ROGUE ACCESS POINT DETECTION USING AI

Ponvedica M S
CB.SC.U4CYS23034

Soffia K N
CB.SC.U4CYS23045

Vedhavarshini Vijayakumar
CB.SC.U4CYS23052

PROBLEM STATEMENT

The ubiquitous availability of public Wi-Fi in locations such as airports, cafes, and hotels has made connectivity seamless. However, users often connect to these networks without verifying their legitimacy, creating a significant security vulnerability. The primary threat in this context is the "Evil Twin" attack, where malicious attackers deploy Rogue Access Points (APs) with SSIDs identical to legitimate, trusted networks. These rogue APs are visually indistinguishable from genuine ones, tricking users into connecting. Once connected, all user traffic can be monitored, intercepted, and manipulated, leading to Man-in-the-Middle (MITM) attacks, data theft, and other security breaches.

The challenge, therefore, is to create an accessible and robust tool that can proactively and reactively defend against such threats.

OBJECTIVES

1. To Design a Neural Network-Based Two-Stage Detection Framework
2. To Develop a Heuristic Pre-Connection Scanner
3. To Train an Artificial Intelligence Model for Behavioral Analysis
4. To Build an Intelligent Cross-Platform Security Application

DATASETS USED

To train and validate our AI model, we utilised two primary Cybersecurity datasets due to the lack of a dedicated Rogue AP dataset.

1. **CIC-IDS2017** (Canadian Institute for Cybersecurity)
   - Contains labeled network traffic flows with both benign and malicious activities.
   - Provides a foundation of real-world network behavior for training.
   - *Source :* https://www.unb.ca/cic/datasets/ids-2017.html

2. **UNSW-NB15** Dataset (UNSW Canberra Cyber Range Lab)
   - Comprises real-world attack data collected over several years.
   - Supplements CIC-IDS2017 with additional diverse attack patterns.
   - *Source:* https://research.unsw.edu.au/projects/unsw-nb15-dataset

PROPOSED SOLUTION

We propose a two-stage, defense-in-depth detection system that provides comprehensive protection against rogue access points:

Stage 1: Pre-Connection Heuristic Scan

- Performs real-time scanning of available Wi-Fi networks before connection
- Uses rule-based algorithms to detect obvious threats
- Identifies high-risk networks based on security type, SSID patterns, and duplicate networks

Stage 2: Post-Connection AI Analysis

- Implements deep learning models to analyze live network traffic
- Detects sophisticated "Evil Twin" APs through behavioral analysis
- Provides real-time threat assessment and safety scoring

ATTACK VECTOR

Spacehuhn Beacon Flooding

Beacon Flood Attack Implementation

1. Attack Simulation Setup:
   - ESP8266 Microcontroller is flashed with Spacehuhn beacon flood firmware using Ardino
   - Broadcast Parameters: 50+ fake SSIDs with randomized MAC addresses
   - Transmission Rate: 50-100ms beacon intervals across channels 1, 6, 11
2. Attack Objectives:
   - Overwhelm wireless scanners with fake AP entries
   - Hide legitimate rogue APs among beacon flood noise
   - Test system's ability to filter malicious beacon patterns
   - Evaluate detection accuracy in high-noise environments

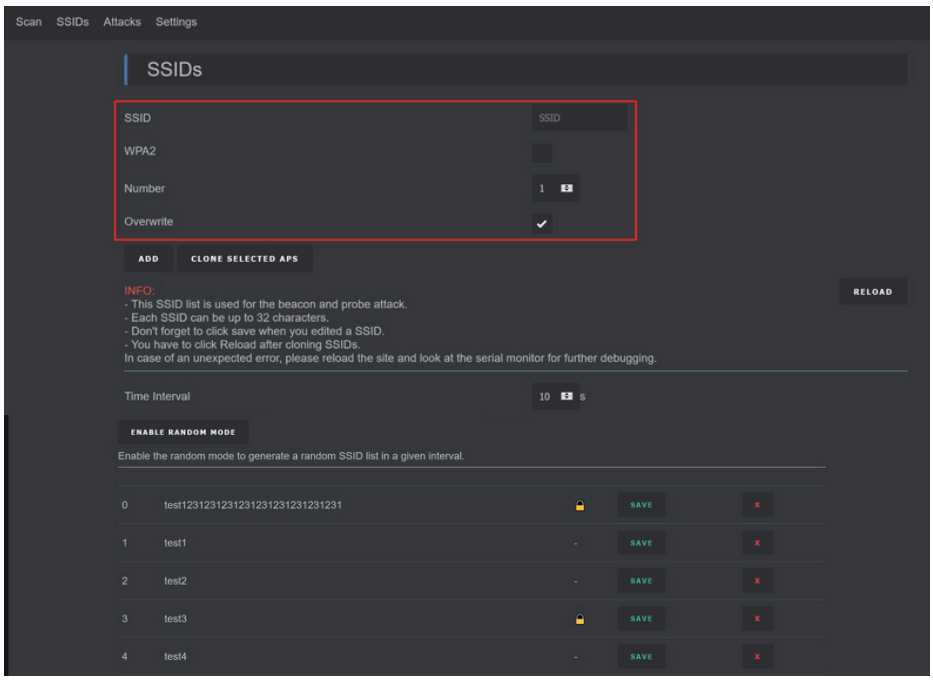


Fig 1.1 : *Configure 50+ fake SSIDs with randomized MAC addresses and enable overwrite mode for dynamic list generation.Set aggressive 10ms beacon intervals across multiple channels and activate random mode for automated attack execution.*

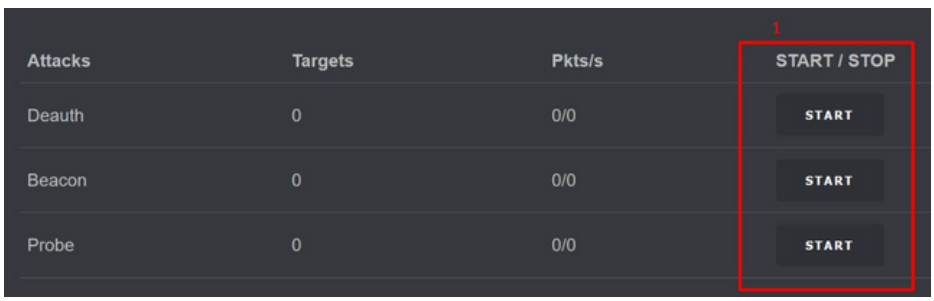


Fig 1.2 : *Click the "START" button in the Beacon attack section to initiate continuous broadcasting, monitoring real-time packet transmission rates.*

Our Defense: Our system detects the above beacon floods by combining heuristic analysis of SSID patterns and MAC randomization with AI-powered behavioral analysis to identify sophisticated threats hidden within the noise.

ALGORITHM

This project implements a hybrid two-stage detection system that combines heuristic pre-connection analysis with AI-powered post-connection behavioral monitoring for comprehensive rogue access point detection.

Step 1: *Data Collection & Feature Engineering*

1.1 *Multi-Dataset Acquisition & Integration*

- Acquired CIC-IDS2017 dataset with 79 flow-level features of normal and malicious traffic.
- Acquired UNSW-NB15 dataset with 49 features from simulated attacks.
- Merged datasets by aligning common metrics and standardizing formats to create unified training data.

1.2 *Feature Selection and Processing*

- Selected 22 critical features from CIC-IDS2017:
  - *Flow Duration, Total Fwd Packets, Total Backward Packets, Total Length of Fwd Packets, Total Length of Bwd Packets, Fwd Packet Length Mean, Bwd Packet Length Mean, Fwd Packet Length Std, Bwd Packet Length Std, Fwd Packets/s, Bwd Packets/s, Fwd PSH Flags, Bwd PSH Flags, Fwd URG Flags, Bwd URG Flags, Init_Win_bytes_forward, Init_Win_bytes_backward, bytes_ratio, traffic_asymmetry, packets_ratio, avg_packet_size.*
- Alignd and selected compatible features from UNSW-NB15:
  - *dur, spkts, dpkts, sbytes, dbytes, rate, sload, dload, sinpkt, dinpkt, sjit, djit, tcprtt, synack, ackdat, bytes_ratio, traffic_asymmetry, packets_ratio, connection_intensity, avg_packet_size.*
- Process features to handle inconsistencies, such as capping infinite values and normalizing scales.

1.3 *Data Preprocessing Pipeline*

- Aligned features by mapping equivalent metrics.
- Replaced missing values with medians, cap infinities, and clip outliers using ***StandardScalar***
- Label data: Assign binary labels (Legitimate vs. Evil Twin) based on attack patterns.
  - **0** = Legitimate Access Point
  - **1** = Evil Twin (Rogue) Access Point

1.4 *Final Feature Set*

- 22 unified network flow characteristics optimized for rogue AP detection, providing balanced representation of temporal, statistical, and behavioral patterns with comprehensive coverage of network traffic signatures for accurate classification.

Stage 1: Pre-Connection Heuristic Analysis

This stage performs passive network reconnaissance to evaluate visible WiFi access points (APs) prior to connection, focusing on structural and contextual indicators of potential RAPs.

Step 2: *Network Discovery and Parameter Extraction*

Utilize system-level utilities (such as ***netsh wlan show networks mode=bssid*** on Windows and ***aiport*** on MacOS) to enumerate proximate APs.

Capture key attributes including:

- Service Set Identifier (SSID)
- Basic Service Set Identifier (BSSID, i.e., MAC address)
- Authentication and encryption types (WPA3, WPA2-Enterprise, WPA2-Personal, WPA, WEP, OPEN)
- Signal strength (in dBm, derived from percentage values using the conversion: dBm ≈ -20 - ((100 - percent) × 0.8))
- Channel number and Organizational Unique Identifier (OUI) from BSSID. Compile a flattened list of AP profiles for subsequent analysis, handling multi-BSSID networks by associating parameters at the SSID level.

Step 3: *Feature Extraction and Pattern Matching*

- Normalized SSIDs to lowercase for case-insensitive processing.

- Extract derived features:

- Security classification: Mapping authenticated strings to standardized categories
  - "WPA2" for WPA2-Personal
  - "WPA2-ENTERPRISE" for 802.1x variants
- Duplicate SSID detection: Grouping APs by SSID and flag instances with multiple distinct BSSIDs.
- Semantic analysis: Scan SSIDs for malicious keywords ("free", "public", "guest", "hotspot", "admin", "setup") and suspicious patterns via regular expressions.
- Manufacturer inference: Extract OUI from BSSID and cross-reference against known vendor lists
- Signal and channel metrics: Compute relative signal differences among duplicate SSIDs and flag non-standard channel usage (deviations from 1, 6, 11).

Step 4: *Heuristic Risk Scoring and Categorization*

Apply a weighted additive scoring model to quantify risk:

- Security vulnerabilities: +50 for OPEN networks, +30 for WEP/WPA, -10 for WPA3/WPA2-Enterprise.
- Duplicate SSIDs: +25 if strongest signal among duplicates (with >15 dBm advantage indicating potential jamming).
- SSID anomalies: +15–25 for malicious keywords/patterns, +15 for mobile OUIs.
- Signal inconsistencies: +10–25 for unusually strong signals (-50 dBm or better) with duplicates.
- Mitigating factors: -5 for standard channels, -10 for trusted manufacturers.
- Normalize the cumulative score to [0, 100] and classify:
  - **Very Low Risk** (0–14)
  - **Low Risk** (15–29)
  - **Medium Risk** (30–49)
  - **High Risk** (50–69)
  - **Critical Risk** (70+)

Generate a ranked report with reasons, duplicate counts, and evil twin candidate summaries for user review.

Stage 2: Post-Connection AI Analysis

This stage actively monitors network traffic on the connected AP using supervised deep learning to detect behavioral anomalies indicative of RAPs, such as traffic interception or redirection. The AI workflow leverages neural network (NN) architectures tailored for sequential and statistical pattern recognition.

Step 5: Model Training and Deployment

- The loaded dataset of 22 features is taken and doing Normalization by applying StandardScaler post train-test split (80/20, stratified) to prevent leakage.

Neural Network Architectures:

We implement three interchangeable neural network architectures in TensorFlow/Keras to address different aspects of network traffic pattern recognition

- **Dense Neural Network (DNN):**
  - *Architecture:* Multilayer perceptron with layers [Dense(64, ReLU, Dropout(0.3)), Dense(32, ReLU, Dropout(0.2)), Dense(16, ReLU), Dense(1, Sigmoid)].
  - *Rationale:* The DNN effectively models static relationships among the 22 network flow features. Its fully connected layers capture high-level statistical patterns in flow metrics like packet counts and protocol behaviors. ReLU activation enables complex feature interactions, while dropout layers (30%, 20%) prevent overfitting on diverse datasets. The sigmoid output provides probabilistic classification for legitimate vs. evil twin detection.

- **Convolutional Neural Network (CNN):**
  - *Architecture:* 1D CNN with [Reshape(input_dim, 1), Conv1D(32, kernel=3, ReLU), Conv1D(16, kernel=3, ReLU), Flatten, Dense(16, ReLU), Dense(1, Sigmoid)].
  - *Rationale:* The 1D CNN detects local sequential patterns within network features using sliding kernels. It identifies spatial correlations in adjacent metrics like packet rates and TCP flags, effectively capturing traffic anomalies indicative of rogue AP behavior. Convolutional layers extract localized patterns while ReLU activation focuses on positive correlations, providing complementary detection to the DNN's global analysis.

- **Long Short-Term Memory (LSTM):**
  - *Architecture:* LSTM network with [Reshape(input_dim, 1), LSTM(32, tanh, return_sequences=True), LSTM(16, tanh), Dense(16, ReLU), Dense(1, Sigmoid)].
  - *Rationale:* The LSTM captures temporal dependencies in network traffic flows, analyzing sequential patterns in metrics like flow duration and packet timing. Its recurrent structure with stacked layers models long-term behavioral trends, effectively identifying subtle anomalies that evolve over time - crucial for detecting sophisticated rogue AP attacks. The tanh activation processes sequential data while maintaining gradient flow, making it ideal for time-series network behavior analysis.

- **Training Configuration:**
  - Use ***Adam*** optimizer (learning rate=0.001) for adaptive convergence, binary cross-entropy loss for binary classification, and early stopping (patience=3, restore best weights) to prevent overfitting. Train for up to 50 epochs with batch size 32, targeting test accuracy >0.90 for high reliability. Then model (traffic_model.h5) and scaler (traffic_scaler.pkl) is saved for reproducible inference further.

Step 6: *Real-Time Traffic Monitoring and Feature Extraction*

Capture packets via Scapy for a fixed duration (default: 30s), grouping into flows by 5-tuple (src/dst IP/port, protocol). Extracted features aligned with training:

- Temporal: Flow duration, inter-arrival times (mean/std), jitter (e.g., sjit, djit).
- Statistical: Packet/byte counts (fwd/bwd: spkts, dpkts, sbytes, dbytes), lengths (mean/std/min/max), rates (fwd_pkts_s, sload).
- Protocol: TCP flags (PSH/URG/SYN/ACK/FIN: fwd_psh_flags, bwd_urg_flags), window sizes (init_win_bytes_fwd), RTT components (tcprtt, synack).
- Wireless-Specific: Beacon counts, SSID/channel variations, signal strengths (avg_signal_strength), encryption types.
- Derived: Ratios (bytes_ratio, packets_ratio), asymmetry (traffic_asymmetry), port/DNS query entropy.
- Output feature dictionaries compatible with the trained model's input schema.

Step 7: *AI Workflow and Inference*

The AI workflow processes live traffic through a structured pipeline:

- Input Preparation: Align live features with training schema, filling missing values with zeros, replacing infinities/NaNs, and scaling via loaded StandardScaler. Reshape for CNN/LSTM inputs (e.g., [1, input_dim, 1]).
- Prediction: Feed features to the selected NN (default: Dense; configurable to CNN/LSTM via model_type). The model outputs a sigmoid probability (0–1) for evil twin likelihood.
- Thresholding: Classify based on probability:
  - 0.7: Evil twin, safety_score = (1 - prob) × 100.
  - <0.3: Legitimate, safety_score = (1 - prob) × 100.
  - 0.3–0.7: Uncertain, safety_score = 50.
- Aggregation: Compute overall risk across flows (e.g., minimum/average safety scores). Flag high-risk if any flow's safety_score <30. Cross-validate with heuristic indicators (e.g., duplicate SSIDs, weak security) for robustness.

Step 8: *Reporting and Response Generation*

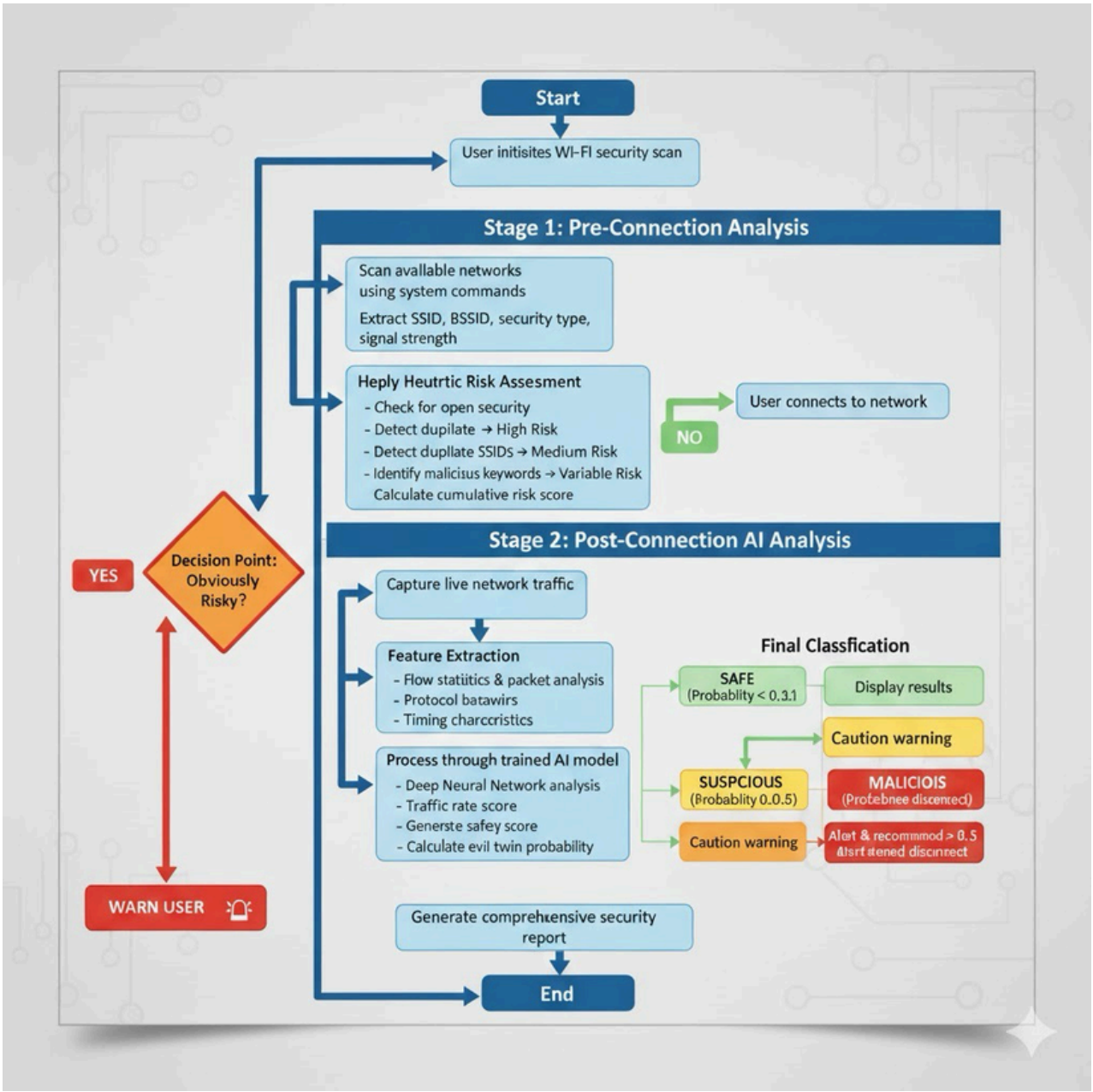
Via Streamlit UI, we present:

- Metrics: Safety score, evil twin probability, risk level (SAFE/CAUTION/UNSAFE).
- Recommendations: For example, "Disconnect immediately" for critical risks (<30 safety), "Proceed with caution" for medium risks.
- Detailed Breakdown: Flow statistics, risk reasons, and wireless-specific insights.

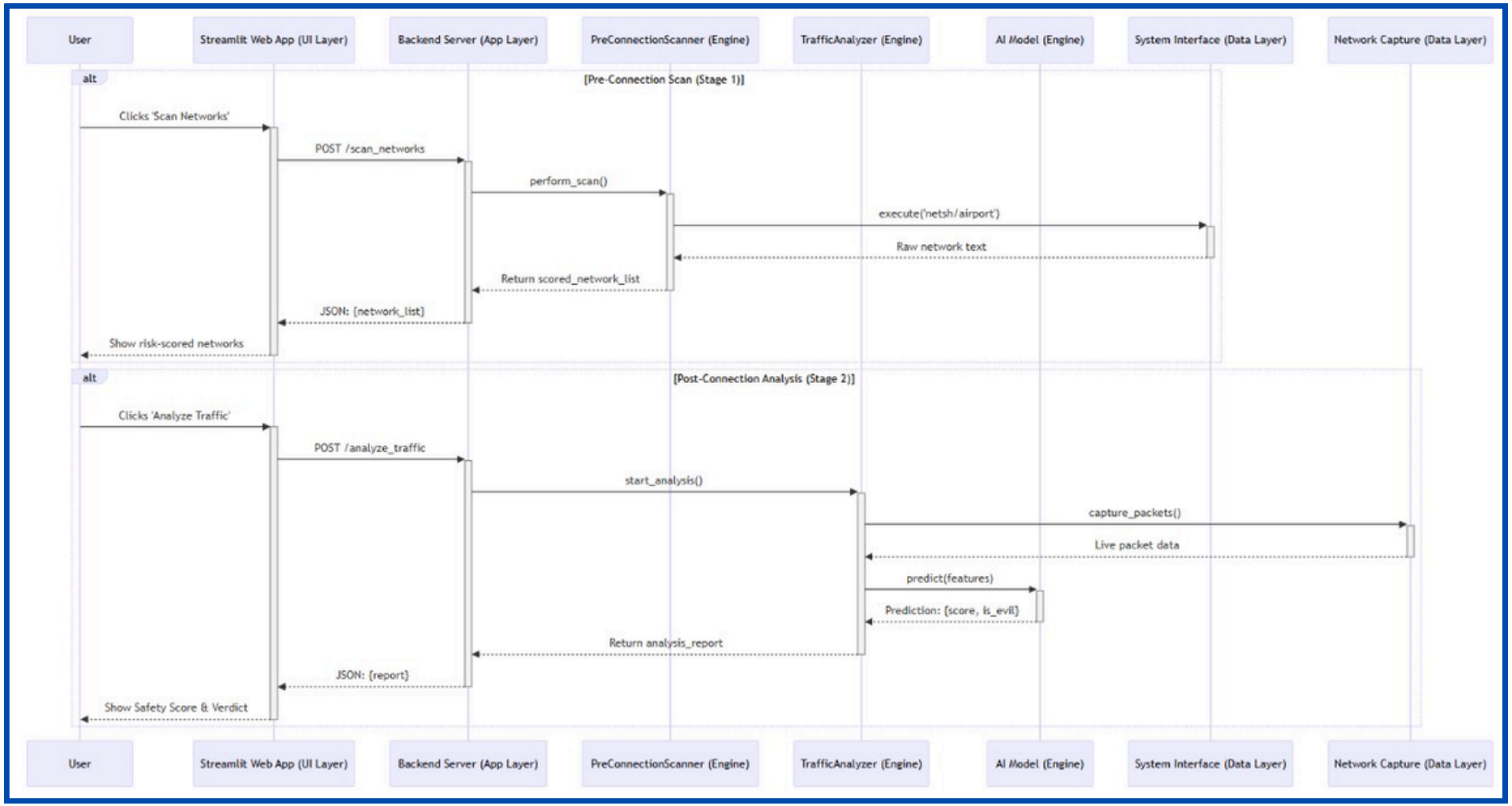
- Support dashboard views for scan results, traffic analysis, and model status.

This dual-stage algorithm integrates deterministic heuristics with probabilistic NN-based inference (DNN/CNN/LSTM), leveraging comprehensive feature engineering and robust validation to detect RAPs with high accuracy while minimizing false positives.

FLOWCHART



ARCHITECTURE DIAGRAM



This architecture diagram illustrates the end-to-end data flow and component interaction in our rogue AP detection system

OUTPUT

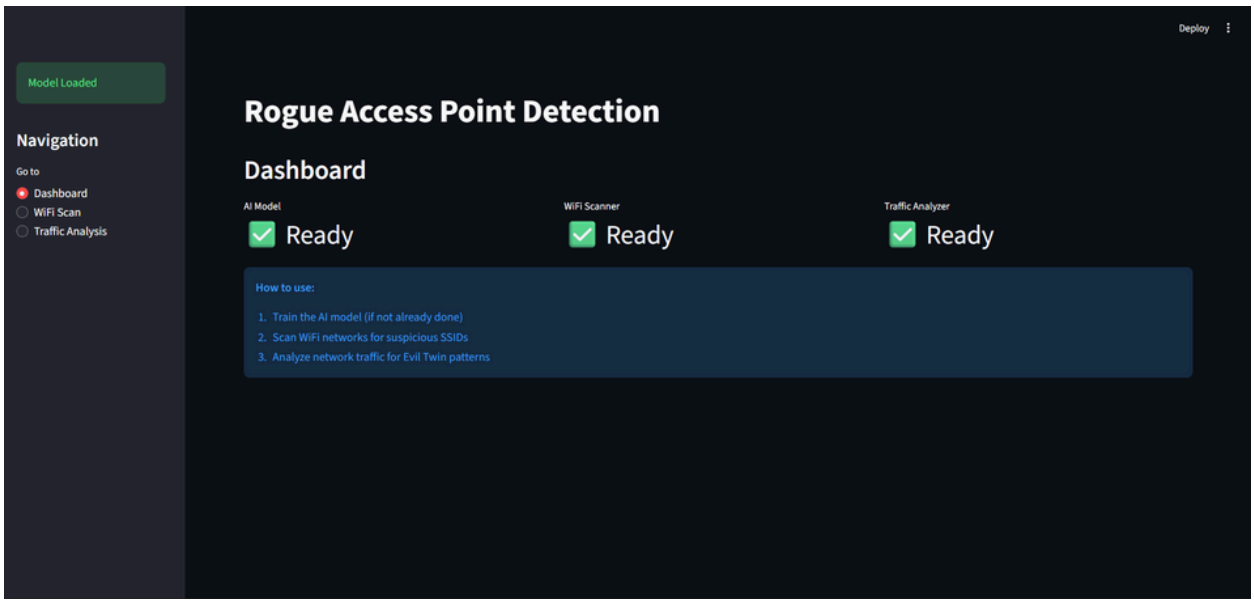


Fig 2.1 : *This diagram shows the Streamlit-based web application interface for rogue access point detection. The left sidebar displays navigation options (Dashboard, WiFi Scan, Traffic Analysis) with the AI model successfully loaded.*

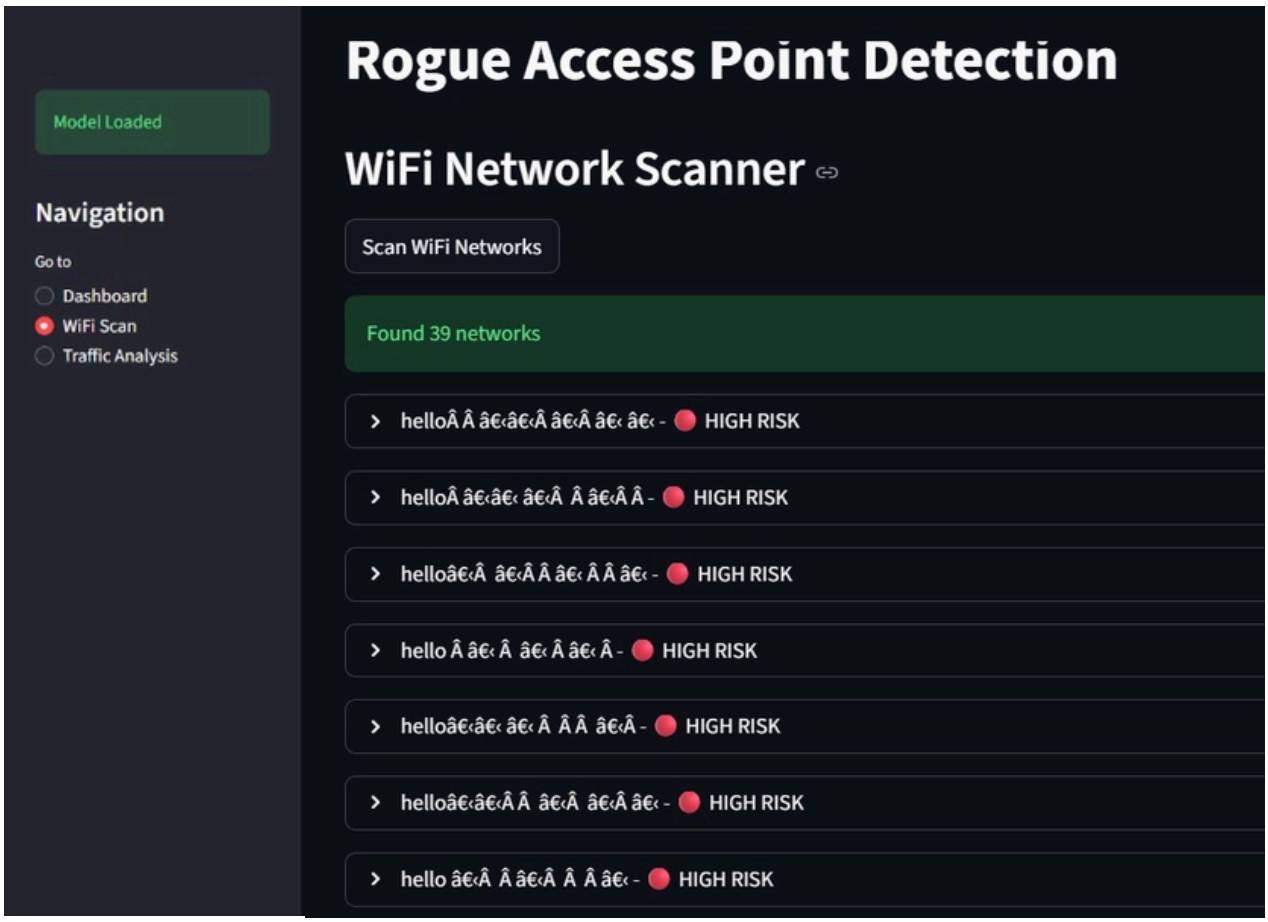


Fig 2.2 : *The main panel shows real-time WiFi scanning results, detecting 39 networks with color-coded risk assessment. Multiple suspicious "hello" networks are flagged with HIGH RISK indicators (red)*

Fig 2.3.1 ***Fig 2.3.2***

Fig 2.3.1 : *This diagram provides detailed analysis of a detected suspicious network. The "hello" access point shows critical risk factors: OPEN security (high vulnerability), strong signal strength (-56 dBm indicating potential proximity spoofing), and elevated risk score (50/100). The system identifies this as a high-risk network based on heuristic analysis of security configuration, signal characteristics, and SSID patterns, warning users against connection.*

Fig 2.3.2 : *The right one - while legitimate "Amrita" campus networks show VERY LOW RISK (green), demonstrating the system's ability to automatically identify and categorize potential threats.*

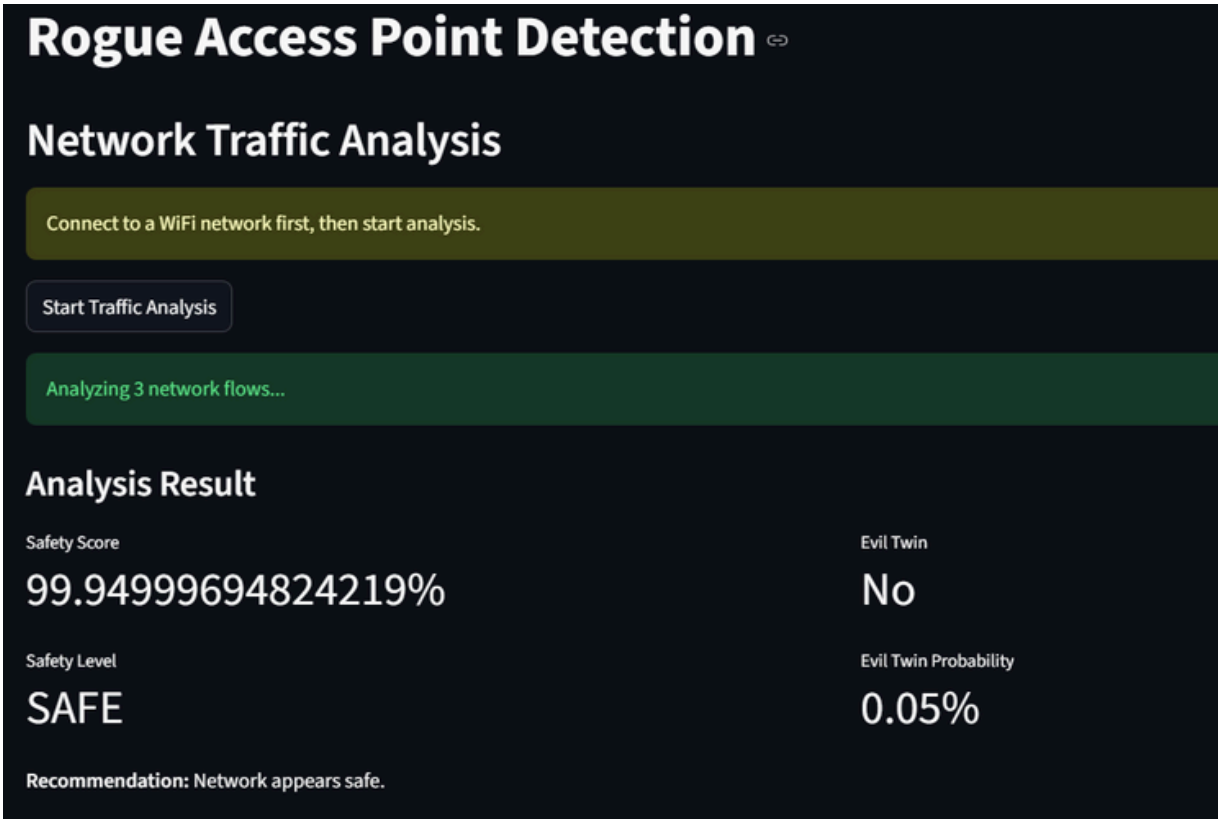


Fig 2.4 : *This diagram demonstrates the system's analysis of a verified legitimate network. When connected to "Amrita" campus WiFi, the traffic analysis module shows excellent safety metrics: 99.95% safety score, "SAFE" classification, and minimal evil twin probability (0.05%).*

The AI model confirms network legitimacy through behavioral analysis, providing users with confidence in their connection security and demonstrating the system's accurate differentiation between genuine and rogue access points.

FUTURE SCOPE

The rogue access point (RAP) detection system, as currently implemented, offers a robust hybrid framework. However, its potential can be significantly expanded through the following strategic enhancements, which aim to integrate advanced technologies, improve model robustness, and evolve it into a collaborative security ecosystem. These enhancements are designed to address current limitations and anticipate future wireless threats.

RF Fingerprinting Integration

- Deployment of SDR Technology: Incorporate Software-Defined Radio (SDR) to analyze physical layer characteristics, such as modulation schemes, carrier frequency offsets, and phase noise, enabling the detection of radio signal patterns unique to individual transmitters.
- Hardware-Based Identification: Develop a methodology to identify APs using inherent transmitter imperfections and manufacturing variations (e.g., non-linear distortions, clock jitter), creating a fingerprint that distinguishes legitimate APs from cloned evil twins.
- Advanced Signal Processing: Implement techniques like wavelet transforms or machine learning-based classifiers to detect subtle RF signature differences, identifying perfectly cloned evil twins that evade network-level detection.
- Radio Distinctiveness Profiles: Establish trusted baseline patterns for legitimate APs by creating radio distinctiveness profiles, allowing the system to flag deviations as potential threats with high confidence.

GAN-Based Synthetic Attack Generation

- Realistic Traffic Simulation: Leverage Generative Adversarial Networks (GANs) to synthesize realistic rogue AP traffic patterns and attack scenarios, including advanced techniques like deauthentication floods or rogue DHCP servers.
- Adversarial Example Generation: Produce adversarial examples that mimic evolving attack strategies to stress-test the DNN, CNN, and LSTM models, enhancing their resilience against sophisticated threats.
- Enhanced Data Diversity: Address data scarcity by augmenting training datasets with synthetic malicious behaviors (e.g., anomalous packet sequences, spoofed SSIDs), improving the model's ability to generalize across rare attack types.
- Continuous Learning: Integrate a feedback loop where the model adapts in real-time to new threat patterns generated by the GAN, ensuring ongoing relevance as attack techniques evolve.
- Firebase-Enabled WiFi Analysis Platform

Cloud-Based Threat Intelligence

- Utilize Firebase to develop a scalable, real-time cloud platform for aggregating crowd-sourced data on detected rogue APs, enhancing global threat visibility.
- Centralized Logging and Analysis: Implement centralized logging of RAP incidents across multiple users and locations, enabling longitudinal analysis to identify trends and persistent threats.

- Live Threat Visualization: Create dynamic threat maps and heat maps using geospatial data from user reports, highlighting areas with high malicious activity concentrations for proactive defense.
- Collaborative Defense System: Build a networked ecosystem where a detection by one user triggers automatic updates and enhanced protection for all platform users, leveraging collective intelligence.
- Real-Time Alerts: Enable push notifications and updates about emerging threats detected across the network, ensuring users receive timely warnings (e.x., at 12:09 AM IST on October 1st, 2025, a new threat pattern could be flagged instantly).

These enhancements will elevate the system from a standalone detection tool to a comprehensive, community-powered security ecosystem. By integrating RF fingerprinting, the system will gain physical-layer precision; GAN-based generation will bolster AI robustness; and a Firebase platform will enable real-time, collaborative threat mitigation. This evolution positions the solution to anticipate and neutralize sophisticated wireless threats, ensuring adaptability in an increasingly complex cybersecurity landscape.

COMPARATIVE STUDY

These points are inferred by comparing the system's hybrid approach, flexibility, and real-time capabilities against typical commercial or open-source solutions (e.g., Wireshark, Kismet, or AI-based tools like Snort with machine learning extensions), which often lack integrated hybrid frameworks or real-time adaptability.

- ***Hybrid Two-Stage Detection Framework***
  - Our model combines a pre-connection heuristic analysis (Stage 1) with a post-connection AI-based behavioral analysis (Stage 2), offering a layered defense mechanism. Unlike many existing tools (e.g., Kismet, which focuses solely on passive scanning), this ***dual approach*** leverages both rule-based risk scoring and deep learning (DNN/CNN/LSTM) to detect RAPs more comprehensively, reducing false negatives across diverse attack scenarios.

- ***Real-Time AI Inference with Multiple Neural Network Options***
  - The system employs interchangeable neural network architectures (DNN, CNN, LSTM) trained on datasets like CIC-IDS2017 and UNSW-NB15, enabling real-time traffic analysis with adaptability to sequential and temporal patterns. Most existing tools (Wireshark or traditional IDS like Snort) *rely on static signature-based detection* or require manual rule updates, lacking the dynamic learning capability our model provides.

- ***Customizable and Open-Source Feature Engineering***
  - With a tailored feature set (e.g., 22 unified metrics like flow_duration, traffic_asymmetry, fwd_pkts_s) and preprocessing pipeline (e.g., outlier clipping, StandardScaler), our model allows fine-tuning for specific network environments. Commercial tools often use predefined feature sets, limiting adaptability, while our open-source design (via Python scripts) enables users to modify and optimize features as needed.

- ***Integrated User Interface with Actionable Insights***
  - The Streamlit-based UI provides a dashboard with real-time metrics (safety scores, risk levels, recommendations) and detailed reports, enhancing usability. Existing tools like Aircrack-ng or Nessus offer raw data or basic alerts but lack an integrated, user-friendly interface that translates complex AI outputs into actionable advice (e.g., “Disconnect immediately” for critical risks).

- ***Efficient Resource Utilization with Early Stopping and Fallback Mechanisms***
  - Our model incorporates early stopping (patience=3) during training and a heuristic fallback for uncertain AI predictions, optimizing computational resources and ensuring reliability. Many AI-based security tools (e.g., open-source ML extensions for Snort) may overtrain or fail to handle edge cases without such mechanisms, leading to higher resource consumption or missed detections.

Feature	Traditional Tools (Kismet, Aircrack-ng)	Enterprise Solutions (Cisco, Aruba)	Our AI Solution
Detection Method	Manual analysis & signature-based	RF fingerprinting & centralized monitoring	Two-stage AI-powered analysis
Pre-Connection Analysis	Limited or manual	Basic scanning	Automated heuristic risk scoring
Post-Connection Analysis	Manual packet inspection	Hardware-dependent monitoring	Real-time AI behavioral analysis
Automation Level	Low (requires expert intervention)	Medium (configured by administrators)	High (fully automated)
Ease of Use	Technical expertise required	Enterprise IT skills needed	User-friendly interface
Cost	Free/Open-source	High (hardware + licensing)	Low (software-based)
Evil Twin Detection	Basic SSID matching	Good with RF features	Advanced behavioral pattern recognition
Beacon Flooding Resilience	Poor (overwhelmed by noise)	Good (enterprise-grade filtering)	Excellent (AI-powered noise filtering)
Real-time Threat Assessment	Manual interpretation	Delayed reporting	Immediate safety scoring
Adaptive Learning	No	Limited	Yes (ML model improves with data)
Accessibility	Security professionals	Large enterprises	General public & organizations

CONCLUSION

- Our AI-powered rogue access point detection system represents a significant advancement in wireless security by bridging the gap between open-source tools and enterprise solutions. The comparative analysis demonstrates that our solution outperforms traditional methods in automation, accuracy, and accessibility while providing enterprise-grade detection capabilities at a fraction of the cost.

- The two-stage AI architecture delivers comprehensive protection by combining pre-connection heuristic analysis with post-connection behavioral monitoring, effectively addressing both basic and sophisticated attack vectors. The system's ability to provide real-time safety scoring and adaptive learning makes it uniquely positioned for widespread adoption among general users, educational institutions, and small-to-medium enterprises who require robust wireless security without the complexity of traditional tools or the expense of enterprise solutions.

- This approach democratizes advanced rogue AP detection, making enterprise-level security accessible to a broader audience while maintaining high detection accuracy and operational efficiency.

GITHUB LINKS

1. ponvedica
2. Soffia-275
3. vedha73varshini

APPENDIX

app.py

```
import streamlit as st
import pandas as pd
import numpy as np
import os
from model import EvilTwinDetector
from wifi_scanner import WiFiScanner
from traffic_analyzer import TrafficAnalyzer

st.set_page_config(
  page_title="Rogue Access Point Detection",
  layout="wide",
  page_icon="🛡️"
)

class EvilTwinApp:
  def __init__(self):
    self.wifi_scanner = WiFiScanner()
    self.traffic_analyzer = TrafficAnalyzer()
    self.detector = None
    self.load_detector()

  def load_detector(self):
    """Load the AI model"""
    try:
      self.detector = EvilTwinDetector()
      if self.detector.model_loaded:
        st.sidebar.success("Model Loaded")
      else:
        st.sidebar.error("AI Model Not Trained")
    except:
      st.sidebar.error("AI Model Not Available")

  def run(self):
    st.title("Rogue Access Point Detection")

    st.sidebar.title("Navigation")
    page = st.sidebar.radio("Go to", ["Dashboard", "WiFi Scan", "Traffic Analysis"])

    if page == "Dashboard":
      self.show_dashboard()
    elif page == "WiFi Scan":
      self.show_wifi_scan()
    elif page == "Traffic Analysis":
      self.show_traffic_analysis()
   def show_dashboard(self):
    st.header("Dashboard")

    # Check model status
    model_exists = os.path.exists('traffic_model.h5')

    col1, col2, col3 = st.columns(3)
    with col1:
      st.metric("AI Model", "Ready" if model_exists else "Not Trained")
```

```
  with col2:
    st.metric("WiFi Scanner", "Ready")
  with col3:
    st.metric("Traffic Analyzer", "Ready")

  if not model_exists:
    st.error("""
    **AI Model Not Trained!**

    Please train the model first:
    ```bash
    python train.py
    ```
    """)

  st.info("""
  How to use:
  1. Train the AI model (if not already done)
  2. Scan WiFi networks for suspicious SSIDs
  3. Analyze network traffic for Evil Twin patterns
  """)

def show_wifi_scan(self):
  st.header("WiFi Network Scanner")

  if st.button("Scan WiFi Networks"):
    with st.spinner("Scanning..."):
      networks = self.wifi_scanner.scan_networks()

    if networks:
      st.success(f"Found {len(networks)} networks")

      for network in networks:
        with st.expander(f"{network['ssid']} - {network['risk_level']}"):
          st.write(f"**Security:** {network['security']}")
          st.write(f"**Signal:** {network['signal_strength']} dBm")
          st.write(f"**Risk Score:** {network['risk_score']}/100")
    else:
      st.error("No networks found or scan failed")

def show_traffic_analysis(self):
  st.header("Network Traffic Analysis")

  if not self.detector or not self.detector.model_loaded:
    st.error("AI model not trained. Please run `python train.py` first.")
    return

  st.warning("Connect to a WiFi network first, then start analysis.")

  if st.button("Start Traffic Analysis"):
    with st.spinner("Capturing traffic for 30 seconds..."):
      features_list = self.traffic_analyzer.capture_traffic(duration=30)

    if features_list:
      st.success(f"Analyzing {len(features_list)} network flows...")

      results = []
      for features in features_list:
        result = self.detector.analyze_network_traffic(features)
        if 'error' not in result:
          results.append(result)
```

```
        # Show worst result
        if results:
          worst = min(results, key=lambda x: x['safety_score'])

          st.subheader("Analysis Result")
          col1, col2 = st.columns(2)
          with col1:
            st.metric("Safety Score", f"{worst['safety_score']}%")
            st.metric("Safety Level", worst['safety_level'])
          with col2:
            st.metric("Evil Twin", "Yes" if worst['is_evil_twin'] else "No")
            st.metric("Evil Twin Probability", worst['probability_evil_twin'])

          st.write(f"**Recommendation:** {worst['recommendation']}")
        else:
          st.error("No valid traffic flows analyzed")
      else:
        st.error("No traffic captured. Please try again.")

if __name__ == "__main__":
  app = EvilTwinApp()
  app.run()
```

model.py

```
import pandas as pd
import numpy as np
import joblib
import warnings
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Conv1D, Flatten, LSTM, Reshape
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping

warnings.filterwarnings('ignore')

class EvilTwinModel:
  def __init__(self, model_type='dense'):
    self.traffic_model = None
    self.traffic_scaler = None
    self.model_type = model_type # 'dense', 'cnn', 'lstm'

  def create_model(self, input_dim):
    """Create model: dense, cnn, or lstm"""
    if self.model_type == 'dense':
      model = Sequential([
        Dense(64, activation='relu', input_shape=(input_dim,)),
        Dropout(0.3),
        Dense(32, activation='relu'),
        Dropout(0.2),
        Dense(16, activation='relu'),
        Dense(1, activation='sigmoid')
      ])
```

```
    elif self.model_type == 'cnn':
      model = Sequential([
        Reshape((input_dim, 1), input_shape=(input_dim,)),
        Conv1D(32, kernel_size=3, activation='relu'),
        Conv1D(16, kernel_size=3, activation='relu'),
        Flatten(),
        Dense(16, activation='relu'),
        Dense(1, activation='sigmoid')
      ])
    elif self.model_type == 'lstm':
      model = Sequential([
        Reshape((input_dim, 1), input_shape=(input_dim,)),
        LSTM(32, activation='tanh', return_sequences=True),
        LSTM(16, activation='tanh'),
        Dense(16, activation='relu'),
        Dense(1, activation='sigmoid')
      ])
    else:
      raise ValueError("Invalid model_type. Choose 'dense', 'cnn', or 'lstm'.")

    model.compile(
      optimizer=Adam(learning_rate=0.001),
      loss='binary_crossentropy',
      metrics=['accuracy']
    )
    return model

  def load_small_dataset(self):
    print(" Loading small dataset for fast training...")
    try:
      df = pd.read_csv('data/cic_evil_twin_processed.csv')
      print(f"Loaded CIC dataset: {len(df)} samples")
    except:
      try:
        df = pd.read_csv('data/unsw_evil_twin_processed.csv')
        print(f"Loaded UNSW dataset: {len(df)} samples")
      except Exception as e:
        print(f"Could not load any dataset: {e}")
        return None
    if len(df) > 2000:
      df = df.sample(n=10000, random_state=42)
      print(f"Using 2000 samples for fast training")
    return df

  def clean_data(self, X):
    print("🧹 Cleaning data (removing infinity/large values)...")
    X_clean = X.replace([np.inf, -np.inf], np.nan)
    inf_count = (X == np.inf).sum().sum() + (X == -np.inf).sum().sum()
    nan_count = X_clean.isna().sum().sum()
    if inf_count > 0: print(f"  Found {inf_count} infinite values")
    if nan_count > 0: print(f"  Found {nan_count} NaN values after cleaning")
    for col in X_clean.columns:
      col_median = X_clean[col].median()
      X_clean[col] = X_clean[col].fillna(col_median)
      upper_limit = X_clean[col].quantile(0.999)
      lower_limit = X_clean[col].quantile(0.001)
      if np.isfinite(upper_limit) and np.isfinite(lower_limit):
        X_clean[col] = np.clip(X_clean[col], lower_limit, upper_limit)
    return X_clean
```

```
  def preprocess_fast(self, df):
    print("🔧 Fast preprocessing...")
    target_column = None
    for col in ['is_evil_twin', 'Label', 'label', 'is_malicious', 'target']:
      if col in df.columns:
        target_column = col
        break
    if target_column is None:
      for col in df.columns:
        if df[col].dtype in [np.int64, np.float64] and df[col].nunique() <= 10:
          target_column = col
          break
    if target_column is None:
      print("No target column found!")
      return None, None, None
    print(f"Target column: {target_column}")
    if df[target_column].dtype == 'object':
      y = np.array([0 if 'BENIGN' in str(label).upper() or 'NORMAL' in str(label).upper()
else 1
              for label in df[target_column]])
    else:
      y = df[target_column].values
      if len(np.unique(y)) > 2: y = (y > 0).astype(int)
    print(f"Class distribution: {np.unique(y, return_counts=True)}")
    exclude_cols = [target_column, 'attack_cat', 'id', 'ssid', 'timestamp', 'time', 'date']
    feature_cols = [col for col in df.columns if col not in exclude_cols and df[col].dtype in
[np.int64, np.float64]]
    if not feature_cols:
      feature_cols = [col for col in df.columns if col not in exclude_cols]
      for col in feature_cols:
        df[col] = pd.to_numeric(df[col], errors='coerce')
      df[feature_cols] = df[feature_cols].fillna(0)
    X = df[feature_cols]
    X_clean = self.clean_data(X)
    print(f"Using {len(feature_cols)} features")
    return X_clean, y, feature_cols

  def train_fast(self):
    print("STARTING FAST TRAINING...")
    try:
      df = self.load_small_dataset()
      if df is None: return False, "Failed to load dataset"
      X, y, features = self.preprocess_fast(df)
      if X is None: return False, "Preprocessing failed"
      if len(X) < 100: return False, f"Not enough data after cleaning: {len(X)} samples"
      print(f"Data ready: {X.shape[0]} samples, {X.shape[1]} features")
      print("Scaling features...")
      self.traffic_scaler = StandardScaler()
      try:
        X_scaled = self.traffic_scaler.fit_transform(X)
      except Exception as e:
        print(f"Standard scaling failed, using RobustScaler: {e}")
        from sklearn.preprocessing import RobustScaler
        self.traffic_scaler = RobustScaler()
        X_scaled = self.traffic_scaler.fit_transform(X)
      X_train, X_test, y_train, y_test = train_test_split(
        X_scaled, y, test_size=0.2, random_state=42, stratify=y
      )
```

```
      print("Training model ")
      history = self.traffic_model.fit(
        X_train, y_train,
        epochs=50,
        batch_size=32,
        validation_data=(X_test, y_test),
        verbose=1,
        callbacks=[EarlyStopping(patience=3, restore_best_weights=True)]
      )
      test_loss, test_accuracy = self.traffic_model.evaluate(X_test, y_test, verbose=0)
      print(f"Training complete! Accuracy: {test_accuracy:.4f}")
      self.traffic_model.save('traffic_model.h5')
      joblib.dump(self.traffic_scaler, 'traffic_scaler.pkl')
      print("Models saved: traffic_model.h5, traffic_scaler.pkl")
      return True, f"Fast training complete! Accuracy: {test_accuracy:.4f}"
    except Exception as e:
      print(f"Training failed with error: {e}")
      return False, f"Training error: {e}"

class EvilTwinDetector:
  def __init__(self):
    try:
      self.traffic_model = tf.keras.models.load_model('traffic_model.h5')
      self.traffic_scaler = joblib.load('traffic_scaler.pkl')
      self.model_loaded = True
      print("AI Model loaded successfully")
    except Exception as e:
      print(f"Error loading model: {e}")
      self.model_loaded = False

  def analyze_network_traffic(self, feature_dict):
    if not self.model_loaded:
      return {'error': "Model not trained. Please train first."}
    try:
      feature_df = pd.DataFrame([feature_dict])
      for feature in self.traffic_scaler.feature_names_in_:
        if feature not in feature_df.columns:
          feature_df[feature] = 0
      feature_df = feature_df[self.traffic_scaler.feature_names_in_].fillna(0)
      for col in feature_df.columns:
        feature_df[col] = pd.to_numeric(feature_df[col], errors='coerce')
      feature_df = feature_df.replace([np.inf, -np.inf], 0)
      feature_df = feature_df.fillna(0)
      # reshape for CNN/LSTM
      feature_scaled = self.traffic_scaler.transform(feature_df)
      if len(self.traffic_model.input_shape) == 3:
        feature_scaled = feature_scaled.reshape(feature_scaled.shape[0],
feature_scaled.shape[1], 1)
      prediction_prob = self.traffic_model.predict(feature_scaled, verbose=0)[0][0]
      if prediction_prob > 0.7:
        is_evil_twin = True
        safety_score = (1 - prediction_prob) * 100
      elif prediction_prob < 0.3:
        is_evil_twin = False
        safety_score = (1 - prediction_prob) * 100
      else:
        is_evil_twin = prediction_prob > 0.5
        safety_score = 50
      if is_evil_twin:
        if safety_score < 30:
          recommendation = "EVIL TWIN DETECTED! Disconnect immediately!"
        else:
          recommendation = "Suspicious network detected."
```

```
      else:
        if safety_score >= 80:
          recommendation = "Network appears safe."
        else:
          recommendation = "Network shows minor anomalies."
      return {
        'safety_score': round(safety_score, 2),
        'safety_level': "SAFE" if safety_score >= 70 else "CAUTION" if safety_score >= 50
else "🔴 UNSAFE",
        'recommendation': recommendation,
        'is_evil_twin': bool(is_evil_twin),
        'probability_evil_twin': f"{prediction_prob:.2%}",
        'probability_legitimate': f"{(1-prediction_prob):.2%}"
      }
    except Exception as e:
      return {'error': f"Analysis failed: {e}"}

def train_model_standalone(model_type='dense'):
  print("=" * 50)
  print(f"EVIL TWIN DETECTOR - FAST TRAINING ({model_type.upper()})")
  print("=" * 50)
  trainer = EvilTwinModel(model_type=model_type)
  success, message = trainer.train_fast()
  if success:
    print("\nTRAINING SUCCESSFUL!")
    print("\nNow you can run: streamlit run app.py")
  else:
    print(f"\nTRAINING FAILED: {message}")

if __name__ == "__main__":
  # Change 'dense' to 'cnn' or 'lstm' as needed
  train_model_standalone(model_type='dense')
```

train.py

```
print("Starting training...")

# Import and run the training function
from model import train_model_standalone

if __name__ == "__main__":
  train_model_standalone()
```

wifi_scanner.py

```
import subprocess
import re
from collections import defaultdict
import time
from datetime import datetime

class WiFiScanner:
  def __init__(self):
    self.trusted_enterprise_prefixes = ['CORP_', 'OFFICE_', 'ENTERPRISE_', 'COMPANY_']
    self.malicious_keywords = ['free', 'public', 'guest', 'hotspot', 'admin', 'setup']
    self.legitimate_brands = ['starbucks', 'att', 'verizon', 'xfinity', 'tmobile', 'google']

    self.suspicious_patterns = [
      r'^[0-9]{10,}$',          # Long numeric SSIDs
      r'^[A-Z]{15,}$',          # Long uppercase strings
      r'.*@.*',              # Contains @ symbol
      r'.*_.*_.*_.*',           # Multiple underscores
      r'^DIRECT-[a-zA-Z0-9]{4}',     # Windows direct connection
      r'^AndroidAP_[0-9a-fA-F]{6}$',   # Generic Android hotspot
    ]

    # Known manufacturer OUI prefixes (first 3 bytes of MAC)
    self.trusted_manufacturers = [
      'Cisco', 'Aruba', 'Ruckus', 'Ubiquiti', 'Meraki',
      'Netgear', 'TP-Link', 'D-Link', 'Linksys'
    ]


  def parse_netsh_output(self, output):
    """Parse netsh command output with improved security detection"""
    networks = []
    current_ssid = None
    networks_dict = {}

    lines = output.split('\n')

    for line in lines:
      line = line.strip()

      if line.startswith('SSID') and 'BSSID' not in line:
        ssid_match = re.match(r'SSID\s*\d+\s*:\s*(.+)', line)
        if ssid_match:
          current_ssid = ssid_match.group(1).strip()
          if current_ssid not in networks_dict:
            networks_dict[current_ssid] = {'bssids': [], 'security': 'UNKNOWN'}

      elif line.startswith('BSSID') and current_ssid:
        bssid_match = re.search(r'BSSID\s*\d+\s*:\s*([0-9a-fA-F:]{17})', line)
        if bssid_match:
          current_bssid = bssid_match.group(1).upper()
          networks_dict[current_ssid]['bssids'].append({
            'mac': current_bssid,
            'signal': -100,
            'signal_percent': 0,
            'channel': 0,
          })
```

```
      elif 'Signal' in line and current_ssid and networks_dict[current_ssid]['bssids']:
        signal_match = re.search(r'Signal\s*:\s*(\d+)%', line, re.IGNORECASE)
        if signal_match:
          signal_percent = int(signal_match.group(1))
          networks_dict[current_ssid]['bssids'][-1]['signal'] =
self.convert_signal_to_dbm(signal_percent)
          networks_dict[current_ssid]['bssids'][-1]['signal_percent'] = signal_percent

      elif 'Channel' in line and current_ssid and networks_dict[current_ssid]['bssids']:
        channel_match = re.search(r'Channel\s*:\s*(\d+)', line, re.IGNORECASE)
        if channel_match:
          networks_dict[current_ssid]['bssids'][-1]['channel'] = int(channel_match.group(1))

      elif 'Authentication' in line and current_ssid:
        auth_match = re.search(r'Authentication\s*:\s*(.+)', line, re.IGNORECASE)
        if auth_match:
          auth_type = auth_match.group(1).strip()
          networks_dict[current_ssid]['security'] = self.classify_security(auth_type)

    # Convert dictionary to list format
    for ssid, data in networks_dict.items():
      for bssid in data['bssids']:
        networks.append({
          'ssid': ssid,
          'bssid_mac': bssid['mac'],
          'security': data['security'], # Use the SSID-level security
          'signal_strength': bssid['signal'],
          'signal_percent': bssid.get('signal_percent', 0),
          'channel': bssid['channel']
        })
    return networks

  def classify_security(self, auth_type):
    """Improved security classification"""
    if not auth_type:
      return 'UNKNOWN'

    auth_lower = auth_type.lower()
    # WPA3
    if 'wpa3' in auth_lower:
      return 'WPA3'

    # Enterprise
    elif any(x in auth_lower for x in ['enterprise', '802.1x']):
      return 'WPA2-ENTERPRISE'

    # WPA2
    elif any(x in auth_lower for x in ['wpa2', 'wpa2-personal']):
      return 'WPA2'

    # WPA
    elif 'wpa' in auth_lower:
      return 'WPA'

    # WEP
    elif 'wep' in auth_lower:
      return 'WEP'

    # Open network
    elif any(x in auth_lower for x in ['open', 'none']):
      return 'OPEN'

    else:
      return 'UNKNOWN'
```

```
  def convert_signal_to_dbm(self, signal_percent):
    """Convert signal percentage to dBm accurately"""
    # More accurate conversion: 100% = -20 dBm, 0% = -100 dBm
    return int(-20 - ((100 - signal_percent) * 0.8))

  def analyze_manufacturer_risk(self, mac_address):
    """Analyze MAC address manufacturer for risk assessment"""
    # Extract OUI (first 3 bytes)
    oui = mac_address.replace(':', '')[:6].upper()

    common_router_ouis = ['000C43', '001DE1', '0022B0', '14CC20', '1C3BF3']
    mobile_ouis = ['A0F849', '885395', 'F0272D']

    if oui in common_router_ouis:
      return -10, "Known router manufacturer"
    elif oui in mobile_ouis:
      return 15, "Mobile device hotspot"
    else:
      return 0, "Unknown manufacturer"

  def analyze_signal_risk(self, network, all_networks):
    """Advanced signal analysis for evil twin detection"""
    risk_score = 0
    reasons = []

    ssid = network['ssid']
    current_bssid = network.get('bssid_mac', '')
    current_signal = network.get('signal_strength', -100)

    # Find all networks with same SSID
    same_ssid_networks = [
      n for n in all_networks
      if n['ssid'] == ssid and n.get('bssid_mac') != current_bssid
    ]

    if same_ssid_networks:
      # Compare signal strengths among duplicates
      strongest_signal = max(
        [n.get('signal_strength', -100) for n in same_ssid_networks] + [current_signal]
      )
      weakest_signal = min(
        [n.get('signal_strength', -100) for n in same_ssid_networks] + [current_signal]
      )

      # If this is the strongest among duplicates
      if current_signal == strongest_signal:
        signal_difference = strongest_signal - weakest_signal
        if signal_difference > 15: # Significant signal advantage
          risk_score += 25
          reasons.append(f"Strongest signal among {len(same_ssid_networks)+1} duplicates (+
{signal_difference}dB advantage)")
        else:
          risk_score += 15
          reasons.append(f"Strongest signal among {len(same_ssid_networks)+1} duplicates")

      # Check for signal strength inconsistencies
      if current_signal > -50 and len(same_ssid_networks) > 0:
        risk_score += 10
        reasons.append("Very strong signal with duplicate SSIDs")
```

```
    if current_signal > -30:
      risk_score += 20
      reasons.append("Extremely strong signal - possible high-power transmitter")
    elif current_signal > -40:
      risk_score += 10
      reasons.append("Very strong signal - monitor for consistency")

    return risk_score, reasons

  def analyze_ssid_pattern(self, ssid):
    """Enhanced SSID pattern analysis"""
    risk_score = 0
    reasons = []
    ssid_lower = ssid.lower()

    # Check for malicious keywords
    for keyword in self.malicious_keywords:
      if keyword in ssid_lower:
        risk_score += 15
        reasons.append(f"Contains suspicious keyword: '{keyword}'")

    # Check for legitimate brands (lower risk)
    is_legitimate_brand = any(brand in ssid_lower for brand in self.legitimate_brands)

    # Check suspicious patterns
    for pattern in self.suspicious_patterns:
      if re.match(pattern, ssid):
        risk_score += 20
        reasons.append("Suspicious SSID pattern detected")
        break

    # Length analysis
    if len(ssid) > 30:
      risk_score += 15
      reasons.append("Unusually long SSID (>30 chars)")
    elif len(ssid) < 2:
      risk_score += 20
      reasons.append("Invalid SSID length (<2 chars)")

    # Character diversity analysis
    if len(ssid) > 8:
      unique_chars = len(set(ssid.lower()))
      diversity_ratio = unique_chars / len(ssid)
      if diversity_ratio < 0.3:
        risk_score += 15
        reasons.append("Low character diversity - possible automated generation")

    # Trusted enterprise networks (negative risk)
    if any(prefix in ssid for prefix in self.trusted_enterprise_prefixes):
      risk_score -= 20
      reasons.append("Trusted enterprise network prefix")

    return risk_score, reasons

  def analyze_security_risk(self, security_type, ssid):
    """Enhanced security risk analysis"""
    risk_score = 0
    reasons = []
```

```
    security_risks = {
      'OPEN': 40,
      'WEP': 35,
      'UNKNOWN': 25,
      'WPA': 10,
      'WPA2': 5,
      'WPA2-ENTERPRISE': -10,
      'WPA3': -15,
      'ENTERPRISE': -10
    }

    risk_points = security_risks.get(security_type, 20)
    risk_score += risk_points

    if risk_points > 0:
      reasons.append(f"{security_type} security: +{risk_points} risk")
    else:
      reasons.append(f"{security_type} security: {risk_points} risk (safe)")

    # Special case: Open network with legitimate-sounding name
    if security_type == 'OPEN' and any(brand in ssid.lower() for brand in
self.legitimate_brands):
      risk_score += 10
      reasons.append("Open network impersonating legitimate brand")

    return risk_score, reasons

  def analyze_network_risk(self, network, all_networks):
    """Comprehensive network risk analysis"""
    ssid = network['ssid']
    risk_score = 0
    risk_reasons = []

    # 1. Security Analysis
    security_risk, security_reasons = self.analyze_security_risk(network['security'], ssid)
    risk_score += security_risk
    risk_reasons.extend(security_reasons)

    # 2. SSID Pattern Analysis
    ssid_risk, ssid_reasons = self.analyze_ssid_pattern(ssid)
    risk_score += ssid_risk
    risk_reasons.extend(ssid_reasons)

    # 3. Signal Analysis (Evil Twin Detection)
    signal_risk, signal_reasons = self.analyze_signal_risk(network, all_networks)
    risk_score += signal_risk
    risk_reasons.extend(signal_reasons)

    # 4. Manufacturer Analysis
    if network.get('bssid_mac'):
      manufacturer_risk, manufacturer_reason =
self.analyze_manufacturer_risk(network['bssid_mac'])
      risk_score += manufacturer_risk
      if manufacturer_risk != 0:
        risk_reasons.append(manufacturer_reason)

    # 5. Channel Analysis
    channel = network.get('channel', 0)
    if channel in [1, 6, 11]: # Standard non-overlapping channels
      risk_score -= 5
      risk_reasons.append("Standard channel usage")

    # Ensure risk score is within bounds
    risk_score = max(0, min(100, risk_score))
```

```
    # Determine risk level
    if risk_score >= 70:
      risk_level = "🔴 CRITICAL RISK"
    elif risk_score >= 50:
      risk_level = "🟠 HIGH RISK"
    elif risk_score >= 30:
      risk_level = "🟡 MEDIUM RISK"
    elif risk_score >= 15:
      risk_level = "🔵 LOW RISK"
    else:
      risk_level = "🟢 VERY LOW RISK"

    return {
      'ssid': ssid,
      'bssid': network.get('bssid_mac', 'Unknown'),
      'security': network.get('security', 'Unknown'),
      'signal_strength': network.get('signal_strength', -99),
      'signal_percent': network.get('signal_percent', 0),
      'channel': channel,
      'risk_level': risk_level,
      'risk_score': risk_score,
      'risk_reasons': risk_reasons,
      'duplicate_count': len([n for n in all_networks if n['ssid'] == ssid])
    }

  def scan_networks(self):
    """Perform comprehensive WiFi scan"""
    print("Scanning for WiFi networks...")

    try:
      result = subprocess.run(
        ['netsh', 'wlan', 'show', 'networks', 'mode=bssid'],
        capture_output=True, text=True, timeout=30
      )

      if result.returncode != 0:
        print("Failed to scan WiFi networks")
        return []

      # DEBUG: See what netsh is returning
      print("=== RAW NETSH OUTPUT ===")
      lines = result.stdout.split('\n')
      for i, line in enumerate(lines):
        if any(keyword in line.lower() for keyword in ['ssid', 'authentication',
'encryption']):
          print(f"{i:3d}: {line.strip()}")
      print("=== END DEBUG ===")

      # FIX: parse_netsh_output already returns flat list, no need to flatten again
      networks_for_analysis = self.parse_netsh_output(result.stdout)

      if not networks_for_analysis:
        print("No networks found")
        return []

      print(f" Found {len(networks_for_analysis)} access points")

      # Print security types for debugging
      security_counts = {}
      for network in networks_for_analysis:
        sec = network['security']
        security_counts[sec] = security_counts.get(sec, 0) + 1

      print("Security type breakdown:", security_counts)
```

```
      # Analyze each network
      analyzed_networks = []
      for network in networks_for_analysis:
        analysis = self.analyze_network_risk(network, networks_for_analysis)
        analyzed_networks.append(analysis)

      # Sort by risk score (highest first)
      analyzed_networks.sort(key=lambda x: x['risk_score'], reverse=True)

      return analyzed_networks

    except subprocess.TimeoutExpired:
      print("WiFi scan timed out")
      return []
    except Exception as e:
      print(f"Error during WiFi scan: {e}")
      return []
  def generate_report(self, networks):
    """Generate comprehensive scan report"""
    if not networks:
      return "No networks found for analysis."

    report = []
    report.append("📡 WiFi Security Scan Report")
    report.append("=" * 50)
    report.append(f"Scan Time: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
    report.append("")

    # Risk statistics
    critical_risk = sum(1 for n in networks if n['risk_score'] >= 70)
    high_risk = sum(1 for n in networks if 50 <= n['risk_score'] < 70)
    medium_risk = sum(1 for n in networks if 30 <= n['risk_score'] < 50)
    low_risk = sum(1 for n in networks if n['risk_score'] < 30)

    report.append("Risk Summary:")
    report.append(f" Critical Risk: {critical_risk}")
    report.append(f" High Risk: {high_risk}")
    report.append(f" Medium Risk: {medium_risk}")
    report.append(f" Low/Very Low Risk: {low_risk}")
    report.append(f" Total Networks: {len(networks)}")
    report.append("")

    # Top risky networks
    report.append("Top 5 Most Risky Networks:")
    report.append("-" * 40)

    for i, network in enumerate(networks[:5]):
      report.append(f"{i+1}. {network['ssid']}")
      report.append(f"  Risk: {network['risk_level']} ({network['risk_score']}/100)")
      report.append(f"  BSSID: {network['bssid']}")
      report.append(f"  Security: {network['security']}")
      report.append(f"  Signal: {network['signal_strength']} dBm
({network['signal_percent']}%)")
      report.append(f"  Channel: {network['channel']}")
      report.append("  Reasons:")
      for reason in network['risk_reasons'][:4]: # Show top 4 reasons
        report.append(f"   • {reason}")
      report.append("")
```

```
      # Evil twin detection summary
      duplicate_ssids = defaultdict(list)
      for network in networks:
        duplicate_ssids[network['ssid']].append(network)

      evil_twin_candidates = {ssid: nets for ssid, nets in duplicate_ssids.items() if len(nets)
  > 1}

      if evil_twin_candidates:
        report.append("Evil Twin Detection Summary:")
        report.append("-" * 35)
        for ssid, nets in list(evil_twin_candidates.items())[:3]:
          strongest = max(nets, key=lambda x: x['signal_strength'])
          report.append(f" '{ssid}': {len(nets)} APs, strongest:
  {strongest['signal_strength']}dBm")

      return "\n".join(report)

  # Example usage
  if __name__ == "__main__":
    scanner = WiFiScanner()
    networks = scanner.scan_networks()

    if networks:
      report = scanner.generate_report(networks)
      print(report)

      # Save report to file
      with open('wifi_scan_report.txt', 'w') as f:
        f.write(report)
      print("\nReport saved to 'wifi_scan_report.txt'")
```

traffic_analyzer.py

```
import time
from collections import defaultdict, deque
import numpy as np
from scapy.all import sniff, IP, TCP, UDP, DNS, DNSQR, Dot11, Dot11Beacon, Dot11Elt,
RadioTap, Ether
import socket
import threading
from datetime import datetime
import pandas as pd
import joblib
import warnings
warnings.filterwarnings('ignore')

try:
  import tensorflow as tf
  AI_AVAILABLE = True
except ImportError:
  AI_AVAILABLE = False
  print("Warning: TensorFlow not available. AI features disabled.")

class EvilTwinDetector:
  def __init__(self):
    self.traffic_model = None
    self.traffic_scaler = None
    self.model_loaded = False
    self.load_model()
```

```
  def load_model(self):
    """Load the trained model and scaler"""
    try:
      self.traffic_model = tf.keras.models.load_model('traffic_model.h5')
      self.traffic_scaler = joblib.load('traffic_scaler.pkl')
      self.model_loaded = True
      print("AI Model loaded successfully for evil twin detection")
    except Exception as e:
      print(f"Error loading model: {e}")
      print("Please make sure you've trained the model first using model.py")
      self.model_loaded = False

  def analyze_network_traffic(self, feature_dict):
    if not self.model_loaded:
      return {'error': "Model not loaded. Please train first."}
    try:
      # Convert feature dict to DataFrame
      feature_df = pd.DataFrame([feature_dict])

      # Ensure all expected features are present
      if hasattr(self.traffic_scaler, 'feature_names_in_'):
        for feature in self.traffic_scaler.feature_names_in_:
          if feature not in feature_df.columns:
            feature_df[feature] = 0
        feature_df = feature_df[self.traffic_scaler.feature_names_in_].fillna(0)

      # Clean the data
      for col in feature_df.columns:
        feature_df[col] = pd.to_numeric(feature_df[col], errors='coerce')
      feature_df = feature_df.replace([np.inf, -np.inf], 0)
      feature_df = feature_df.fillna(0)

      # Scale features
      feature_scaled = self.traffic_scaler.transform(feature_df)

      # Reshape for CNN/LSTM if needed
      if len(self.traffic_model.input_shape) == 3:
        feature_scaled = feature_scaled.reshape(feature_scaled.shape[0],
feature_scaled.shape[1], 1)

      # Make prediction
      prediction_prob = self.traffic_model.predict(feature_scaled, verbose=0)[0][0]

      # Interpret results
      if prediction_prob > 0.7:
        is_evil_twin = True
        safety_score = (1 - prediction_prob) * 100
      elif prediction_prob < 0.3:
        is_evil_twin = False
        safety_score = (1 - prediction_prob) * 100
      else:
        is_evil_twin = prediction_prob > 0.5
        safety_score = 50

      # Generate recommendation
      if is_evil_twin:
        if safety_score < 30:
          recommendation = "EVIL TWIN DETECTED! Disconnect immediately!"
        else:
          recommendation = "Suspicious network detected. Proceed with caution."
```

```
      else:
        if safety_score >= 80:
          recommendation = "Network appears safe."
        else:
          recommendation = "Network shows minor anomalies."

      return {
        'safety_score': round(safety_score, 2),
        'safety_level': "SAFE" if safety_score >= 70 else "CAUTION" if safety_score >= 50
else "UNSAFE",
        'recommendation': recommendation,
        'is_evil_twin': bool(is_evil_twin),
        'probability_evil_twin': f"{prediction_prob:.2%}",
        'probability_legitimate': f"{(1-prediction_prob):.2%}"
      }
    except Exception as e:
      return {'error': f"Analysis failed: {e}"}

class Flow:
  """Network flow analysis"""
  def __init__(self, packet, local_ip):
    self.packets = deque([packet], maxlen=1000)
    self.start_time = packet.time
    self.end_time = packet.time
    self.local_ip = local_ip
    self.protocol = 'TCP' if TCP in packet else 'UDP' if UDP in packet else 'Other'
    self.dns_queries = []
    self.ports = set()
    self.wireless_packets = []

  def add_packet(self, packet):
    self.packets.append(packet)
    self.end_time = packet.time

    # Check for wireless packets (for evil twin detection)
    if Dot11 in packet:
      self.wireless_packets.append(packet)

    if DNS in packet and DNSQR in packet:
      self.dns_queries.append(packet[DNSQR].qname.decode() if hasattr(packet[DNSQR].qname,
'decode') else str(packet[DNSQR].qname))

    if TCP in packet:
      self.ports.add(packet[TCP].sport)
      self.ports.add(packet[TCP].dport)
    elif UDP in packet:
      self.ports.add(packet[UDP].sport)
      self.ports.add(packet[UDP].dport)

  def get_wireless_features(self):
    """Extract wireless-specific features for evil twin detection"""
    if not self.wireless_packets:
      return {}

    beacon_count = 0
    ssids = set()
    channels = set()
    signal_strengths = []
    encryption_types = set()
```

```
    for packet in self.wireless_packets:
      if Dot11Beacon in packet:
        beacon_count += 1

        # Extract SSID
        if packet[Dot11Elt].ID == 0: # SSID
          try:
            ssid = packet[Dot11Elt].info.decode('utf-8', errors='ignore')
            if ssid and ssid.strip():
              ssids.add(ssid)
          except:
            pass

        # Extract channel
        if packet[Dot11Elt].ID == 3: # DS Parameter Set (channel)
          try:
            channels.add(packet[Dot11Elt].info[0])
          except:
            pass

        # Signal strength
        if RadioTap in packet:
          if hasattr(packet[RadioTap], 'dBm_AntSignal'):
            signal_strengths.append(packet[RadioTap].dBm_AntSignal)

      # Extract encryption info
      if Dot11Elt in packet:
        if packet[Dot11Elt].ID == 48: # RSN Information
          encryption_types.add('WPA2')
        elif packet[Dot11Elt].ID == 221: # Vendor Specific
          if b'WPA' in packet[Dot11Elt].info:
            encryption_types.add('WPA')

    return {
      'beacon_count': beacon_count,
      'unique_ssids': len(ssids),
      'unique_channels': len(channels),
      'avg_signal_strength': np.mean(signal_strengths) if signal_strengths else -100,
      'encryption_types': len(encryption_types),
      'ssid_changes': len(ssids) > 1 # Multiple SSIDs from same MAC
    }

  def get_features(self):
    """Extract features for AI analysis"""
    if len(self.packets) < 2:
      return None

    duration_sec = self.end_time - self.start_time
    if duration_sec == 0:
      duration_sec = 1e-6

    fwd_packets = [p for p in self.packets if IP in p and p[IP].src == self.local_ip]
    bwd_packets = [p for p in self.packets if IP in p and p[IP].dst == self.local_ip]

    fwd_lengths = [len(p) for p in fwd_packets]
    bwd_lengths = [len(p) for p in bwd_packets]

    tcp_fwd = [p for p in fwd_packets if TCP in p]
    tcp_bwd = [p for p in bwd_packets if TCP in p]

    # Get wireless features
    wireless_features = self.get_wireless_features()
```

```
    wireless_features = self.get_wireless_features()

    features = {
      'Flow Duration': duration_sec * 1_000_000,
      'Total Fwd Packets': len(fwd_packets),
      'Total Backward Packets': len(bwd_packets),
      'Total Length of Fwd Packets': sum(fwd_lengths),
      'Total Length of Bwd Packets': sum(bwd_lengths),

      'Fwd Packet Length Mean': np.mean(fwd_lengths) if fwd_lengths else 0,
      'Bwd Packet Length Mean': np.mean(bwd_lengths) if bwd_lengths else 0,
      'Fwd Packet Length Std': np.std(fwd_lengths) if len(fwd_lengths) > 1 else 0,
      'Bwd Packet Length Std': np.std(bwd_lengths) if len(bwd_lengths) > 1 else 0,
      'Fwd Packet Length Max': max(fwd_lengths) if fwd_lengths else 0,
      'Bwd Packet Length Max': max(bwd_lengths) if bwd_lengths else 0,
      'Fwd Packet Length Min': min(fwd_lengths) if fwd_lengths else 0,
      'Bwd Packet Length Min': min(bwd_lengths) if bwd_lengths else 0,

      'Flow Packets/s': len(self.packets) / duration_sec,
      'Fwd Packets/s': len(fwd_packets) / duration_sec,
      'Bwd Packets/s': len(bwd_packets) / duration_sec,
      'Flow Bytes/s': (sum(fwd_lengths) + sum(bwd_lengths)) / duration_sec,

      'Fwd PSH Flags': sum(1 for p in tcp_fwd if TCP in p and 'P' in p[TCP].flags),
      'Bwd PSH Flags': sum(1 for p in tcp_bwd if TCP in p and 'P' in p[TCP].flags),
      'Fwd URG Flags': sum(1 for p in tcp_fwd if TCP in p and 'U' in p[TCP].flags),
      'Bwd URG Flags': sum(1 for p in tcp_bwd if TCP in p and 'U' in p[TCP].flags),
      'Fwd FIN Flags': sum(1 for p in tcp_fwd if TCP in p and 'F' in p[TCP].flags),
      'Bwd FIN Flags': sum(1 for p in tcp_bwd if TCP in p and 'F' in p[TCP].flags),
      'Fwd SYN Flags': sum(1 for p in tcp_fwd if TCP in p and 'S' in p[TCP].flags),
      'Bwd SYN Flags': sum(1 for p in tcp_bwd if TCP in p and 'S' in p[TCP].flags),
      'Fwd RST Flags': sum(1 for p in tcp_fwd if TCP in p and 'R' in p[TCP].flags),
      'Bwd RST Flags': sum(1 for p in tcp_bwd if TCP in p and 'R' in p[TCP].flags),

      'Init_Win_bytes_forward': next((p[TCP].window for p in tcp_fwd if TCP in p), 0),
      'Init_Win_bytes_backward': next((p[TCP].window for p in tcp_bwd if TCP in p), 0),

      'bytes_ratio': sum(fwd_lengths) / sum(bwd_lengths) if sum(bwd_lengths) > 0 else 1,
      'traffic_asymmetry': abs((sum(fwd_lengths) / sum(bwd_lengths) if sum(bwd_lengths) > 0
else 1) - 1),
      'packets_ratio': len(fwd_packets) / len(bwd_packets) if len(bwd_packets) > 0 else 1,
      'avg_packet_size': (sum(fwd_lengths) + sum(bwd_lengths)) / len(self.packets),

      'avg_fwd_segment_size': np.mean(fwd_lengths) if fwd_lengths else 0,
      'avg_bwd_segment_size': np.mean(bwd_lengths) if bwd_lengths else 0,
      'fwd_header_length': sum(len(p[TCP]) if TCP in p else 0 for p in fwd_packets),
      'subflow_fwd_packets': len(fwd_packets) // 2,
      'subflow_bwd_packets': len(bwd_packets) // 2,
      'subflow_fwd_bytes': sum(fwd_lengths) // 2,
      'subflow_bwd_bytes': sum(bwd_lengths) // 2,

      'dns_query_count': len(self.dns_queries),
      'unique_ports': len(self.ports),

      # Evil twin specific features
      'beacon_frame_count': wireless_features.get('beacon_count', 0),
      'multiple_ssids': 1 if wireless_features.get('ssid_changes', False) else 0,
      'signal_strength_variance':
```

```
wireless_features.get('avg_signal_strength', -100),
      'channel_changes': wireless_features.get('unique_channels', 0),
      'encryption_inconsistencies': 1 if wireless_features.get('encryption_types', 0) > 1
else 0,
      'authentication_failures': sum(1 for p in self.packets if TCP in p and p[TCP].dport in
[80, 443] and 'R' in p[TCP].flags),
      'dns_anomalies': 1 if any('fake' in query.lower() or 'evil' in query.lower() or 'phish'
in query.lower() for query in self.dns_queries) else 0
    }
    return features
class TrafficAnalyzer:
  def __init__(self):
    self.flows = defaultdict(lambda: None)
    self.local_ip = self.get_local_ip()
    self.capture_stats = {
      'total_packets': 0,
      'total_flows': 0,
      'start_time': None,
      'end_time': None
    }

    # Initialize AI model for evil twin detection
    if AI_AVAILABLE:
      self.evil_twin_detector = EvilTwinDetector()
      self.ai_model_loaded = self.evil_twin_detector.model_loaded
    else:
      self.ai_model_loaded = False
      print("⚠️ TensorFlow not available - evil twin detection disabled")

  def get_local_ip(self):
    """Get local IP address"""
    try:
      s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
      s.connect(("8.8.8.8", 80))
      local_ip = s.getsockname()[0]
      s.close()
      return local_ip
    except:
      return "127.0.0.1"
  def get_flow_key(self, packet):
    """Generate unique flow key"""
    if IP not in packet:
      return None

    src_ip = packet[IP].src
    dst_ip = packet[IP].dst

    if TCP in packet:
      src_port = packet[TCP].sport
      dst_port = packet[TCP].dport
      protocol = 'TCP'
    elif UDP in packet:
      src_port = packet[UDP].sport
      dst_port = packet[UDP].dport
      protocol = 'UDP'
    else:
      return None

    if src_ip < dst_ip:
      key = f"{src_ip}:{src_port}-{dst_ip}:{dst_port}-{protocol}"
    else:
      key = f"{dst_ip}:{dst_port}-{src_ip}:{src_port}-{protocol}"

    return key
```

```
 def packet_handler(self, packet):
    """Handle incoming packets"""
    self.capture_stats['total_packets'] += 1

    if IP not in packet and Dot11 not in packet:
      return

    flow_key = self.get_flow_key(packet) if IP in packet else "wireless"
    if not flow_key:
      return

    if self.flows[flow_key] is None:
      self.flows[flow_key] = Flow(packet, self.local_ip)
      self.capture_stats['total_flows'] += 1
    else:
      self.flows[flow_key].add_packet(packet)

  def detect_evil_twin(self, features):
    """Use AI model to detect evil twin attacks"""
    if not self.ai_model_loaded:
      return {
        'error': 'AI model not loaded',
        'is_evil_twin': False,
        'safety_score': 0,
        'recommendation': 'AI model not available'
      }

    try:
      result = self.evil_twin_detector.analyze_network_traffic(features)
      return result
    except Exception as e:
      return {
        'error': f'AI analysis failed: {e}',
        'is_evil_twin': False,
        'safety_score': 0,
        'recommendation': 'Check model configuration'
      }

  def capture_traffic(self, duration=60, packet_count=10000):
    """Capture network traffic"""
    print(f"Starting traffic capture for {duration}s...")
    print("🔍 Evil twin detection enabled" if self.ai_model_loaded else "⚠️ Evil twin
detection disabled")

    self.capture_stats['start_time'] = datetime.now()

    stop_event = threading.Event()

    def stop_capture():
      time.sleep(duration)
      stop_event.set()

    timer_thread = threading.Thread(target=stop_capture)
    timer_thread.daemon = True
    timer_thread.start()

    # Capture both wired and wireless traffic
    sniff(prn=self.packet_handler, stop_filter=lambda x: stop_event.is_set(),
count=packet_count)

    self.capture_stats['end_time'] = datetime.now()

    print(f"Capture complete. Processed {self.capture_stats['total_packets']} packets,
{len(self.flows)} flows.")
    return self.extract_features()
```

```
  def extract_features(self):
     """Extract features from all flows and run evil twin detection"""
     features_list = []
     evil_twin_results = []
     valid_flows = 0

     for flow_key, flow in self.flows.items():
       if flow and len(flow.packets) >= 2:
         features = flow.get_features()
         if features:
           features_list.append(features)
           valid_flows += 1

           # Run evil twin detection on this flow
           if self.ai_model_loaded:
             detection_result = self.detect_evil_twin(features)
             detection_result['flow_key'] = flow_key
             detection_result['packet_count'] = len(flow.packets)
             evil_twin_results.append(detection_result)

     print(f"Extracted features from {valid_flows} valid flows.")

     # Analyze overall evil twin risk
     overall_risk = self.analyze_overall_risk(evil_twin_results)

     return {
       'features': features_list,
       'evil_twin_analysis': evil_twin_results,
       'overall_risk': overall_risk
     }

   def analyze_overall_risk(self, evil_twin_results):
     """Analyze overall network risk based on evil twin detection results"""
     if not evil_twin_results:
       return {
         'overall_safety_score': 100,
         'risk_level': 'LOW',
         'evil_twin_detected': False,
         'suspicious_flows': 0,
         'recommendation': 'No suspicious activity detected'
       }

     high_risk_flows = sum(1 for result in evil_twin_results
                 if result.get('is_evil_twin', False) and
                  result.get('safety_score', 100) < 30)

     medium_risk_flows = sum(1 for result in evil_twin_results
                  if result.get('safety_score', 100) < 70 and
                   not result.get('is_evil_twin', False))

     safety_scores = [result.get('safety_score', 100) for result in evil_twin_results
             if 'safety_score' in result and not isinstance(result.get('safety_score'), str)]

     avg_safety_score = np.mean(safety_scores) if safety_scores else 100

     if high_risk_flows > 0:
       risk_level = 'CRITICAL'
       recommendation = '🚨 EVIL TWIN DETECTED! Disconnect immediately!'
     elif medium_risk_flows > 2:
       risk_level = 'HIGH'
       recommendation = 'Multiple suspicious flows detected. Avoid sensitive activities.'
     elif avg_safety_score < 70:
       risk_level = 'MEDIUM'
```

```
      recommendation = 'Network shows some anomalies. Proceed with caution.'
    else:
      risk_level = 'LOW'
      recommendation = 'Network appears safe.'

    return {
      'overall_safety_score': round(avg_safety_score, 2),
      'risk_level': risk_level,
      'evil_twin_detected': high_risk_flows > 0,
      'suspicious_flows': high_risk_flows + medium_risk_flows,
      'high_risk_flows': high_risk_flows,
      'medium_risk_flows': medium_risk_flows,
      'recommendation': recommendation
    }

  def get_statistics(self):
    """Get capture statistics"""
    duration = (self.capture_stats['end_time'] -
self.capture_stats['start_time']).total_seconds()
    packets_per_second = self.capture_stats['total_packets'] / duration if duration > 0 else
0

    stats = {
      'capture_duration': f"{duration:.2f}s",
      'total_packets': self.capture_stats['total_packets'],
      'total_flows': len(self.flows),
      'packets_per_second': f"{packets_per_second:.2f}",
      'average_flow_length': f"{self.capture_stats['total_packets'] / len(self.flows) if
self.flows else 0:.2f}",
      'ai_model_loaded': self.ai_model_loaded
    }

    return stats

  def print_evil_twin_report(self, analysis_results):
    """Print a detailed evil twin detection report"""
    if 'evil_twin_analysis' not in analysis_results:
      return

    print("\n" + "="*60)
    print("🔍 EVIL TWIN DETECTION REPORT")
    print("="*60)

    overall_risk = analysis_results.get('overall_risk', {})

    print(f"Overall Safety Score: {overall_risk.get('overall_safety_score', 'N/A')}%")
    print(f"Risk Level: {overall_risk.get('risk_level', 'UNKNOWN')}")
    print(f"Evil Twin Detected: {'YES ' if overall_risk.get('evil_twin_detected') else 'No
'}")
    print(f"Suspicious Flows: {overall_risk.get('suspicious_flows', 0)}")
    print(f"Recommendation: {overall_risk.get('recommendation', 'N/A')}")

    if analysis_results['evil_twin_analysis']:
      print("\nDetailed Flow Analysis:")
      print("-" * 40)

      for i, result in enumerate(analysis_results['evil_twin_analysis'][:5]): # Show top 5
        if 'error' not in result:
          print(f"Flow {i+1}: Safety={result.get('safety_score', 'N/A')}% | "
             f"Evil Twin: {result.get('is_evil_twin', False)} | "
             f"Packets: {result.get('packet_count', 0)}")
```

```
if __name__ == "__main__":
  analyzer = TrafficAnalyzer()

  # Capture traffic for 30 seconds
  results = analyzer.capture_traffic(duration=30)

  # Print statistics
  stats = analyzer.get_statistics()
  print("\nCapture Statistics:")
  for key, value in stats.items():
    print(f" {key}: {value}")

  # Print evil twin report
  analyzer.print_evil_twin_report(results)
```