    def reset(self):
        self.packet_counts.clear()
        self.source_ips.clear()
```

8.2 Anomaly Class

```
class AnomalyDetector:
    """Anomaly detection using KMeans with auto-calibrated parameters."""

    def __init__(self, interface):
        self.interface = interface
        self.consecutive_anomalies = 0
        self.anomaly_threshold = 3  # Number of consecutive windows needed to
trigger alert (This can be changed by the user or could be updated dynamically
for different times of the day)
        self.pending_anomaly = None  # Store details of pending anomaly
        self.in_anomaly_state = False  # Track if we're currently in an anomaly
state
        # Load history from persistent storage
        self.history = load_traffic_data(interface)
        print(f"Loaded {len(self.history)} historical traffic data points for
{interface}")

        # Flag to indicate if we're in training mode
        self.training_mode = len(self.history) < 10

        # Store historical maximums to prevent false positives during quiet
periods
        self.historical_max_syn = 0
        self.historical_max_udp = 0
        if len(self.history) > 0:
            self.historical_max_syn = max([point[0] for point in self.history])
            self.historical_max_udp = max([point[1] for point in self.history])
            print(f"Historical max values - SYN: {self.historical_max_syn},
UDP: {self.historical_max_udp}")

        # Auto-determine parameters based on data
        self.calibrate_parameters()

        # Keep track of current window counts for display during training
        self.current_syn = 0
        self.current_udp = 0

        if self.training_mode:
            print(f"Training mode active - collecting baseline data
({len(self.history)}/10 windows)")
        else:
            print("Anomaly detection active")

    def calibrate_parameters(self):
        """Calculate appropriate n_clusters and distance_threshold from
data."""
        if len(self.history) < 5:  # Need some minimum data
            self.n_clusters = 1
            self.distance_threshold = 10.0  # Default fallback
            return
        # Convert to numpy array for calculations
        X = np.array(self.history)

        # Determine number of clusters using silhouette method or simple
heuristic
        if len(self.history) > 30:
            # Use a heuristic based on data size
            self.n_clusters = min(5, max(1, len(self.history) // 20))
        else:
            self.n_clusters = 1  # Default for small datasets

        # Fit K-means
        kmeans = KMeans(n_clusters=self.n_clusters, random_state=0)
        labels = kmeans.fit_predict(X)

        # Calculate distances to centroids for all points
        distances = []
        for i, point in enumerate(X):
            centroid = kmeans.cluster_centers_[labels[i]]
            dist = np.linalg.norm(point - centroid)
            distances.append(dist)

        # Set threshold based on distribution of distances (e.g., mean + 2*std
for better sensitivity)
        mean_dist = np.mean(distances)
        std_dist = np.std(distances)

        # For more sensitive detection (use 2 std instead of 3)
        # self.distance_threshold = mean_dist + 3 * std_dist

        # self.distance_threshold = max(100, mean_dist + 3 * std_dist)

        min_threshold = max(50, min(self.historical_max_syn,
self.historical_max_udp) * 0.1)
        self.distance_threshold = max(min_threshold, mean_dist + 3 * std_dist)

        print(f"Auto-calibrated parameters: n_clusters={self.n_clusters},
distance_threshold={self.distance_threshold:.2f}")
```

```
    def update_counts(self, packet):
        """Track SYN and UDP packets for the current window"""
        if packet.haslayer(TCP):
            tcp_layer = packet.getlayer(TCP)
            if tcp_layer.flags == 'S':  # SYN flag
                self.current_syn += 1
        if packet.haslayer(UDP):
            self.current_udp += 1

        # Display current counts during training
        if self.training_mode and (self.current_syn + self.current_udp) % 10 ==
0:  # Show every 10 packets
            print(f"Training window counts - SYN: {self.current_syn}, UDP:
{self.current_udp}")

    def update(self, syn_count, udp_count):
        # Reset current window counts
        self.current_syn = 0
        self.current_udp = 0

        # Check if we're still in training mode
        if self.training_mode:
            self.history.append([syn_count, udp_count])
            # Save to persistent storage after each update
            append_traffic_data(self.interface, syn_count, udp_count)
            print(f"Training: Window {len(self.history)}/10 - SYN={syn_count},
UDP={udp_count}")

            # Exit training mode if we have enough data
            if len(self.history) >= 10:
                self.training_mode = False
                self.calibrate_parameters()
                print("Training complete - anomaly detection now active")
            return False, None  # No anomaly detection during training

        # Check if this data point is an anomaly BEFORE adding it to history
        is_anomaly, anomaly_info = self.is_anomaly(syn_count, udp_count)

        # Only add non-anomalous data to the history
        if not is_anomaly:
            # Reset consecutive counter if this window is normal
            if self.consecutive_anomalies > 0:
                print(f"Resetting anomaly counter (was
{self.consecutive_anomalies})")

                # If we had pending anomalies that didn't reach the threshold,
                # add them to the history since they weren't true anomalies
                if self.pending_anomaly is not None and
self.consecutive_anomalies < self.anomaly_threshold:
                    print(f"Adding {self.consecutive_anomalies} pending
anomalies to history that didn't reach threshold")

                    # Need to track all potential anomaly data points
                    # First, add the one we stored explicitly
                    if "syn_count" in self.pending_anomaly and "udp_count" in
self.pending_anomaly:
                        pending_data = [self.pending_anomaly["syn_count"],
self.pending_anomaly["udp_count"]]
                        self.history.append(pending_data)
                        append_traffic_data(self.interface, pending_data[0],
pending_data[1])
                        print(f"Added first pending anomaly to history:
SYN={pending_data[0]}, UDP={pending_data[1]}")

                    # Now add the current window data (which would be the
second potential anomaly)
                    # Only do this if we had exactly 2 potential anomalies
                    if self.consecutive_anomalies == 2:
                        self.history.append([syn_count, udp_count])
                        append_traffic_data(self.interface, syn_count,
udp_count)
                        print(f"Added second pending anomaly to history:
SYN={syn_count}, UDP={udp_count}")

                self.consecutive_anomalies = 0
                self.pending_anomaly = None

            # If we were in an anomaly state and now it's over
            if self.in_anomaly_state:
                self.in_anomaly_state = False
                # Signal the end of an anomaly (for logging purposes)
                return False, {"anomaly_ended": True}


            # Update historical maximums for non-anomalous traffic
            if syn_count > self.historical_max_syn:
                self.historical_max_syn = syn_count
            if udp_count > self.historical_max_udp:
                self.historical_max_udp = udp_count

            # Add to history and save
            self.history.append([syn_count, udp_count])
            append_traffic_data(self.interface, syn_count, udp_count)

            # Limit history size to prevent memory issues
            if len(self.history) > 100:
                # Keep first 20% (oldest) and last 80% (newest) to maintain
perspective
                history_length = len(self.history)
                keep_historical = int(history_length * 0.2)
                keep_recent = history_length - int(history_length * 0.2)
                self.history = self.history[:keep_historical] +
self.history[-keep_recent:]

                # Update storage after trimming
                save_traffic_data(self.interface, self.history)
        else:
            self.consecutive_anomalies += 1
            print(f"Potential anomaly detected
({self.consecutive_anomalies}/{self.anomaly_threshold})")

            # Store the first anomaly's details if we don't have pending info
            if self.pending_anomaly is None:
                self.pending_anomaly = {
                    "first_seen": time.time(),
                    "syn_count": syn_count,
                    "udp_count": udp_count,
                    "info": anomaly_info
                }
            # Never add anomalous traffic to history, regardless of consecutive
count
            print(f"Potential anomaly detected - NOT adding to baseline:
SYN={syn_count}, UDP={udp_count}")

            # Check if we've reached the threshold for alerting
            if self.consecutive_anomalies >= self.anomaly_threshold and not
self.in_anomaly_state:
                self.in_anomaly_state = True
                print(f"Anomaly confirmed after {self.consecutive_anomalies}
consecutive windows")
                return True, anomaly_info
            elif self.consecutive_anomalies < self.anomaly_threshold:
                # Not enough consecutive anomalies yet to trigger an alert
                return False, None
            else:
                # We're in an ongoing anomaly state
                print(f"Ongoing anomaly - window {self.consecutive_anomalies}")
                return True, anomaly_info

        # Recalibrate periodically (every 20 points instead of 10)
        if len(self.history) % 20 == 0 and not is_anomaly:
            self.calibrate_parameters()

        return False, None

    def is_anomaly(self, syn_count, udp_count):
        """Check if the given counts represent an anomaly"""
        # First check: is traffic significantly above historical maximums?
        # Allow for up to 40% above historical maximum without triggering (more
sensitive)
        traffic_ratio = max(
            syn_count / max(1, self.historical_max_syn),
            udp_count / max(1, self.historical_max_udp)
        )

        # Print traffic statistics for debugging
        print(f"Traffic stats - SYN: {syn_count}, UDP: {udp_count}, "
              f"Historical max - SYN: {self.historical_max_syn}, UDP:
{self.historical_max_udp}")
```

```
        print(f"Traffic ratio: {traffic_ratio:.2f}")

        # Use a more sensitive threshold for high traffic
        if traffic_ratio > 2.0:  # 100% above historical instead of 40%
            anomaly_info = {
                "reason": "High traffic volume",
                "details": f"Traffic {int(traffic_ratio*100-100)}% above
historical maximum"
            }
            return True, anomaly_info

        if len(self.history) < self.n_clusters:
            return False, None

        # Get the point we're testing
        test_point = np.array([[syn_count, udp_count]])

        # Fit KMeans on existing history
        X = np.array(self.history)
        kmeans = KMeans(n_clusters=self.n_clusters, random_state=0)
        kmeans.fit(X)

        # Find closest centroid for the new point
        new_label = kmeans.predict(test_point)[0]
        centroid = kmeans.cluster_centers_[new_label]

        # Calculate distance to closest centroid
        distance = np.linalg.norm(test_point - centroid)

        print(f"Distance to nearest centroid: {distance:.2f}, Threshold:
{self.distance_threshold:.2f}")

        is_anomaly = distance > self.distance_threshold

        if is_anomaly:
            anomaly_info = {
                "reason": "Abnormal traffic pattern",
                "details": f"Distance {distance:.2f} > threshold
{self.distance_threshold:.2f}"
            }
            print(f"ANOMALY DETECTED - distance {distance:.2f} > threshold
{self.distance_threshold:.2f}")
            return True, anomaly_info

        return False, None
```

8.3 Attack Logs

```
def log_attack(attack_type, start_time=None, end_time=None, sources=None,
extra_info=None, detection_method=None):
    # Use the current session's metrics file
    global current_metrics_file

    with open(current_metrics_file, 'a') as f:
        now_str = time.strftime('%Y-%m-%d %H:%M:%S', time.localtime())
        f.write(f"Log Entry Time: {now_str}\n")

        if start_time is not None:
            start_str = time.strftime('%Y-%m-%d %H:%M:%S',
time.localtime(start_time))
            f.write(f"Start Time: {start_str}\n")

        if end_time is not None and start_time is not None:
            end_str = time.strftime('%Y-%m-%d %H:%M:%S',
time.localtime(end_time))
            duration = end_time - start_time
            f.write(f"End Time: {end_str}\n")
            f.write(f"Duration: {duration:.2f} seconds\n")

        if detection_method is not None:
            f.write(f"Detection Method: {detection_method}\n")

        if sources:
            if isinstance(sources, set):
                sources = list(sources)
            f.write(f"Sources: {', '.join(sources)}\n")

        if extra_info:
            f.write(f"Details: {extra_info}\n")
    f.write("------------------------------------------------------\n\n")
```

9. References

1. Doriguzzi-Corin, R., Millar, S., Scott-Hayward, S., Martinez-del-Rincon, J., & Siracusa, D. (2020). LUCID: A Practical, Lightweight Deep Learning Solution for DDoS Attack Detection. IEEE Transactions on Network and Service Management.

2. Singh, C., & Jain, A. K. (2024). A comprehensive survey on DDoS attacks detection & mitigation in SDN-IoT network. e-Prime: Advances in Electrical Engineering, Electronics and Energy, 8, 100543.

3. Rahmad, I. F., Ahmad, T., & Gata, W. (2021). DDoS Attack Detection Using Neural Network Based on K-means Clustering. Journal of Information Systems and Informatics, 3(2), 307-320.

4. Scapy Development Team, “Scapy Documentation,” https://scapy.net

5. F. Pedregosa et al., “Scikit-learn: Machine Learning in Python,” https://scikit-learn.org