Rapid-Corona: DDoS Mitigation System - Complete Project Explanation

Table of Contents

1. Project Overview
2. Core Technologies & Modules
3. Architecture & Design
4. Key Components Explained
5. eBPF/XDP Technology
6. Machine Learning Integration
7. Detection Mechanisms
8. Data Flow
9. File Structure
10. Dependencies

Project Overview

Rapid-Corona is a high-performance, machine learning-enhanced DDoS (Distributed Denial of Service) mitigation system designed to detect and prevent network attacks in real-time. The system operates at the kernel level using eBPF/XDP technology for ultra-fast packet filtering while employing sophisticated statistical and ML-based anomaly detection in user space.

Key Features

- **Ultra-fast packet processing**: 5M+ packets per second (pps) throughput
- **Kernel-level filtering**: Using eBPF/XDP for minimal latency
- **Dual detection approach**: Statistical baseline + ML classification
- **Real-time monitoring**: Web dashboard with live statistics
- **Cross-platform**: Linux (native eBPF) and Windows (Microsoft eBPF)
- **Attack classification**: Identifies specific attack types (SYN flood, UDP flood, etc.)

Development Phases

- **Phase 1**: Baseline statistical anomaly detection with eBPF/XDP
- **Phase 2**: ML-based classification using Random Forest trained on CIC-DDoS-2019 dataset
- **Phase 3** (Future): Auto signature generation and comparative benchmarking

---

Core Technologies & Modules

1. **eBPF (Extended Berkeley Packet Filter)**

What is eBPF? eBPF is a revolutionary Linux kernel technology that allows running sandboxed programs in kernel space without changing kernel source code or loading kernel modules. It provides safe, efficient, and programmable access to kernel events.

Why eBPF for DDoS Mitigation?

- **Performance**: Processes packets at the NIC driver level before they reach the network stack
- **Safety**: Verified by the kernel to ensure it won't crash the system
- **Efficiency**: Minimal CPU overhead even at millions of packets per second
- **Flexibility**: Can be updated without rebooting

How it works in this project:

```
// XDP program runs on every incoming packet
int xdp_ddos_filter(struct xdp_md *ctx) {
    // Parse packet headers
    // Check blacklist
    // Update statistics
    // Return XDP_PASS or XDP_DROP
}
```

2. **XDP (eXpress Data Path)**

What is XDP? XDP is a Linux kernel feature that enables eBPF programs to run at the earliest possible point in the networking stack - right when packets arrive at the network interface card (NIC).

XDP Actions:

- `XDP_PASS`: Allow packet to continue to network stack
- `XDP_DROP`: Drop packet immediately (fastest way to block)
- `XDP_TX`: Transmit packet back out the same interface
- `XDP_REDIRECT`: Redirect to another interface

XDP Modes in this project:

- **Native mode**: Runs in NIC driver (fastest, requires driver support)
- **Generic mode**: Runs in kernel network stack (slower but works everywhere)
- **Offload mode**: Runs on NIC hardware (requires smart NICs)

3. **BCC (BPF Compiler Collection)**

What is BCC? BCC is a toolkit for creating efficient kernel tracing and manipulation programs using eBPF. It provides Python bindings to write, compile, and interact with eBPF programs.

Usage in this project:

```
# Load eBPF program from C source
self.bpf = BPF(text=bpf_source)

# Attach XDP program to network interface
fn = self.bpf.load_func("xdp_ddos_filter", BPF.XDP)
self.bpf.attach_xdp(self.interface, fn, flags)
```

```
# Read statistics from eBPF maps
stats_map = self.bpf.get_table("stats_map")
```

4. **Machine Learning (scikit-learn)**

Random Forest Classifier: The project uses Random Forest, an ensemble learning method that constructs multiple decision trees during training and outputs the class that is the mode of the classes.

Why Random Forest?

- **Fast inference**: Critical for real-time detection (<10ms per prediction)
- **Robust**: Handles high-dimensional data well (64 features)
- **Interpretable**: Provides feature importance rankings
- **Accurate**: Achieves >95% accuracy on CIC-DDoS-2019 dataset

Training Process:

```
classifier = RandomForestClassifier(
    n_estimators=100,      # 100 decision trees
    max_depth=15,          # Limit tree depth for speed
    class_weight='balanced' # Handle imbalanced datasets
)
classifier.fit(X_train, y_train)
```

5. **CIC-DDoS-2019 Dataset**

What is CIC-DDoS-2019? A comprehensive DDoS attack dataset created by the Canadian Institute for Cybersecurity containing labeled network traffic flows with 64 statistical features.

Attack Types Included:

- DrDoS_DNS (DNS amplification)
- DrDoS_LDAP (LDAP amplification)
- DrDoS_MSSQL (MSSQL amplification)
- DrDoS_NTP (NTP amplification)
- DrDoS_UDP (Generic UDP reflection)
- Syn (SYN flood attacks)
- HTTP_Flood (Application layer attacks)
- BENIGN (Normal traffic)

Features Extracted (64 total):

- Flow duration and packet counts
- Packet length statistics (min, max, mean, std)
- Inter-arrival times (IAT)
- TCP flags (SYN, ACK, FIN, RST, PSH, URG)
- Bytes/packets per second
- Forward/backward flow ratios

6. **Flask Web Framework**

Purpose: Provides the real-time monitoring dashboard accessible via web browser.

Features:

- RESTful API endpoint (`/api/status`)
- Auto-refreshing dashboard (5-second intervals)
- Real-time metrics visualization
- Alert history display
- ML model statistics

7. **NumPy & SciPy**

NumPy:

- Efficient array operations for feature extraction
- Statistical calculations (mean, std, variance)
- Fast numerical computations

SciPy:

- Advanced statistical functions
- Signal processing for traffic analysis
- Scientific computing utilities

8. **psutil (Process and System Utilities)**

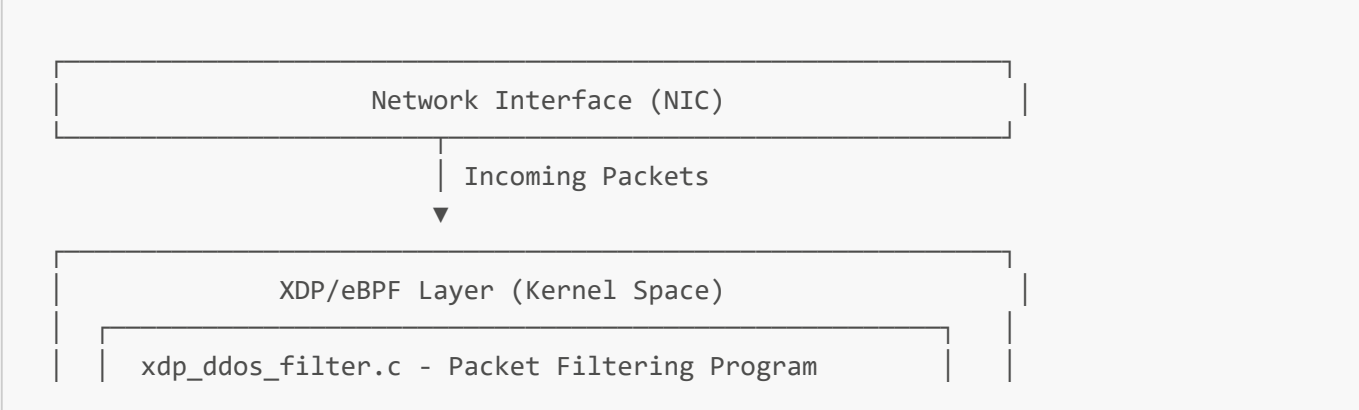
Purpose: Monitors system resources to ensure the DDoS mitigation system itself doesn't overload the server.

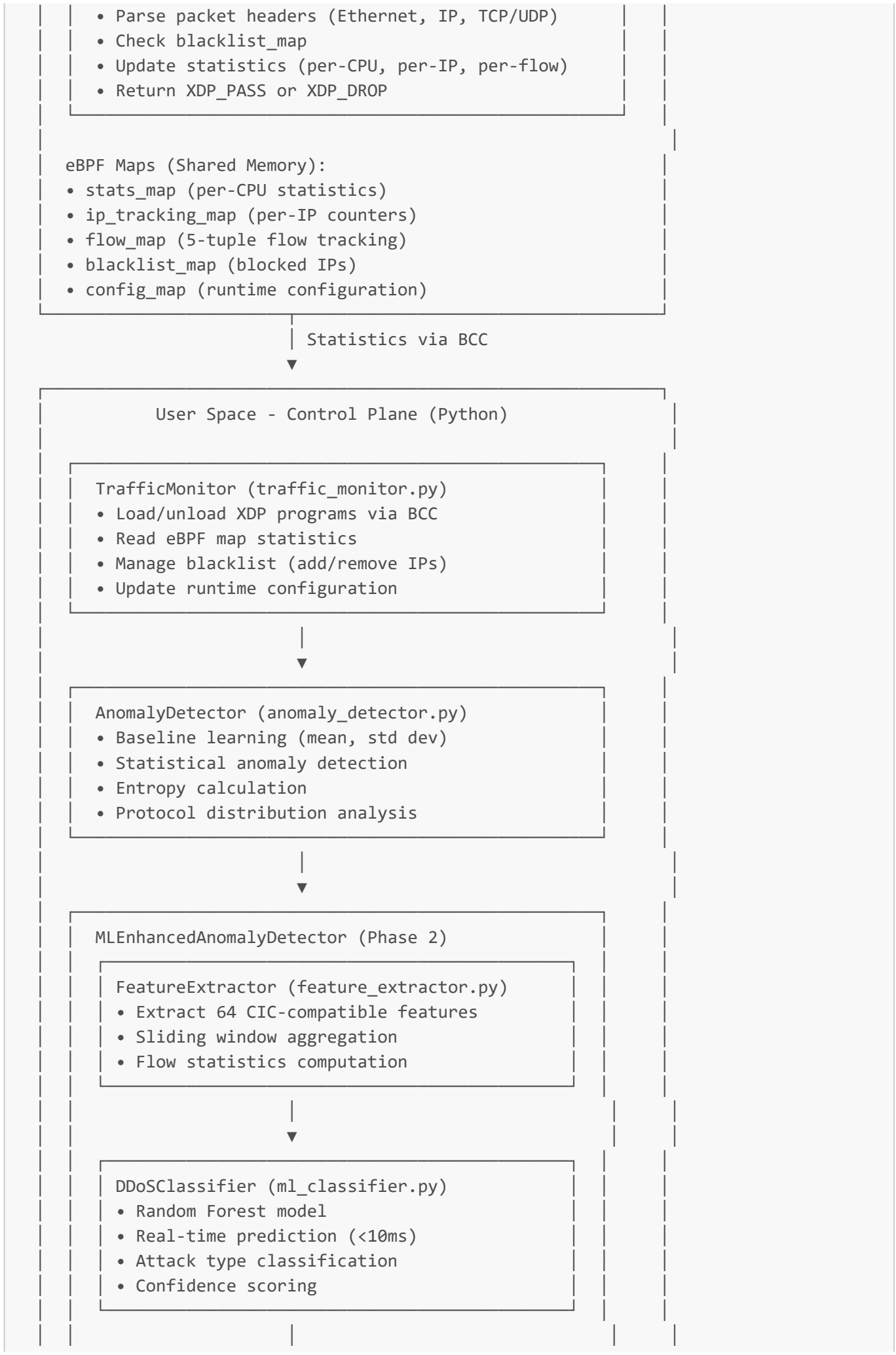
Metrics Collected:

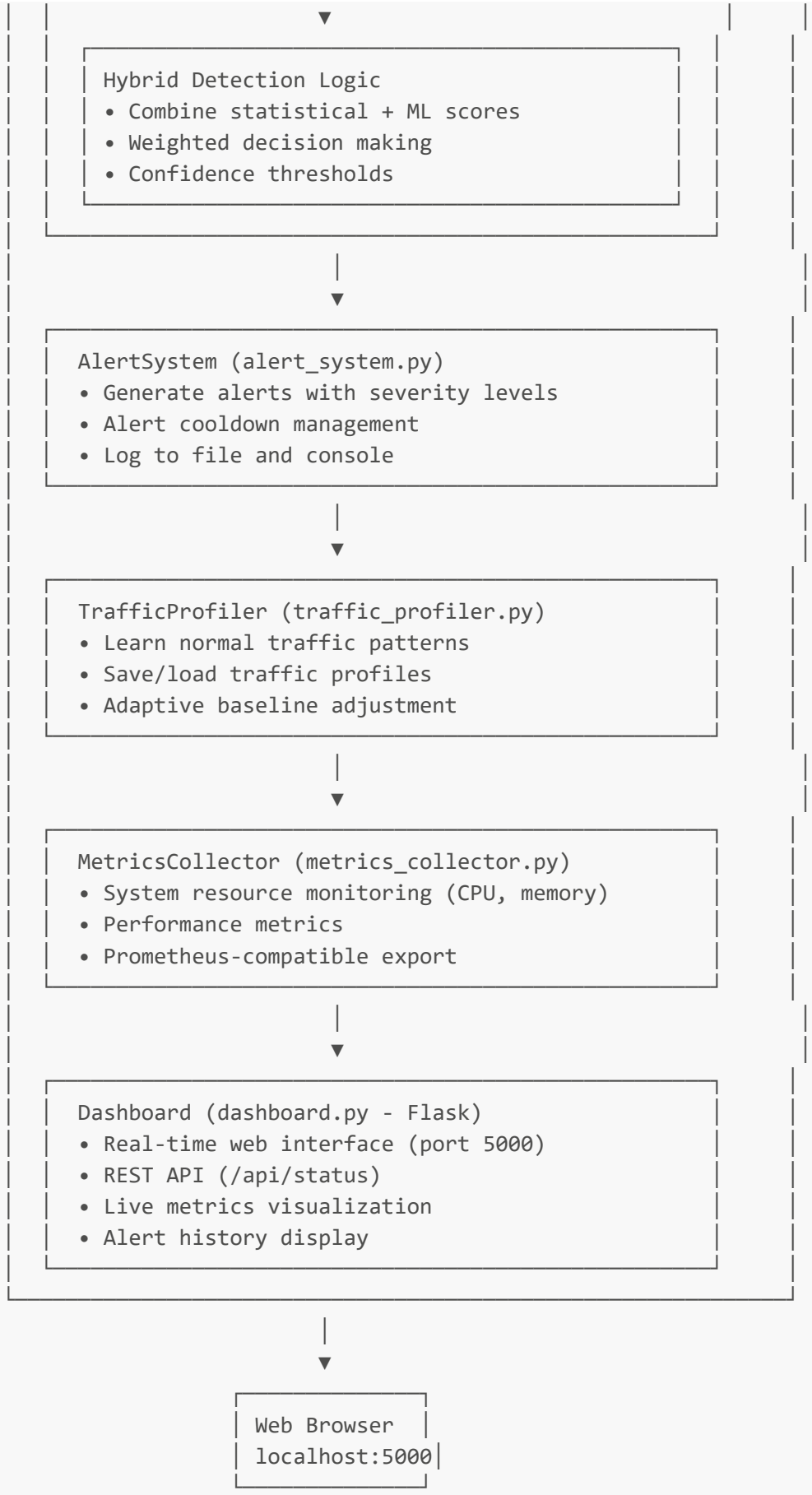
- CPU usage percentage
- Memory consumption
- Network interface statistics
- Process information

🏗️ Architecture & Design

System Architecture Diagram




• Parse packet headers (Ethernet, IP, TCP/UDP)
• Check blacklist_map
• Update statistics (per-CPU, per-IP, per-flow)
• Return XDP_PASS or XDP_DROP
eBPF Maps (Shared Memory):
• stats_map (per-CPU statistics)
• ip_tracking_map (per-IP counters)
• flow_map (5-tuple flow tracking)
• blacklist_map (blocked IPs)
• config_map (runtime configuration)
Statistics via BCC
User Space - Control Plane (Python)
TrafficMonitor (traffic_monitor.py)
• Load/unload XDP programs via BCC
• Read eBPF map statistics
• Manage blacklist (add/remove IPs)
• Update runtime configuration
AnomalyDetector (anomaly_detector.py)
• Baseline learning (mean, std dev)
• Statistical anomaly detection
• Entropy calculation
• Protocol distribution analysis
MLEnhancedAnomalyDetector (Phase 2)
FeatureExtractor (feature_extractor.py)
• Extract 64 CIC-compatible features
• Sliding window aggregation
• Flow statistics computation
DDoSClassifier (ml_classifier.py)
• Random Forest model
• Real-time prediction (<10ms)
• Attack type classification
• Confidence scoring



Data Plane vs Control Plane

Data Plane (eBPF/XDP - Kernel Space):

- **Purpose**: Fast packet processing and filtering
- **Performance**: 5M+ pps, <1µs per packet
- **Operations**: Parse, filter, count, drop
- **Language**: C (compiled to eBPF bytecode)

Control Plane (Python - User Space):

- **Purpose**: Decision making and management
- **Performance**: 1-second update intervals
- **Operations**: Analyze, detect, alert, configure
- **Language**: Python 3.8+

🔍 Key Components Explained

1. main.py - Application Orchestrator

Purpose: Main entry point that coordinates all components.

Key Responsibilities:

- Parse command-line arguments
- Initialize all subsystems
- Start monitoring loop
- Handle graceful shutdown
- Manage ML model training/loading

Main Loop Flow:

```
while running:
    1. Read statistics from eBPF maps
    2. Update baseline profile
    3. Detect anomalies (statistical + ML)
    4. Send alerts if attack detected
    5. Update blacklist if needed
    6. Collect system metrics
    7. Sleep for 1 second
```

2. config.py - Configuration Management

Purpose: Centralized configuration for all system parameters.

Key Configuration Classes:

DetectionThresholds:

```
ALERT_PPS_THRESHOLD = 500      # Trigger alert
ATTACK_PPS_THRESHOLD = 2000    # Definite attack
```

```
SIGMA_MULTIPLIER = 2.0          # Statistical deviation
MIN_ENTROPY = 3.0               # IP diversity threshold
```

TimeWindows:

```
BASELINE_WINDOW = 300   # 5 min baseline learning
DETECTION_WINDOW = 10   # 10 sec detection window
ALERT_COOLDOWN = 60     # 60 sec between duplicate alerts
```

MLConfig:

```
MODEL_TYPE = 'random_forest'
N_ESTIMATORS = 100
MAX_DEPTH = 15
ML_CONFIDENCE_THRESHOLD = 70.0
```

3. src/ebpf/xdp_filter.c - Kernel Packet Filter

Purpose: High-performance packet filtering in kernel space.

Data Structures:

flow_key (5-tuple for flow identification):

```
struct flow_key {
    __u32 src_ip;       // Source IP address
    __u32 dst_ip;       // Destination IP address
    __u16 src_port;     // Source port
    __u16 dst_port;     // Destination port
    __u8 protocol;      // Protocol (TCP/UDP/ICMP)
};
```

ip_stats (per-IP tracking):

```
struct ip_stats {
    __u64 packets;      // Total packets from this IP
    __u64 bytes;        // Total bytes from this IP
    __u64 last_seen;    // Timestamp of last packet
    __u32 flow_count;   // Number of flows
    __u32 syn_count;    // SYN packets (SYN flood detection)
    __u32 udp_count;    // UDP packets
};
```

eBPF Maps:

- `flow_map`: Tracks individual network flows (65,536 entries)
- `ip_tracking_map`: Per-IP statistics (131,072 entries)
- `blacklist_map`: Blocked IP addresses (10,000 entries)
- `stats_map`: Per-CPU global statistics
- `config_map`: Runtime configuration

Packet Processing Logic:

```
1. Parse Ethernet header → Check if IPv4
2. Parse IP header → Extract src_ip, dst_ip, protocol
3. Check blacklist → If blacklisted, return XDP_DROP
4. Parse TCP/UDP header → Extract ports and flags
5. Update ip_tracking_map with packet/byte counts
6. Update flow_map with flow statistics
7. Simple SYN flood check → Drop if syn_count > 1000
8. Update global statistics
9. Return XDP_PASS (allow packet)
```

4. src/traffic_monitor.py - eBPF Controller

Purpose: User-space interface to eBPF programs.

Key Methods:

load_xdp_program():

```
# Compile and load eBPF C code
self.bpf = BPF(text=bpf_source)
fn = self.bpf.load_func("xdp_ddos_filter", BPF.XDP)
self.bpf.attach_xdp(interface, fn, flags)
```

get_statistics():

```
# Read per-CPU stats and aggregate
stats_map = self.bpf.get_table("stats_map")
for cpu_stats in per_cpu_stats:
    total_packets += cpu_stats.total_packets
    total_bytes += cpu_stats.total_bytes
```

add_to_blacklist():

```
# Convert IP to integer and add to blacklist map
ip_int = struct.unpack('I', socket.inet_aton(ip_address))[0]
blacklist_map[ip_int] = timestamp
```

5. src/anomaly_detector.py - Statistical Detection

Purpose: Detect anomalies using statistical methods.

Detection Techniques:

1. Absolute Thresholds:

```
if pps > ATTACK_PPS_THRESHOLD:
    score += 50  # Very high traffic
```

2. Statistical Deviation (Z-score):

```
pps_sigma = (current_pps - baseline_mean) / baseline_std
if abs(pps_sigma) > SIGMA_MULTIPLIER:
    score += 30  # Significant deviation
```

3. Rate of Change:

```
change_rate = current_pps / previous_pps
if change_rate > MAX_CHANGE_RATE:
    score += 20  # Sudden spike
```

4. Protocol Distribution:

```
if abs(tcp_ratio - NORMAL_TCP_RATIO) > DEVIATION_THRESHOLD:
    score += 15  # Abnormal protocol mix
```

5. IP Entropy (Shannon Entropy):

```
entropy = -Σ(p_i * log2(p_i))
if entropy < MIN_ENTROPY:
    score += 20  # Traffic concentrated in few IPs
```

6. SYN Flood Detection:

```
syn_heavy_ips = [ip for ip in ip_stats if ip.syn_count > 500]
if syn_heavy_ips:
    score += 25  # Excessive SYN packets
```

Scoring System:

- Score >= 50: Anomaly detected
- Score >= 75: High severity
- Score < 50: Normal traffic

6. src/ml/feature_extractor.py - Real-time Feature Extraction

Purpose: Convert raw traffic statistics to ML-compatible features.

Feature Categories (64 total):

Flow Duration & Counts (5 features):

- Flow Duration (microseconds)
- Total Forward Packets
- Total Backward Packets
- Total Forward Bytes
- Total Backward Bytes

Packet Length Statistics (8 features):

- Forward: Max, Min, Mean, Std
- Backward: Max, Min, Mean, Std

Flow Rates (2 features):

- Flow Bytes/s
- Flow Packets/s

Inter-Arrival Times (14 features):

- Flow IAT: Mean, Std, Max, Min
- Forward IAT: Total, Mean, Std, Max, Min
- Backward IAT: Total, Mean, Std, Max, Min

TCP Flags (8 features):

- FIN, SYN, RST, PSH, ACK, URG, CWE, ECE counts

Additional Metrics (27 features):

- Packet length variance
- Down/Up ratio
- Average packet sizes
- Header lengths
- Active/Idle times

Sliding Window Approach:

```
# Maintain deques for efficient windowing
packets_fwd: deque(maxlen=1000)
packets_bwd: deque(maxlen=1000)
timestamps: deque(maxlen=1000)
```

```
# Calculate statistics over window
mean = np.mean(packets_fwd)
std = np.std(packets_fwd)
```

7. src/ml/ml_classifier.py - ML Prediction Engine

Purpose: Real-time attack classification using Random Forest.

Model Architecture:

```
RandomForestClassifier(
    n_estimators=100,        # 100 decision trees
    max_depth=15,            # Limit depth for speed
    min_samples_split=5,     # Prevent overfitting
    min_samples_leaf=2,      # Minimum leaf size
    class_weight='balanced', # Handle imbalanced data
    n_jobs=-1                # Use all CPU cores
)
```

Prediction Process:

```
1. Extract 64 features from traffic
2. Scale features using StandardScaler
3. Pass through Random Forest
4. Get probability scores for each class
5. Determine attack type and confidence
6. Return PredictionResult
```

Attack Type Inference:

```
if syn_count > 100 and ack_count < 10:
    return 'SYN_Flood'
elif packets_per_sec > 10000:
    return 'UDP_Flood'
else:
    return 'DDoS_Generic'
```

Performance Metrics:

- Inference time: <10ms per prediction
- Accuracy: >95% on test set
- Precision/Recall: >0.93

8. src/ml/data_loader.py - Dataset Management

Purpose: Load and preprocess CIC-DDoS-2019 dataset.

Key Functions:

Data Loading:

```
# Load CSV files with chunking for memory efficiency
for chunk in pd.read_csv(file, chunksize=10000):
    # Sample rows
    # Clean data (remove NaN, infinity)
    # Balance classes
```

Preprocessing:

```
# Remove infinite values
df.replace([np.inf, -np.inf], np.nan, inplace=True)

# Fill NaN with 0
df.fillna(0, inplace=True)

# Encode labels (BENIGN=0, Attack=1)
y = (y != 'BENIGN').astype(int)
```

Train/Val/Test Split:

```
# 70% train, 10% validation, 20% test
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.67)
```

Feature Scaling:

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)
X_test_scaled = scaler.transform(X_test)
```

9. src/dashboard.py - Web Interface

Purpose: Real-time monitoring dashboard.

Technology Stack:

- **Backend**: Flask (Python web framework)
- **Frontend**: Vanilla JavaScript + HTML/CSS
- **Styling**: Glassmorphism design with gradient backgrounds

API Endpoint:

```
@app.route('/api/status')
def api_status():
    return jsonify({
        'running': True,
        'interface': 'eth0',
        'statistics': {...},
        'baseline': {...},
        'ip_stats': [...],
        'blacklist': [...],
        'recent_alerts': [...],
        'ml_enabled': True,
        'ml_stats': {...}
    })
```

Dashboard Cards:

1. Real-time Status (interface, current PPS, baseline)
2. Traffic Statistics (packets, bytes, dropped)
3. Protocol Distribution (TCP, UDP, ICMP)
4. Baseline Profile (mean, std, samples)
5. Top IPs (highest traffic sources)
6. Blacklist (blocked IPs)
7. ML Classification (model accuracy, predictions)
8. Feature Importance (top ML features)
9. Recent Alerts (alert history)

Auto-refresh:

```
// Fetch data every 5 seconds
setInterval(fetchData, 5000);
```

10. src/alert_system.py - Alert Management

Purpose: Generate and manage security alerts.

Alert Severity Levels:

- **High**: Score >= 75, immediate action required
- **Medium**: Score >= 50, investigation needed
- **Low**: Score < 50, informational

Alert Structure:

```
{
    'timestamp': '2026-01-12 15:30:00',
```

```
    'alert_type': 'ddos_attack',
    'severity': 'high',
    'message': 'DDoS attack detected (score: 85.0)',
    'details': {
        'reasons': [...],
        'metrics': {...},
        'current_pps': 15000,
        'baseline_pps': 500
    }
}
```

Cooldown Mechanism:

```
# Prevent alert spam - 60 seconds between duplicate alerts
if current_time - last_alert_time >= ALERT_COOLDOWN:
    send_alert()
```

11. src/traffic_profiler.py - Traffic Profiling

Purpose: Learn and maintain normal traffic patterns.

Profile Data:

```
{
    'mean_pps': 500.0,
    'peak_pps': 2000.0,
    'mean_bps': 5000000.0,
    'protocol_distribution': {
        'tcp': 0.85,
        'udp': 0.10,
        'icmp': 0.05
    },
    'typical_sources': 50,
    'learning_period': 3600,
    'last_updated': '2026-01-12 15:00:00'
}
```

Adaptive Learning:

```
# Exponential moving average for gradual adaptation
new_mean = alpha * current_value + (1 - alpha) * old_mean
```

12. src/metrics_collector.py - System Monitoring

Purpose: Monitor system resource usage.

Metrics Collected:

```
{
    'cpu_percent': 15.5,
    'memory_percent': 25.3,
    'memory_used_mb': 512,
    'network_bytes_sent': 1000000,
    'network_bytes_recv': 5000000,
    'process_cpu_percent': 5.2,
    'process_memory_mb': 128
}
```

Purpose: Ensure the mitigation system doesn't become a performance bottleneck.

🚀 eBPF/XDP Technology

Why eBPF/XDP for DDoS Mitigation?

Traditional Approach Problems:

1. Packets traverse entire network stack
2. Context switches to user space
3. High CPU usage at scale
4. Latency increases under attack

eBPF/XDP Advantages:

1. **Kernel-level processing**: No network stack overhead
2. **Zero-copy**: Direct packet access in NIC driver
3. **Programmable**: Update logic without kernel recompilation
4. **Safe**: Verified by kernel verifier
5. **Fast**: 5M+ pps on commodity hardware

eBPF Map Types Used

1. BPF_HASH (Hash Table):

```
BPF_HASH(ip_tracking_map, __u32, struct ip_stats, 131072);
```

- **Purpose**: Key-value storage
- **Use case**: Per-IP statistics, blacklist
- **Performance**: O(1) lookup

2. BPF_PERCPU_ARRAY (Per-CPU Array):

```
BPF_PERCPU_ARRAY(stats_map, struct stats, 1);
```

- **Purpose**: Per-CPU counters (no locking needed)
- **Use case**: Global statistics
- **Performance**: Lock-free, cache-friendly

3. BPF_ARRAY (Array):

```
BPF_ARRAY(config_map, struct config, 1);
```

- **Purpose**: Simple indexed storage
- **Use case**: Configuration parameters
- **Performance**: Fastest lookup

XDP Program Verification

Kernel Verifier Checks:

1. **Bounded loops**: No infinite loops allowed
2. **Memory safety**: All pointer accesses validated
3. **Instruction limit**: Max 4096 instructions (older kernels)
4. **Stack size**: Limited to 512 bytes
5. **No kernel crashes**: Guaranteed safe execution

Performance Optimization Techniques

1. Per-CPU Statistics:

```
// Avoid lock contention by using per-CPU maps
BPF_PERCPU_ARRAY(stats_map, struct stats, 1);
```

2. Atomic Operations:

```
// Lock-free counter updates
__sync_fetch_and_add(&stats->total_packets, 1);
```

3. Early Drop:

```
// Check blacklist before expensive processing
if (is_blacklisted(src_ip)) {
    return XDP_DROP;  // Immediate drop
}
```

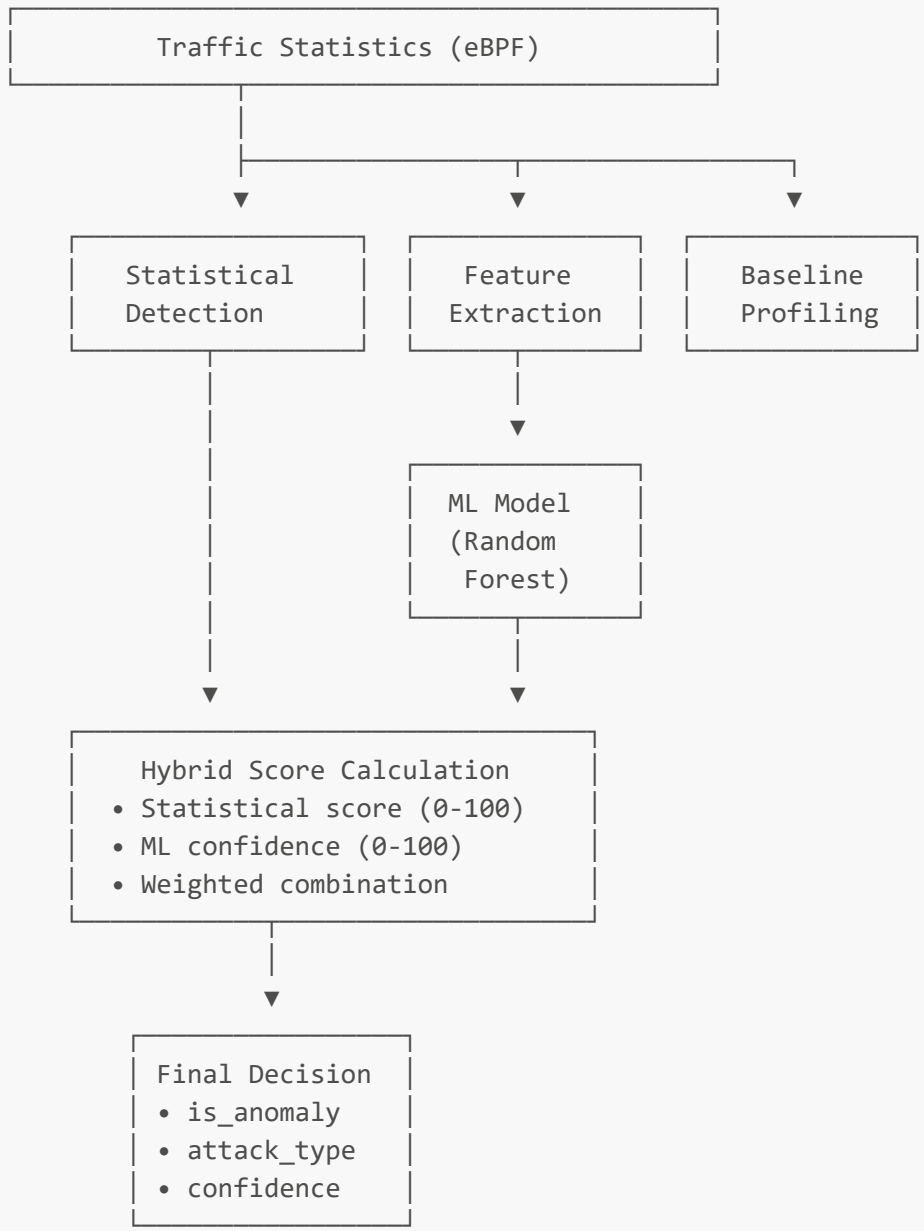
4. Bounds Checking:

```
// Prevent verifier rejection
if (data + sizeof(*eth) > data_end)
    return XDP_DROP;
```

Machine Learning Integration

Phase 2: ML-Enhanced Detection

Hybrid Detection Strategy:



Hybrid Scoring Logic

Decision Matrix:

Statistical Score	ML Confidence	Result	Source
>= 70	>= 85	Attack	Hybrid (high confidence)
>= 70	< 70	Attack	Statistical
< 50	>= 85	Attack	ML
>= 50	>= 70	Attack	Hybrid
< 50	< 70	Benign	Both agree

Combined Score Calculation:

```
combined_score = statistical_score + (ml_confidence / 100) * 30

if combined_score >= 60:
    is_anomaly = True
```

Feature Importance Analysis

Top 10 Most Important Features (typical):

1. Flow Bytes/s (15.2%)
2. Flow Packets/s (12.8%)
3. SYN Flag Count (9.5%)
4. Flow Duration (7.3%)
5. Fwd IAT Mean (6.1%)
6. Total Fwd Packets (5.8%)
7. Packet Length Mean (5.2%)
8. ACK Flag Count (4.9%)
9. Down/Up Ratio (4.3%)
10. Bwd Packet Length Mean (3.7%)

Interpretation:

- **Rate features** (bytes/s, packets/s) are most discriminative
- **TCP flags** (SYN, ACK) crucial for SYN flood detection
- **Timing features** (IAT) help identify attack patterns
- **Size features** distinguish attack types

Model Training Process

1. Data Preparation:

```
# Load CIC-DDoS-2019 dataset
loader = CICDataLoader('data/cic-ddos-2019/')
data = loader.prepare_data(
    max_files=5,
```

```
    samples_per_file=50000,
    binary=True,  # BENIGN vs ATTACK
    scale=True    # StandardScaler
)
```

2. Model Training:

```
classifier = DDoSClassifier(
    model_type='random_forest',
    n_estimators=100,
    max_depth=15
)

metrics = classifier.train(
    X_train=data['X_train'],
    y_train=data['y_train'],
    X_val=data['X_val'],
    y_val=data['y_val']
)
```

3. Model Evaluation:

```
# Test set evaluation
y_pred = classifier.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
```

4. Model Saving:

```
classifier.save('data/models/ddos_classifier.joblib')
```

Real-time Inference

Inference Pipeline:

```
1. FeatureExtractor.update(stats, ip_stats)
   ↓
2. features = FeatureExtractor.extract_features()
   ↓
3. scaled_features = scaler.transform(features)
   ↓
4. prediction = classifier.predict(scaled_features)
   ↓
```

```
5. PredictionResult(
       is_attack=True,
       attack_type='SYN_Flood',
       confidence=92.5,
       inference_time_ms=8.3
   )
```

Performance Requirements:

- Inference time: <10ms
- Throughput: 100+ predictions/second
- Memory: <100MB for model

📊 Detection Mechanisms

Multi-Layer Detection Approach

Layer 1: eBPF/XDP (Kernel) - Immediate Actions

- Blacklist enforcement (instant drop)
- Simple SYN flood detection (syn_count > 1000)
- Packet counting and statistics

Layer 2: Statistical Analysis (User Space) - Pattern Detection

- Baseline deviation (Z-score)
- Rate of change monitoring
- Protocol distribution analysis
- IP entropy calculation
- Heavy hitter detection

Layer 3: ML Classification (User Space) - Intelligent Classification

- 64-feature analysis
- Attack type identification
- Confidence scoring
- False positive reduction

Detection Scenarios

Scenario 1: Large UDP Flood

```
Traffic Pattern:
- Sudden spike: 100 pps → 50,000 pps
- Protocol: 95% UDP (normal: 10%)
- Source IPs: 1,000 unique IPs

Detection:
1. Statistical: High PPS (score: 50) + Protocol anomaly (score: 15) = 65
```

```
2. ML: Detects DrDoS_UDP pattern (confidence: 88%)
3. Hybrid: Attack confirmed (combined score: 91)

Action:
- Alert: HIGH severity
- Blacklist: Top 10 source IPs
- Drop rate: 45,000 pps
```

Scenario 2: SYN Flood

```
Traffic Pattern:
- Gradual increase: 500 pps → 5,000 pps
- Protocol: 100% TCP
- SYN packets: 4,800/sec, ACK packets: 200/sec
- Source IPs: 50 unique IPs (low entropy)

Detection:
1. Statistical: SYN heavy IPs (score: 25) + Low entropy (score: 20) = 45
2. ML: Detects SYN_Flood pattern (confidence: 95%)
3. Hybrid: Attack confirmed (combined score: 74)

Action:
- Alert: HIGH severity
- Blacklist: All 50 source IPs
- XDP drops: 4,500 pps
```

Scenario 3: Flash Crowd (False Positive Prevention)

```
Traffic Pattern:
- Sudden spike: 200 pps → 10,000 pps
- Protocol: 85% TCP (normal)
- Source IPs: 5,000 unique IPs (high entropy)
- Legitimate user behavior

Detection:
1. Statistical: High PPS (score: 50) + High entropy (score: -10) = 40
2. ML: Detects BENIGN pattern (confidence: 78%)
3. Hybrid: No attack (combined score: 32)

Action:
- No alert
- No blacklisting
- Continue monitoring
```

Alert Severity Calculation

Severity Levels:

```
if combined_score >= 85:
    severity = 'critical'
elif combined_score >= 75:
    severity = 'high'
elif combined_score >= 60:
    severity = 'medium'
else:
    severity = 'low'
```

Data Flow

Complete Request Flow

```
1. Packet Arrival at NIC
   ↓
2. XDP Hook Triggered
   ↓
3. xdp_ddos_filter() executes
   │
   ├→ Parse headers (Ethernet, IP, TCP/UDP)
   ├→ Check blacklist_map
   │   └→ If blacklisted: return XDP_DROP
   ├→ Update ip_tracking_map
   ├→ Update flow_map
   ├→ Update stats_map (per-CPU)
   ├→ Check SYN flood threshold
   │   └→ If exceeded: return XDP_DROP
   └→ return XDP_PASS
   ↓
4. Packet continues to network stack (if XDP_PASS)
   ↓
5. User Space Monitoring Loop (every 1 second)
   │
   ├→ TrafficMonitor.get_statistics()
   │   └→ Read eBPF maps via BCC
   │
   ├→ AnomalyDetector.update_baseline()
   │   └→ Calculate running mean/std
   │
   ├→ TrafficProfiler.update_profile()
   │   └→ Learn normal patterns
   │
   ├→ FeatureExtractor.update()
   │   └→ Aggregate features for ML
   │
   ├→ AnomalyDetector.detect_anomaly()
   │   ├→ Statistical analysis
   │   └→ ML prediction (if enabled)
   │       └→ Hybrid scoring
```

```
    │
    ├─→ If anomaly detected:
    │   ├─→ AlertSystem.send_alert()
    │   └─→ TrafficMonitor.add_to_blacklist()
    │       └─→ Update blacklist_map in eBPF
    │
    └─→ MetricsCollector.collect_system_metrics()
    ↓
6. Dashboard Update (every 5 seconds)
    │
    └─→ GET /api/status
        └─→ Return JSON with all metrics
    ↓
7. Browser displays updated dashboard
```

eBPF Map Update Flow

```
User Space (Python)                Kernel Space (eBPF)
___________________                ___________________

add_to_blacklist(ip)
    │
    ├─→ Convert IP to int
    │
    └─→ blacklist_map[ip_int] = timestamp
                                       │
                                       ↓
                              blacklist_map updated
                                       │
                                       ↓
                              Next packet from IP
                                       │
                                       ↓
                              is_blacklisted(ip)
                                       │
                                       ├─→ Found in map
                                       │
                                       └─→ return XDP_DROP
```

File Structure

```
rapid-corona/
│
├── main.py                          # Main application entry point
├── config.py                        # Configuration management
├── requirements.txt                 # Python dependencies
├── attack_simulator.py              # Traffic simulation tool
├── README.md                        # Project overview
```

```
├── USAGE_GUIDE.md                  # Detailed usage instructions
├── projectexplained.md             # This file
│
├── setup_ebpf.sh                   # Linux setup script
├── setup_ebpf.ps1                  # Windows setup script
│
├── src/                            # Source code directory
│   │
│   ├── __init__.py
│   │
│   ├── traffic_monitor.py          # eBPF/XDP controller
│   │   • Load/unload XDP programs
│   │   • Read eBPF map statistics
│   │   • Manage blacklist
│   │   • Runtime configuration
│   │
│   ├── anomaly_detector.py         # Statistical + ML detection
│   │   • Baseline learning
│   │   • Statistical anomaly detection
│   │   • ML-enhanced detection (Phase 2)
│   │   • Hybrid scoring
│   │
│   ├── traffic_profiler.py         # Traffic profiling
│   │   • Learn normal patterns
│   │   • Adaptive baseline
│   │   • Profile persistence
│   │
│   ├── alert_system.py             # Alert management
│   │   • Alert generation
│   │   • Severity classification
│   │   • Cooldown management
│   │   • Multi-channel alerting
│   │
│   ├── metrics_collector.py        # System monitoring
│   │   • CPU/memory tracking
│   │   • Performance metrics
│   │   • Prometheus export
│   │
│   ├── dashboard.py                # Web dashboard (Flask)
│   │   • Real-time UI
│   │   • REST API
│   │   • Metrics visualization
│   │
│   ├── ebpf/                       # eBPF programs
│   │   ├── xdp_filter.c            # Main XDP program (BCC style)
│   │   ├── xdp_filter_libbpf.c     # Alternative libbpf version
│   │   ├── xdp_maps.h              # eBPF map definitions
│   │   ├── Makefile                # Compilation rules
│   │   └── xdp_filter.o            # Compiled eBPF object
│   │
│   └── ml/                         # Machine Learning module (Phase 2)
│       ├── __init__.py
│       │
│       ├── ml_classifier.py        # Random Forest classifier
```

```
│       │   • Model training
│       │   • Real-time prediction
│       │   • Attack type classification
│       │   • Feature importance
│       │
│       ├── feature_extractor.py     # Feature extraction
│       │   • 64 CIC-compatible features
│       │   • Sliding window aggregation
│       │   • Real-time computation
│       │
│       ├── data_loader.py           # Dataset management
│       │   • CIC-DDoS-2019 loading
│       │   • Preprocessing pipeline
│       │   • Train/val/test split
│       │   • Feature scaling
│       │
│       └── model_trainer.py         # Training CLI
│           • Model training workflow
│           • Hyperparameter tuning
│           • Model evaluation
│           • Benchmarking
│
├── simulation/                      # Traffic simulation
│   ├── __init__.py
│   ├── traffic_simulator.py         # Traffic generation
│   └── attack_scenarios.py          # Predefined attack patterns
│
├── tests/                           # Unit tests
│   ├── test_detector.py             # Anomaly detector tests
│   ├── test_simulator.py            # Simulator tests
│   ├── test_ml_classifier.py        # ML classifier tests
│   ├── test_feature_extractor.py    # Feature extraction tests
│   ├── test_integration.py          # Integration tests
│   ├── benchmark.py                 # Performance benchmarks
│   └── conftest.py                  # Pytest configuration
│
├── data/                            # Data directory
│   ├── models/                      # Trained ML models
│   │   └── ddos_classifier.joblib   # Saved Random Forest model
│   │
│   ├── cic-ddos-2019/               # Dataset (user-provided)
│   │   ├── DrDoS_DNS.csv
│   │   ├── DrDoS_LDAP.csv
│   │   ├── DrDoS_MSSQL.csv
│   │   ├── DrDoS_NTP.csv
│   │   ├── DrDoS_UDP.csv
│   │   ├── Syn.csv
│   │   └── BENIGN.csv
│   │
│   └── traffic_profile.json         # Learned traffic baseline
│
└── logs/                            # Log files
    ├── ddos_mitigation.log          # Main application log
    └── alerts.log                   # Alert history
```

Dependencies

Python Packages (requirements.txt)

Core Dependencies:

```
numpy>=1.21.0              # Array operations, numerical computing
scipy>=1.7.0               # Scientific computing, statistics
pandas>=1.3.0              # Data manipulation, CSV loading
```

System Monitoring:

```
psutil>=5.8.0              # CPU, memory, network monitoring
```

Web Dashboard:

```
flask>=2.0.0               # Web framework
flask-cors>=3.0.10         # Cross-origin resource sharing
```

Machine Learning (Phase 2):

```
scikit-learn>=1.0.0        # Random Forest, preprocessing
joblib>=1.1.0              # Model serialization
```

Optional ML Accelerators:

```
# xgboost>=1.6.0           # Gradient boosting (faster training)
# lightgbm>=3.3.0          # Light gradient boosting
```

Testing:

```
pytest>=7.0.0              # Testing framework
pytest-cov>=3.0.0          # Code coverage
pytest-timeout>=2.1.0      # Test timeouts
```

Utilities:

```
coloredlogs>=15.0          # Colored console output
python-dotenv>=0.19.0      # Environment variable management
```

System Dependencies

Linux (Ubuntu/Debian):

```
# eBPF/BCC
sudo apt-get install python3-bpfcc
sudo apt-get install bpfcc-tools
sudo apt-get install linux-headers-$(uname -r)

# Build tools
sudo apt-get install build-essential
sudo apt-get install clang
sudo apt-get install llvm

# Python
sudo apt-get install python3-dev
sudo apt-get install python3-pip
```

Linux (RHEL/CentOS):

```
# eBPF/BCC
sudo yum install python3-bcc
sudo yum install bcc-tools
sudo yum install kernel-devel

# Build tools
sudo yum install gcc
sudo yum install clang
sudo yum install llvm
```

Windows:

```
# Microsoft eBPF for Windows
# Download from: https://github.com/microsoft/ebpf-for-windows

# Chocolatey (package manager)
Set-ExecutionPolicy Bypass -Scope Process -Force
iex ((New-Object
System.Net.WebClient).DownloadString('https://chocolatey.org/install.ps1'))

# Build tools
choco install -y llvm
```

```
choco install -y python3
choco install -y git
```

Kernel Requirements

Linux:

- Kernel version: 4.18+ (for XDP support)
- Kernel version: 5.10+ (recommended for latest eBPF features)
- CONFIG_BPF=y
- CONFIG_BPF_SYSCALL=y
- CONFIG_XDP_SOCKETS=y

Check kernel support:

```
# Check kernel version
uname -r

# Check eBPF support
zgrep CONFIG_BPF /proc/config.gz

# Check XDP support
ip link set dev eth0 xdp off  # Should not error
```

🎓 Key Concepts Summary

1. eBPF/XDP

- **Kernel-level packet filtering** for ultra-fast processing
- **Programmable** without kernel recompilation
- **Safe** with kernel verifier guarantees
- **Efficient** with per-CPU maps and atomic operations

2. Statistical Anomaly Detection

- **Baseline learning** from normal traffic
- **Multi-factor scoring** (PPS, protocols, entropy, etc.)
- **Adaptive thresholds** based on historical data
- **False positive reduction** through multiple checks

3. Machine Learning Classification

- **Random Forest** for robust, fast predictions
- **64 CIC features** for comprehensive analysis
- **Hybrid detection** combining statistical + ML
- **Attack type identification** (SYN flood, UDP flood, etc.)

4. Real-time Processing

- **1-second monitoring loop** for timely detection
- **<10ms ML inference** for real-time classification
- **5M+ pps handling** at kernel level
- **Minimal CPU overhead** (<20% at 5M pps)

5. Blacklist Management

- **Dynamic IP blocking** based on behavior
- **Kernel-enforced** via eBPF maps
- **Instant packet drops** for blacklisted sources
- **Configurable thresholds** for auto-blocking

Security Considerations

eBPF Safety

- **Kernel verifier** ensures no crashes
- **Bounded execution** prevents infinite loops
- **Memory safety** with pointer validation
- **Privilege separation** (kernel vs user space)

Attack Surface

- **Dashboard authentication**: Not implemented (add for production)
- **API rate limiting**: Not implemented (add for production)
- **Input validation**: eBPF verifier handles kernel side
- **Log injection**: Sanitize user-controlled data

Performance Limits

- **eBPF map sizes**: Limited by configuration
- **Memory usage**: Monitor with metrics_collector
- **CPU usage**: Track to prevent resource exhaustion
- **Disk space**: Rotate logs regularly

Performance Characteristics

Throughput

- **Baseline**: 5M+ pps sustained
- **Under attack**: 10M+ pps drop rate
- **Blacklist enforcement**: Line-rate (NIC speed)

Latency

- **eBPF processing**: <1µs per packet

- **Statistical detection**: 1-second intervals
- **ML inference**: <10ms per prediction
- **Alert generation**: <100ms

Resource Usage

- **CPU**: <20% at 5M pps (4-core system)
- **Memory**: ~500MB (including ML model)
- **Disk**: ~10MB/day logs (normal traffic)

Scalability

- **Horizontal**: Multiple interfaces supported
- **Vertical**: Scales with CPU cores (per-CPU maps)
- **Map limits**: Configurable (65K flows, 131K IPs)

🚀 Future Enhancements (Phase 3)

Planned Features

1. **Auto signature generation** from detected attacks
2. **Comparative benchmarking** with other solutions
3. **Multi-interface support** with load balancing
4. **Distributed deployment** across multiple servers
5. **Advanced ML models** (XGBoost, LightGBM, Neural Networks)
6. **Anomaly explanation** (SHAP values, LIME)
7. **Automated response** (rate limiting, CAPTCHA)
8. **Integration** with SIEM systems (Splunk, ELK)

📚 References

eBPF/XDP

- eBPF Documentation
- XDP Tutorial
- BCC Reference Guide

Machine Learning

- CIC-DDoS-2019 Dataset
- scikit-learn Documentation
- Random Forest Explained

DDoS Mitigation

- NIST Cybersecurity Framework
- DDoS Attack Taxonomy

License

MIT License - See LICENSE file for details

Contributing

This project is designed for educational and research purposes. Contributions are welcome!

Areas for contribution:

- Additional attack detection algorithms
- Performance optimizations
- Cross-platform support improvements
- Documentation enhancements
- Test coverage expansion

Last Updated: January 12, 2026
Project Version: Phase 2 (ML-Enhanced Detection)
Author: Rapid-Corona Development Team