AI threat detection

Table of Contents

**No table of contents entries found.**

**Project Overview: AI-Powered Threat Detection System for Network Security**

This project is an **AI-driven system** designed to enhance network security by detecting anomalies, predicting potential threats, and automating alerts and mitigation strategies. It integrates multiple data sources, including **RIPE Atlas traceroute data** and **BGP/RPKI feeds**, to provide a comprehensive threat detection solution. Below is a detailed breakdown of the project:

**Key Features**

1. **Network Anomaly Detection**:
   * Analyzes traceroute data to identify unusual patterns in network paths.
   * Uses machine learning (Isolation Forest) to detect anomalies in real-time.
2. **BGP/RPKI Route Validation**:
   * Validates BGP routes against RPKI (Resource Public Key Infrastructure) data to detect potential hijacks or misconfigurations.
   * Uses a Random Forest classifier to predict suspicious BGP announcements.
3. **Threat Prediction**:
   * Combines traceroute anomalies and BGP hijack predictions to calculate an overall **threat score**.
   * Classifies threats into levels: **Low**, **Medium**, **High**, and **Critical**.
4. **Alert System**:
   * Automatically generates alerts when threats exceed a predefined threshold.
   * Sends notifications via **email** or **API** to relevant stakeholders.
   * Provides actionable **mitigation strategies** for detected threats.
5. **Reporting**:
   * Generates detailed reports summarizing detected anomalies, threats, and actions taken.

**Components**

The system is modular and consists of the following components:

1. **Atlas Data Analyzer**:
   * Fetches and analyzes RIPE Atlas traceroute data.
   * Extracts features like hop count, RTT (Round-Trip Time), IP changes, and ASN changes.
   * Uses **Isolation Forest** for anomaly detection.
2. **BGP/RPKI Validator**:
   * Fetches BGP data and validates it against RPKI ROA (Route Origin Authorization) records.
   * Predicts potential BGP hijacks using a **Random Forest Classifier**.
3. **Threat Predictor**:
   * Integrates data from traceroute analysis and BGP validation.
   * Calculates a **threat score** based on the severity of anomalies and hijack predictions.
4. **Alert System**:
   * Generates alerts for high-threat events.
   * Sends notifications via email or API.
   * Suggests mitigation strategies like BGP route verification and increased monitoring.
5. **Threat Detection System**:
   * The main class that orchestrates all components.
   * Initializes the system, runs detection cycles, and generates reports.

**How It Works**

1. **Data Collection**:
   * Fetches traceroute data from RIPE Atlas and BGP data from route views or other sources.
   * Validates BGP routes against RPKI data.
2. **Feature Extraction**:
   * Extracts relevant features from traceroute and BGP data for analysis.
3. **Anomaly Detection**:
   * Uses machine learning models to detect anomalies in traceroute data and suspicious BGP announcements.
4. **Threat Scoring**:
   * Combines results from traceroute and BGP analysis to calculate an overall threat score.
5. **Alerting and Mitigation**:
   * Sends alerts for high-threat events and suggests mitigation strategies.
6. **Reporting**:
   * Generates a summary report of detected threats and actions taken.

**Code Structure**

The code is organized into several classes, each responsible for a specific component of the system:

1. **AtlasDataAnalyzer**:
   * Handles fetching and analyzing traceroute data.
   * Trains and uses an Isolation Forest model for anomaly detection.
2. **BGPRPKIValidator**:
   * Validates BGP routes against RPKI data.
   * Predicts potential hijacks using a Random Forest Classifier.
3. **ThreatPredictor**:
   * Integrates data from traceroute and BGP analysis.
   * Calculates threat scores and classifies threats.
4. **AlertSystem**:
   * Generates alerts and sends notifications.
   * Suggests mitigation strategies.
5. **ThreatDetectionSystem**:
   * The main class that initializes and runs the system.

**Main code**

**"""**

**AI-Powered Threat Detection System for Network Security**

**This system analyzes Atlas traceroute data and BGP/RPKI feeds to:**

**1. Detect network anomalies**

**2. Predict potential threats**

**3. Automate alerts and mitigation strategies**

**Components:**

**- Atlas Data Analyzer (traceroute anomaly detection)**

**- BGP/RPKI Validator (route validation)**

**- Machine Learning Predictor (threat prediction)**

**- Alert System (notification mechanism)**

**"""**

import pandas as pd

import numpy as np

import requests

import json

import time

import logging

from datetime import datetime

from sklearn.ensemble import IsolationForest, RandomForestClassifier

from sklearn.preprocessing import StandardScaler

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import classification\_report

# Configure logging

logging.basicConfig(

level=logging.INFO,

format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',

handlers=[

logging.FileHandler("threat\_detection.log"),

logging.StreamHandler()

]

)

logger = logging.getLogger("ai\_threat\_detection")

class AtlasDataAnalyzer:

"""Component for analyzing RIPE Atlas traceroute data to detect anomalies"""

def \_\_init\_\_(self, api\_key=None):

self.api\_key = api\_key

self.base\_url = "https://atlas.ripe.net/api/v2/"

self.scaler = StandardScaler()

self.model = IsolationForest(contamination=0.05, random\_state=42)

self.historical\_data = None

def fetch\_traceroute\_data(self, measurement\_ids, start\_time=None, end\_time=None):

"""Fetch traceroute data from RIPE Atlas API"""

if not isinstance(measurement\_ids, list):

measurement\_ids = [measurement\_ids]

results = []

for measurement\_id in measurement\_ids:

url = f"{self.base\_url}measurements/{measurement\_id}/results/"

params = {}

if start\_time:

params["start"] = start\_time

if end\_time:

params["stop"] = end\_time

if self.api\_key:

params["key"] = self.api\_key

try:

response = requests.get(url, params=params)

response.raise\_for\_status()

results.extend(response.json())

logger.info(f"Successfully fetched data for measurement {measurement\_id}")

except requests.exceptions.RequestException as e:

logger.error(f"Error fetching data for measurement {measurement\_id}: {e}")

return results

def extract\_features(self, traceroute\_data):

"""Extract relevant features from traceroute data"""

features = []

for result in traceroute\_data:

# Basic features

rtt\_values = []

hop\_count = 0

ip\_changes = 0

prev\_ip = None

asn\_changes = 0

prev\_asn = None

# Process each hop

if "result" in result:

for hop in result.get("result", []):

hop\_count += 1

# Extract RTT values

for reply in hop.get("result", []):

if "rtt" in reply and reply["rtt"] is not None:

rtt\_values.append(reply["rtt"])

# Count IP changes

current\_ip = None

for reply in hop.get("result", []):

if "from" in reply:

current\_ip = reply["from"]

break

if current\_ip and prev\_ip and current\_ip != prev\_ip:

ip\_changes += 1

prev\_ip = current\_ip

# Count ASN changes (would require ASN lookup in a real implementation)

# Simplified version for demonstration

current\_asn = hop.get("asn", None)

if current\_asn and prev\_asn and current\_asn != prev\_asn:

asn\_changes += 1

prev\_asn = current\_asn

# Compute aggregate features

avg\_rtt = np.mean(rtt\_values) if rtt\_values else 0

min\_rtt = np.min(rtt\_values) if rtt\_values else 0

max\_rtt = np.max(rtt\_values) if rtt\_values else 0

rtt\_std = np.std(rtt\_values) if len(rtt\_values) > 1 else 0

features.append({

"timestamp": result.get("timestamp", 0),

"hop\_count": hop\_count,

"avg\_rtt": avg\_rtt,

"min\_rtt": min\_rtt,

"max\_rtt": max\_rtt,

"rtt\_std": rtt\_std,

"ip\_changes": ip\_changes,

"asn\_changes": asn\_changes,

"probe\_id": result.get("prb\_id", 0)

})

return pd.DataFrame(features)

def train\_anomaly\_detector(self, historical\_data):

"""Train the anomaly detection model on historical traceroute data"""

if historical\_data.empty:

logger.warning("No historical data provided for training")

return False

self.historical\_data = historical\_data

# Select numerical features

numeric\_features = historical\_data.select\_dtypes(include=[np.number])

exclude\_cols = ["timestamp", "probe\_id"]

feature\_cols = [col for col in numeric\_features.columns if col not in exclude\_cols]

# Scale features

X = self.scaler.fit\_transform(historical\_data[feature\_cols])

# Train isolation forest

self.model.fit(X)

logger.info("Anomaly detection model trained successfully")

return True

def detect\_anomalies(self, new\_data):

"""Detect anomalies in new traceroute data"""

if self.historical\_data is None:

logger.error("Model not trained. Call train\_anomaly\_detector first.")

return None

# Select numeric features

numeric\_features = new\_data.select\_dtypes(include=[np.number])

exclude\_cols = ["timestamp", "probe\_id"]

feature\_cols = [col for col in numeric\_features.columns if col not in exclude\_cols]

# Scale features

X = self.scaler.transform(new\_data[feature\_cols])

# Predict anomalies

# Isolation Forest: -1 for anomalies, 1 for normal

predictions = self.model.predict(X)

anomaly\_scores = self.model.decision\_function(X)

# Add results to the dataframe

new\_data["is\_anomaly"] = [1 if pred == -1 else 0 for pred in predictions]

new\_data["anomaly\_score"] = anomaly\_scores

logger.info(f"Detected {np.sum(new\_data['is\_anomaly'])} anomalies out of {len(new\_data)} traceroutes")

return new\_data

class BGPRPKIValidator:

"""Component for validating BGP routes against RPKI data"""

def \_\_init\_\_(self, api\_key=None):

self.api\_key = api\_key

self.rpki\_cache = {}

self.bgp\_data = None

self.model = RandomForestClassifier(random\_state=42)

def fetch\_bgp\_data(self, source="routeviews", timeframe="latest"):

"""Fetch BGP data from a specified source"""

# In a real implementation, this would connect to a BGP data source

# For demonstration, we'll create mock data

mock\_bgp\_data = []

current\_time = int(time.time())

# Create 100 sample BGP announcements

for i in range(100):

prefix = f"192.{i % 255}.{(i\*3) % 255}.0/24"

origin\_asn = 64500 + (i % 100)

# Introduce some potential anomalies

if i % 20 == 0:

# Potentially suspicious ASN

origin\_asn = 65000 + (i % 10)

# Create path with length variation

path\_length = 3 + (i % 5)

as\_path = [64000 - j for j in range(path\_length)]

as\_path.append(origin\_asn)

entry = {

"timestamp": current\_time - (i \* 60),

"prefix": prefix,

"origin\_asn": origin\_asn,

"as\_path": as\_path,

"next\_hop": f"10.0.0.{i % 255}",

"communities": [f"64500:{i}", f"64500:{i+100}"],

}

mock\_bgp\_data.append(entry)

self.bgp\_data = pd.DataFrame(mock\_bgp\_data)

logger.info(f"Fetched {len(mock\_bgp\_data)} BGP announcements")

return self.bgp\_data

def fetch\_rpki\_data(self):

"""Fetch RPKI ROA data"""

# In a real implementation, this would fetch ROA data from RPKI validators

# For demonstration, we'll create mock data

mock\_roa\_data = []

# Create matching ROAs for most of our BGP data

for i in range(80):

prefix = f"192.{i % 255}.{(i\*3) % 255}.0/24"

max\_length = 24

asn = 64500 + (i % 100)

# Introduce some invalid ROAs

if i % 20 == 0:

# ROA with different ASN than what we'll see in BGP

asn = 64600 + (i % 100)

roa = {

"prefix": prefix,

"max\_length": max\_length,

"asn": asn,

"ta": "RIPE NCC" # Trust Anchor

}

mock\_roa\_data.append(roa)

# Cache the ROA data

for roa in mock\_roa\_data:

prefix = roa["prefix"]

if prefix not in self.rpki\_cache:

self.rpki\_cache[prefix] = []

self.rpki\_cache[prefix].append(roa)

logger.info(f"Fetched {len(mock\_roa\_data)} RPKI ROAs")

return mock\_roa\_data

def validate\_routes(self, bgp\_data=None):

"""Validate BGP routes against RPKI data"""

if bgp\_data is None:

bgp\_data = self.bgp\_data

if bgp\_data is None:

logger.error("No BGP data available. Call fetch\_bgp\_data first.")

return None

if not self.rpki\_cache:

self.fetch\_rpki\_data()

validated\_routes = []

for \_, route in bgp\_data.iterrows():

prefix = route["prefix"]

origin\_asn = route["origin\_asn"]

validation\_state = "unknown"

matching\_roas = []

# Check if we have ROAs for this prefix

if prefix in self.rpki\_cache:

matching\_roas = self.rpki\_cache[prefix]

# Check if any ROA validates this announcement

for roa in matching\_roas:

if roa["asn"] == origin\_asn:

# Check prefix length

announced\_prefix\_length = int(prefix.split("/")[1])

if announced\_prefix\_length <= roa["max\_length"]:

validation\_state = "valid"

break

# If we found ROAs but none validated, it's invalid

if matching\_roas and validation\_state != "valid":

validation\_state = "invalid"

route\_data = route.to\_dict()

route\_data["validation\_state"] = validation\_state

route\_data["matching\_roas"] = len(matching\_roas)

validated\_routes.append(route\_data)

validated\_df = pd.DataFrame(validated\_routes)

# Log validation summary

summary = validated\_df["validation\_state"].value\_counts()

logger.info(f"Route validation summary: {summary.to\_dict()}")

return validated\_df

def extract\_features(self, validated\_routes):

"""Extract features for BGP hijack prediction model"""

features = []

for \_, route in validated\_routes.iterrows():

# Basic features

prefix\_length = int(route["prefix"].split("/")[1])

as\_path\_length = len(route["as\_path"])

origin\_asn = route["origin\_asn"]

# Convert validation state to numeric

validation\_numeric = {

"valid": 2,

"unknown": 1,

"invalid": 0

}[route["validation\_state"]]

# More advanced features

is\_private\_asn = 1 if (64512 <= origin\_asn <= 65534) else 0

feature\_dict = {

"timestamp": route["timestamp"],

"prefix\_length": prefix\_length,

"as\_path\_length": as\_path\_length,

"validation\_state\_numeric": validation\_numeric,

"matching\_roas": route["matching\_roas"],

"is\_private\_asn": is\_private\_asn,

"origin\_asn": origin\_asn,

}

features.append(feature\_dict)

return pd.DataFrame(features)

def train\_hijack\_detector(self, labeled\_data):

"""Train the BGP hijack prediction model"""

# In a real system, you would have labeled data with known hijacks

# For demonstration, we'll create synthetic labels

# Synthetic labels: mark all invalid routes and some unknowns as potential hijacks

labeled\_data["is\_suspicious"] = (

(labeled\_data["validation\_state\_numeric"] == 0) |

((labeled\_data["validation\_state\_numeric"] == 1) &

(labeled\_data["is\_private\_asn"] == 1))

).astype(int)

# Features for training

feature\_cols = [

"prefix\_length", "as\_path\_length", "validation\_state\_numeric",

"matching\_roas", "is\_private\_asn"

]

X = labeled\_data[feature\_cols]

y = labeled\_data["is\_suspicious"]

# Train-test split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X, y, test\_size=0.3, random\_state=42

)

# Train the model

self.model.fit(X\_train, y\_train)

# Evaluate

train\_score = self.model.score(X\_train, y\_train)

test\_score = self.model.score(X\_test, y\_test)

logger.info(f"BGP hijack detector trained. Train accuracy: {train\_score:.3f}, Test accuracy: {test\_score:.3f}")

# Feature importance

importances = self.model.feature\_importances\_

feature\_importance = {feature: importance for feature, importance in zip(feature\_cols, importances)}

logger.info(f"Feature importance: {feature\_importance}")

return self.model

def predict\_hijacks(self, route\_features):

"""Predict potential BGP hijacks"""

if not hasattr(self.model, 'predict'):

logger.error("Model not trained. Call train\_hijack\_detector first.")

return None

feature\_cols = [

"prefix\_length", "as\_path\_length", "validation\_state\_numeric",

"matching\_roas", "is\_private\_asn"

]

X = route\_features[feature\_cols]

predictions = self.model.predict(X)

probabilities = self.model.predict\_proba(X)[:, 1] # Probability of class 1 (suspicious)

route\_features["predicted\_hijack"] = predictions

route\_features["hijack\_probability"] = probabilities

hijack\_count = np.sum(predictions)

logger.info(f"Detected {hijack\_count} potential hijacks out of {len(route\_features)} routes")

return route\_features

class ThreatPredictor:

"""Component for integrating multiple data sources and predicting threats"""

def \_\_init\_\_(self):

self.atlas\_analyzer = AtlasDataAnalyzer()

self.bgp\_validator = BGPRPKIValidator()

self.combined\_features = None

def integrate\_data(self, traceroute\_analysis, bgp\_analysis):

"""Integrate data from multiple sources"""

# In a real system, you would need to correlate different data sources

# This is a simplified demonstration

if traceroute\_analysis is None:

logger.error("traceroute\_analysis is None in integrate\_data")

# Create an empty DataFrame with required columns

traceroute\_analysis = pd.DataFrame(columns=["timestamp", "is\_anomaly", "anomaly\_score",

"hop\_count", "avg\_rtt", "ip\_changes"])

if bgp\_analysis is None:

logger.error("bgp\_analysis is None in integrate\_data")

# Create an empty DataFrame with required columns

bgp\_analysis = pd.DataFrame(columns=["timestamp", "predicted\_hijack", "hijack\_probability",

"validation\_state\_numeric", "as\_path\_length", "is\_private\_asn"])

# Group traceroute anomalies by timestamp (rounding to nearest hour)

traceroute\_analysis["hour"] = np.floor(traceroute\_analysis["timestamp"] / 3600) \* 3600

traceroute\_grouped = traceroute\_analysis.groupby("hour").agg({

"is\_anomaly": "sum",

"anomaly\_score": "mean",

"hop\_count": "mean",

"avg\_rtt": "mean",

"ip\_changes": "mean"

}).reset\_index()

# Group BGP data by timestamp hour

bgp\_analysis["hour"] = np.floor(bgp\_analysis["timestamp"] / 3600) \* 3600

bgp\_grouped = bgp\_analysis.groupby("hour").agg({

"predicted\_hijack": "sum",

"hijack\_probability": "mean",

"validation\_state\_numeric": "mean",

"as\_path\_length": "mean",

"is\_private\_asn": "sum"

}).reset\_index()

# Merge the datasets on hour

combined = pd.merge(

traceroute\_grouped, bgp\_grouped,

on="hour", how="outer",

suffixes=("\_trace", "\_bgp")

).fillna(0)

self.combined\_features = combined

logger.info(f"Integrated {len(traceroute\_grouped)} traceroute periods with {len(bgp\_grouped)} BGP periods")

return combined

def calculate\_threat\_score(self, combined\_data=None):

"""Calculate overall threat score based on integrated data"""

if combined\_data is None:

combined\_data = self.combined\_features

if combined\_data is None:

logger.error("No combined data available. Call integrate\_data first.")

return None

# Define weights for different components

weights = {

"is\_anomaly": 0.2,

"anomaly\_score": -0.15, # Lower is more anomalous

"predicted\_hijack": 0.3,

"hijack\_probability": 0.25,

"validation\_state\_numeric": -0.1 # Higher is more valid

}

# Calculate normalized threat score

combined\_data["threat\_score"] = 0

for feature, weight in weights.items():

if feature in combined\_data.columns:

# Normalize the feature to [0, 1] range

min\_val = combined\_data[feature].min()

max\_val = combined\_data[feature].max()

if max\_val > min\_val:

normalized = (combined\_data[feature] - min\_val) / (max\_val - min\_val)

else:

normalized = 0

combined\_data["threat\_score"] += weight \* normalized

# Scale to 0-100 range

min\_score = combined\_data["threat\_score"].min()

max\_score = combined\_data["threat\_score"].max()

if max\_score > min\_score:

combined\_data["threat\_score"] = (

(combined\_data["threat\_score"] - min\_score) / (max\_score - min\_score) \* 100

)

# Classify threat levels

conditions = [

combined\_data["threat\_score"] < 20,

(combined\_data["threat\_score"] >= 20) & (combined\_data["threat\_score"] < 50),

(combined\_data["threat\_score"] >= 50) & (combined\_data["threat\_score"] < 80),

combined\_data["threat\_score"] >= 80

]

choices = ["Low", "Medium", "High", "Critical"]

combined\_data["threat\_level"] = np.select(conditions, choices, default="Unknown")

logger.info("Threat scores calculated")

level\_summary = combined\_data["threat\_level"].value\_counts()

logger.info(f"Threat level summary: {level\_summary.to\_dict()}")

return combined\_data

class AlertSystem:

"""Component for sending alerts and initiating mitigations"""

def \_\_init\_\_(self, email\_config=None, api\_config=None):

self.email\_config = email\_config or {}

self.api\_config = api\_config or {}

self.alert\_history = []

self.threshold = 50 # Default threshold for alerts

def set\_threshold(self, threshold):

"""Set the threshold for generating alerts"""

self.threshold = threshold

logger.info(f"Alert threshold set to {threshold}")

def generate\_alerts(self, threat\_data):

"""Generate alerts based on threat data"""

alerts = []

for \_, event in threat\_data.iterrows():

if event["threat\_score"] >= self.threshold:

alert = {

"timestamp": datetime.fromtimestamp(event["hour"]),

"threat\_score": event["threat\_score"],

"threat\_level": event["threat\_level"],

"traceroute\_anomalies": event.get("is\_anomaly", 0),

"bgp\_hijacks": event.get("predicted\_hijack", 0),

"details": {

"traceroute": {

"anomaly\_score": event.get("anomaly\_score", 0),

"hop\_count": event.get("hop\_count", 0),

"avg\_rtt": event.get("avg\_rtt", 0),

"ip\_changes": event.get("ip\_changes", 0)

},

"bgp": {

"hijack\_probability": event.get("hijack\_probability", 0),

"validation\_state": event.get("validation\_state\_numeric", 0),

"as\_path\_length": event.get("as\_path\_length", 0),

"private\_asns": event.get("is\_private\_asn", 0)

}

},

"alert\_id": f"ALERT-{int(time.time())}-{len(self.alert\_history)}"

}

alerts.append(alert)

self.alert\_history.append(alert)

logger.info(f"Generated {len(alerts)} alerts")

return alerts

def send\_email\_alerts(self, alerts, recipients=None):

"""Send email alerts (simulated)"""

if not alerts:

logger.info("No alerts to send via email")

return

recipients = recipients or self.email\_config.get("default\_recipients", ["admin@example.com"])

for alert in alerts:

# In a real implementation, this would send an actual email

logger.info(f"[SIMULATED EMAIL] To: {recipients}, Subject: {alert['threat\_level']} Threat Alert - Score: {alert['threat\_score']:.1f}")

logger.info(f"[SIMULATED EMAIL] Body: Detected {alert['traceroute\_anomalies']} traceroute anomalies and {alert['bgp\_hijacks']} potential BGP hijacks at {alert['timestamp']}")

def send\_api\_notifications(self, alerts, endpoints=None):

"""Send API notifications to ISPs or other systems (simulated)"""

if not alerts:

logger.info("No alerts to send via API")

return

endpoints = endpoints or self.api\_config.get("notification\_endpoints", ["https://api.example.com/notifications"])

for alert in alerts:

# Filter for high-severity alerts for API notifications

if alert["threat\_level"] in ["High", "Critical"]:

# In a real implementation, this would make API calls

for endpoint in endpoints:

logger.info(f"[SIMULATED API CALL] POST to {endpoint}")

logger.info(f"[SIMULATED API CALL] Data: {json.dumps(alert)}")

def suggest\_mitigations(self, alerts):

"""Suggest mitigations based on alert details"""

mitigations = []

for alert in alerts:

mitigation = {

"alert\_id": alert["alert\_id"],

"timestamp": alert["timestamp"],

"threat\_level": alert["threat\_level"],

"actions": []

}

# Suggest actions based on the type of threat

if alert["bgp\_hijacks"] > 0:

mitigation["actions"].append({

"type": "BGP\_MITIGATION",

"description": "Potential BGP hijack detected. Verify and contact affected ISPs.",

"priority": "High" if alert["threat\_level"] in ["High", "Critical"] else "Medium",

"automated": False

})

if alert["traceroute\_anomalies"] > 0:

mitigation["actions"].append({

"type": "TRACEROUTE\_INVESTIGATION",

"description": "Network path anomalies detected. Investigate routing changes.",

"priority": "Medium",

"automated": False

})

# Add general actions for all threats

mitigation["actions"].append({

"type": "MONITORING",

"description": "Increase monitoring frequency for affected networks",

"priority": "Low",

"automated": True

})

mitigations.append(mitigation)

logger.info(f"Generated {len(mitigations)} mitigation plans")

return mitigations

class ThreatDetectionSystem:

"""Main class integrating all components of the threat detection system"""

def \_\_init\_\_(self, config=None):

self.config = config or {}

self.atlas\_analyzer = AtlasDataAnalyzer(api\_key=self.config.get("95979aa4-6398-43ec-a455-50334269a44c"))

self.bgp\_validator = BGPRPKIValidator(api\_key=self.config.get("bgp\_api\_key"))

self.threat\_predictor = ThreatPredictor()

self.alert\_system = AlertSystem(

email\_config=self.config.get("email\_config"),

api\_config=self.config.get("api\_config")

)

def initialize(self):

"""Initialize the system with historical data and model training"""

logger.info("Initializing AI Threat Detection System")

# Initialize Atlas data analyzer

# In a real system, you would fetch historical data

historical\_traceroutes = self.atlas\_analyzer.fetch\_traceroute\_data(

measurement\_ids=[1234, 5678], # Example IDs

start\_time=int(time.time()) - (30 \* 24 \* 3600) # Last 30 days

)

historical\_features = self.atlas\_analyzer.extract\_features(historical\_traceroutes)

self.atlas\_analyzer.train\_anomaly\_detector(historical\_features)

# Initialize BGP validator

historical\_bgp = self.bgp\_validator.fetch\_bgp\_data()

self.bgp\_validator.fetch\_rpki\_data()

validated\_routes = self.bgp\_validator.validate\_routes(historical\_bgp)

route\_features = self.bgp\_validator.extract\_features(validated\_routes)

self.bgp\_validator.train\_hijack\_detector(route\_features)

logger.info("System initialization complete")

def run\_detection\_cycle(self):

"""Run a complete detection cycle"""

logger.info("Starting threat detection cycle")

# Fetch and analyze new traceroute data

try:

new\_traceroutes = self.atlas\_analyzer.fetch\_traceroute\_data(

measurement\_ids=[1234, 5678], # Example IDs

start\_time=int(time.time()) - (24 \* 3600) # Last 24 hours

)

if not new\_traceroutes:

logger.warning("No traceroute data returned")

traceroute\_analysis = None

else:

traceroute\_features = self.atlas\_analyzer.extract\_features(new\_traceroutes)

traceroute\_analysis = self.atlas\_analyzer.detect\_anomalies(traceroute\_features)

except Exception as e:

logger.error(f"Error in traceroute analysis: {e}")

traceroute\_analysis = None

# Fetch and analyze new BGP data

try:

new\_bgp = self.bgp\_validator.fetch\_bgp\_data()

if new\_bgp is None or new\_bgp.empty:

logger.warning("No BGP data returned")

bgp\_analysis = None

else:

validated\_routes = self.bgp\_validator.validate\_routes(new\_bgp)

route\_features = self.bgp\_validator.extract\_features(validated\_routes)

bgp\_analysis = self.bgp\_validator.predict\_hijacks(route\_features)

except Exception as e:

logger.error(f"Error in BGP analysis: {e}")

bgp\_analysis = None

# Integrate data and predict threats

integrated\_data = self.threat\_predictor.integrate\_data(traceroute\_analysis, bgp\_analysis)

threat\_assessment = self.threat\_predictor.calculate\_threat\_score(integrated\_data)

# Generate alerts and mitigations

alerts = self.alert\_system.generate\_alerts(threat\_assessment)

mitigations = self.alert\_system.suggest\_mitigations(alerts)

# Send notifications

self.alert\_system.send\_email\_alerts(alerts)

self.alert\_system.send\_api\_notifications(alerts)

# Prepare results

results = {

"run\_time": datetime.now().strftime("%Y-%m-%d %H:%M:%S"),

"traceroute\_anomalies": traceroute\_analysis["is\_anomaly"].sum(),

"bgp\_hijacks": bgp\_analysis["predicted\_hijack"].sum(),

"threat\_levels": threat\_assessment["threat\_level"].value\_counts().to\_dict(),

"alerts\_generated": len(alerts),

"mitigations\_suggested": len(mitigations)

}

logger.info(f"Threat detection cycle completed: {results}")

return results

def generate\_report(self, results):

"""Generate a summary report of the detection cycle"""

report = f"""

AI-Powered Threat Detection System - Summary Report

===================================================

Generated: {results['run\_time']}

Detection Results:

-----------------

Traceroute Anomalies Detected: {results['traceroute\_anomalies']}

Potential BGP Hijacks Detected: {results['bgp\_hijacks']}

Threat Levels:

-------------

{' | '.join([f"{level}: {count}" for level, count in results['threat\_levels'].items()])}

Actions Taken:

-------------

Alerts Generated: {results['alerts\_generated']}

Mitigation Plans Suggested: {results['mitigations\_suggested']}

System Status: Operational

"""

logger.info("Report generated successfully")

return report

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

# Configuration

config = {

"atlas\_api\_key": "your\_atlas\_api\_key\_here",

"bgp\_api\_key": "your\_bgp\_api\_key\_here",

"email\_config": {

"default\_recipients": ["security-team@example.com", "noc@example.com"],

"smtp\_server": "smtp.example.com",

"smtp\_port": 587,

"smtp\_user": "alerts@example.com",

"smtp\_password": "password123"

},

"api\_config": {

"notification\_endpoints": [

"https://api.example.com/security-notifications",

"https://api.ispprovider.com/route-security/alerts"

]

}

}

# Initialize the system

system = ThreatDetectionSystem(config)

system.initialize()

# Run a detection cycle

results = system.run\_detection\_cycle()

# Generate and print report

report = system.generate\_report(results)

print(report)

# Command-line interface

def main():

"""Command-line interface for the threat detection system"""

import argparse

parser = argparse.ArgumentParser(description="AI-Powered Threat Detection System for Network Security")

parser.add\_argument("--config", help="Path to configuration file", default="config.json")

parser.add\_argument("--mode", choices=["initialize", "detect", "report", "full"], default="full",

help="Operation mode (initialize, detect, report, or full cycle)")

parser.add\_argument("--alert-threshold", type=float, default=50.0,

help="Threshold for generating alerts (0-100)")

parser.add\_argument("--output", help="Output file for report", default="threat\_report.txt")

parser.add\_argument("--verbose", action="store\_true", help="Enable verbose logging")

args = parser.parse\_args()

# Set logging level based on verbose flag

if args.verbose:

logger.setLevel(logging.DEBUG)

# Load configuration

try:

with open(args.config, "r") as f:

config = json.load(f)

except FileNotFoundError:

logger.warning(f"Configuration file {args.config} not found. Using default configuration.")

config = {}

# Create and configure the system

system = ThreatDetectionSystem(config)

# Set alert threshold

system.alert\_system.set\_threshold(args.alert\_threshold)

# Execute selected mode

if args.mode in ["initialize", "full"]:

system.initialize()

if args.mode in ["detect", "full"]:

results = system.run\_detection\_cycle()

if args.mode in ["report", "full"]:

if "results" not in locals():

# If we didn't run detection, use dummy results

results = {

"run\_time": datetime.now().strftime("%Y-%m-%d %H:%M:%S"),

"traceroute\_anomalies": 0,

"bgp\_hijacks": 0,

"threat\_levels": {"Low": 0, "Medium": 0, "High": 0, "Critical": 0},

"alerts\_generated": 0,

"mitigations\_suggested": 0

}

report = system.generate\_report(results)

# Output report

if args.output:

with open(args.output, "w") as f:

f.write(report)

logger.info(f"Report written to {args.output}")

else:

print(report)

logger.info("AI Threat Detection System execution completed")

if \_\_name\_\_ == "\_\_main\_\_":

main()

### ****Enhancements for the AI-Powered Threat Detection System****

The current system is a solid foundation for detecting network anomalies and potential threats. However, to make it more robust and scalable, we can implement the following enhancements:

### ****1. Real-Time Monitoring****

To enable real-time threat detection, we need to integrate the system with live data feeds. Here's how we can achieve this:

#### **Implementation Steps**:

1. **Streaming Data Integration**:
   * Use APIs or streaming platforms (e.g., Kafka, Apache Flink) to ingest live traceroute and BGP data.
   * Modify the fetch\_traceroute\_data and fetch\_bgp\_data methods to support real-time data fetching.
2. **Continuous Detection Cycle**:
   * Implement a scheduler (e.g., using APScheduler or Celery) to run the detection cycle at regular intervals (e.g., every 5 minutes).
   * Example:

from apscheduler.schedulers.background import BackgroundScheduler

def run\_system():

system = ThreatDetectionSystem(config)

system.initialize()

while True:

system.run\_detection\_cycle()

time.sleep(300) # Run every 5 minutes

scheduler = BackgroundScheduler()

scheduler.add\_job(run\_system, 'interval', minutes=5)

scheduler.start()

1. **Real-Time Alerts**:
   * Use WebSocket or push notifications for real-time alerts instead of email or API calls.

### ****2. Advanced Machine Learning Models****

To improve anomaly detection and threat prediction, we can experiment with more advanced models:

#### **Implementation Steps**:

1. **Deep Learning Models**:
   * Replace the Isolation Forest and Random Forest models with deep learning models like LSTM (for sequential data) or Autoencoders (for anomaly detection).
   * Example:

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM, Dense

class LSTMModel:

def \_\_init\_\_(self, input\_shape):

self.model = Sequential([

LSTM(64, input\_shape=input\_shape),

Dense(1, activation='sigmoid')

])

self.model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

def train(self, X, y):

self.model.fit(X, y, epochs=10, batch\_size=32)

def predict(self, X):

return self.model.predict(X)

1. **Hyperparameter Tuning**:
   * Use tools like Optuna or GridSearchCV to optimize model hyperparameters.
2. **Ensemble Models**:
   * Combine multiple models (e.g., Isolation Forest + LSTM) for better performance.

### ****3. Web-Based Dashboard****

A dashboard will provide a user-friendly interface for visualizing threats and alerts.

#### **Implementation Steps**:

1. **Frontend Framework**:
   * Use a frontend framework like React or Vue.js for building the dashboard.
2. **Backend API**:
   * Create a REST API using Flask or FastAPI to serve threat data to the frontend.
   * Example:

from fastapi import FastAPI

from fastapi.middleware.cors import CORSMiddleware

app = FastAPI()

app.add\_middleware(

CORSMiddleware,

allow\_origins=["\*"],

allow\_methods=["\*"],

allow\_headers=["\*"],

)

@app.get("/threats")

def get\_threats():

system = ThreatDetectionSystem(config)

results = system.run\_detection\_cycle()

return results

1. **Visualization**:
   * Use libraries like Plotly or D3.js to create interactive charts and graphs.

### ****4. Integration with Security Tools****

To automate mitigation, we can integrate the system with firewalls, IDS/IPS, and other security tools.

#### **Implementation Steps**:

1. **Firewall Integration**:
   * Use APIs provided by firewalls (e.g., Palo Alto, Cisco) to block suspicious IPs or routes.
   * Example:

def block\_ip(ip\_address):

url = "https://firewall-api.example.com/block"

payload = {"ip": ip\_address}

response = requests.post(url, json=payload)

if response.status\_code == 200:

logger.info(f"Blocked IP: {ip\_address}")

else:

logger.error(f"Failed to block IP: {ip\_address}")

1. **IDS/IPS Integration**:
   * Send alerts to IDS/IPS systems for further investigation.
2. **Automated Mitigation**:
   * Implement automated actions based on threat levels (e.g., block routes, increase monitoring).

**Updated Code with Enhancements**

Here’s how the updated system might look with some of these enhancements:

# Real-Time Monitoring

from apscheduler.schedulers.background import BackgroundScheduler

# Advanced ML Models

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM, Dense

# Web Dashboard API

from fastapi import FastAPI

from fastapi.middleware.cors import CORSMiddleware

# Firewall Integration

import requests

# Real-Time Monitoring

def start\_real\_time\_monitoring():

scheduler = BackgroundScheduler()

scheduler.add\_job(run\_detection\_cycle, 'interval', minutes=5)

scheduler.start()

# Advanced ML Models

class LSTMModel:

def \_\_init\_\_(self, input\_shape):

self.model = Sequential([

LSTM(64, input\_shape=input\_shape),

Dense(1, activation='sigmoid')

])

self.model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

def train(self, X, y):

self.model.fit(X, y, epochs=10, batch\_size=32)

def predict(self, X):

return self.model.predict(X)

# Web Dashboard API

app = FastAPI()

app.add\_middleware(

CORSMiddleware,

allow\_origins=["\*"],

allow\_methods=["\*"],

allow\_headers=["\*"],

)

@app.get("/threats")

def get\_threats():

system = ThreatDetectionSystem(config)

results = system.run\_detection\_cycle()

return results

# Firewall Integration

def block\_ip(ip\_address):

url = "https://firewall-api.example.com/block"

payload = {"ip": ip\_address}

response = requests.post(url, json=payload)

if response.status\_code == 200:

logger.info(f"Blocked IP: {ip\_address}")

else:

logger.error(f"Failed to block IP: {ip\_address}")

# Main Execution

if \_\_name\_\_ == "\_\_main\_\_":

start\_real\_time\_monitoring()

**Deploy the System**:

* + Use Docker to containerize the application.
  + Deploy it on a cloud platform (e.g., AWS, GCP).