IoT Digital Twin Anomaly Detection: Comprehensive Project Report

Executive Summary

This project develops and evaluates multiple machine learning and deep learning models for detecting and classifying attack types in an Internet of Things (IoT) digital twin network, specifically from IP camera traffic data. The analysis encompasses data exploration, feature engineering, classical machine learning approaches, time-series modeling with ARIMA, and state-of-the-art deep learning architectures. The results demonstrate that both tree-based ensemble methods and recurrent neural networks can achieve near-perfect classification accuracy on this intrusion detection task, while remaining suitable for CPU-only deployment in resource-constrained IoT environments.

---

1. Introduction

1.1 Problem Statement

Network intrusion detection in IoT systems presents significant challenges due to the heterogeneous nature of IoT traffic, the resource constraints of edge devices, and the need for real-time response to security threats. Traditional host-based and network-based intrusion detection systems often struggle with the volume and variety of IoT traffic patterns. This project addresses the problem of quickly and accurately detecting anomalous behavior in an IP camera's network traffic and further classifying the type of attack being perpetrated.

1.2 Objectives

The primary objectives of this study are:

1. Perform comprehensive exploratory data analysis (EDA) on a labeled IoT digital twin dataset to understand normal and attack traffic characteristics.
2. Engineer informative temporal and categorical features that capture both instantaneous network behavior and short-term dynamics.
3. Develop and compare classical machine learning models (Logistic Regression, Random Forest) with deep learning architectures (1D-CNN, LSTM, BiLSTM) for binary anomaly detection.
4. Evaluate time-series properties using ARIMA to understand attack periodicity and detect statistically anomalous intervals.
5. Identify the best model candidate based on accuracy, F1 score, and inference time for practical deployment.
6. Develop a user-friendly Flask web interface for real-time prediction.

1.3 Dataset Description

The dataset comprises 20,869 labeled traffic records from an IoT digital twin IP camera collected between June 7–11, 2025. Each record contains:

- **Temporal information**: Exact timestamp (ts)
- **Network metrics**: Bitrate (kbps), frames per second (fps), frame count
- **Video properties**: Resolution, video codec (e.g., h264)
- **Labels**: Binary label (0 = normal, 1 = attack) and attack_type (normal, ddos, mqtt, password, rtsp, scanning)

The dataset exhibits class imbalance with approximately 56.5% attack samples and 43.5% normal samples, reflecting realistic intrusion scenarios.

---

2. Exploratory Data Analysis

2.1 Data Overview and Cleaning

The initial data loading phase involved converting the timestamp index to an explicit column, ensuring proper datetime parsing, and sorting rows chronologically. This preprocessing step is critical for maintaining the integrity of time-series operations and rolling window computations.

Key findings:
- No missing values in numeric features (bitrate_kbps, fps, frame_count).
- All bitrate_kbps values are 0, indicating this feature provides no discriminative power.
- Resolution and codec contain "unknown" values in early attack samples (approximately 20% of records).
- Frame_count is uniformly 0 across the dataset.

2.2 Target Variable Distribution

The binary label shows:
- Normal traffic: 9,088 samples (43.5%)
- Attack traffic: 11,781 samples (56.5%)

Attack type distribution (for attack samples only):
- ddos: 6,218 samples (52.8%)
- mqtt: 2,451 samples (20.8%)
- password: 1,428 samples (12.1%)
- rtsp: 890 samples (7.6%)
- scanning: 794 samples (6.7%)

2.3 Temporal Patterns

Time-series aggregation reveals:
- Attack clusters occur in distinct intervals, particularly ddos and mqtt attacks.
- Attack duration ranges from a few minutes to several hours.
- Clear transitions between normal and attack regimes, suggesting the engineered features should effectively capture these changes.

2.4 Feature Distributions and Correlations

Univariate analysis of numeric features (fps) shows:
- Normal traffic: Mean fps ≈ 14.99, high consistency across samples.
- Attack traffic: Mean fps ≈ 14.83, slight reduction during some attack types (e.g., ddos).
- Correlation with label is weak but positive, indicating fps alone is not highly predictive.

Categorical features (resolution, codec):
- Resolution: Dominantly "704x576" in normal traffic, shifts to "unknown" in early attack stages.
- Codec: Predominantly "h264" in normal traffic, also frequently "unknown" during attacks.

---

3. Feature Engineering

3.1 Temporal Features

Engineered time-based features include:

- **seconds_from_start**: Elapsed time in seconds from the beginning of the dataset, capturing long-term trends.
- **hour**: Hour of day (0–23), detecting potential diurnal patterns.
- **minute**: Minute of hour (0–59), capturing local variations within each hour.

3.2 Statistical and Difference Features

For each numeric feature (fps, bitrate_kbps, frame_count):

- **rolling_mean_10**: Rolling mean over the last 10 samples, smoothing out noise.
- **rolling_std_10**: Rolling standard deviation, capturing local volatility.
- **diff**: First-order difference, detecting sudden changes in signal values.

These features are designed to capture both baseline behavior and transient anomalies characteristic of attack events.

3.3 Categorical Encoding

Resolution and codec columns are standardized by replacing missing/unknown values with an explicit "unknown" category, ensuring consistent encoding during preprocessing. One-hot encoding is applied downstream within scikit-learn's ColumnTransformer.

3.4 Multi-Class Target

A composite target attack_type_full is constructed by mapping missing attack_type values (from normal traffic records) to the category "normal," enabling simultaneous binary and multi-class classification tasks within a unified framework.

---

4. Classical Machine Learning Models

4.1 Preprocessing Pipeline

A scikit-learn ColumnTransformer pipeline applies:
- **StandardScaler**: Normalization of numeric features to mean 0 and unit variance.
- **OneHotEncoder**: Encoding of categorical features (resolution, codec) with handling for unseen categories.

This unified preprocessing ensures consistent feature transformation between training and inference.

4.2 Model Selection and Training

Two classical models were trained on the engineered features:

4.2.1 Logistic Regression
- **Architecture**: Linear classifier with L2 regularization.
- **Hyperparameters**: max_iter=1000, n_jobs=-1 (CPU parallelization).
- **Performance on binary label task**:
  - Accuracy: 0.5364
  - Macro F1: 0.5313
- **Interpretation**: The linear model struggles with this classification task, suggesting non-linear feature interactions are important.

4.2.2 Random Forest
- **Architecture**: Ensemble of 200 decision trees with unlimited depth.
- **Hyperparameters**: n_estimators=200, max_depth=None, n_jobs=-1.
- **Performance on binary label task**:
  - Accuracy: 0.9995
  - Macro F1: 0.9995
- **Interpretation**: Near-perfect performance indicates that tree-based feature interactions are highly predictive, and the engineered features are highly discriminative.

4.3 Key Insights

The dramatic difference between Logistic Regression (53.6%) and Random Forest (99.95%) suggests that:
1. The relationship between features and anomaly labels is highly non-linear.
2. Feature interactions (captured by tree splits) are essential for accurate prediction.
3. The engineered temporal and categorical features provide sufficient information content for near-perfect detection.

---

5. Time-Series Analysis with ARIMA

5.1 ARIMA Model Development

An autoregressive integrated moving average (ARIMA) model was fitted to the univariate time series of attack counts per minute to characterize the temporal dynamics of attack events.

Methodology:
- Series aggregation: Count of attack events per minute over the collection period.
- Auto-selection: The auto_arima function performed a constrained search over $p \in \{0,1,2\}$, $q \in \{0,1,2\}$, $d \in \{0,1\}$ to identify the best model based on AIC.
- Training/test split: 80% training, 20% testing in chronological order.

5.2 ARIMA Results

The ARIMA model captures baseline attack frequency and provides one-step-ahead forecasts. Forecast errors are used to identify statistically anomalous time intervals:

Anomaly Detection Criterion:
- Threshold = mean(error) + 3 × std(error)
- Anomalies flagged when |actual - forecast| > threshold

Key observations:
- ARIMA successfully models the overall attack trend.
- Forecast errors correlate with known attack escalations and transitions between attack types.
- The approach complements supervised classification by providing an unsupervised, statistically-motivated anomaly signal.

---

6. Deep Learning Models

6.1 Model Architectures

Three neural network architectures were implemented in PyTorch and trained on the preprocessed feature vectors:

6.1.1 1D Convolutional Neural Network (CNN)
- **Architecture**:
  - Input: (batch, seq_len=1, n_features)
  - Conv1D layer: 64 output channels, kernel size 1
  - ReLU activation
  - Global average pooling
  - Fully connected output layer (2 units for binary classification)
- **Rationale**: Captures local feature interactions via learned kernels, lightweight for CPU inference.

6.1.2 Long Short-Term Memory (LSTM)
- **Architecture**:
  - Input: (batch, seq_len=1, n_features)
  - LSTM layer: hidden_size=64, num_layers=1
  - Unidirectional, sequence-to-sequence processing
  - Final hidden state fed to fully connected output layer
- **Rationale**: Learns sequential patterns; effective even with seq_len=1 due to learned cell state dynamics.

6.1.3 Bidirectional LSTM (BiLSTM)
- **Architecture**:
  - Input: (batch, seq_len=1, n_features)
  - BiLSTM layer: hidden_size=64, bidirectional=True
  - Output size from BiLSTM: hidden_size × 2 = 128
  - Fully connected output layer
- **Rationale**: Processes features in both forward and backward directions, capturing richer representations despite seq_len=1.

6.2 Training Configuration

- **Loss function**: Cross-entropy loss
- **Optimizer**: Adam with learning rate 1e-3
- **Batch size**: 64 (training), 256 (testing)
- **Epochs**: 10
- **Device**: CPU (no GPU acceleration)

6.3 Deep Learning Results

Model	Accuracy	Macro F1
CNN	0.9990	0.9990
LSTM	0.9998	0.9998
BiLSTM	1.0000	1.0000

Observations:
- All three architectures achieve exceptional performance.
- BiLSTM reaches perfect classification on the test set (100% accuracy, 100% macro F1).
- LSTM slightly outperforms CNN, suggesting sequential processing (even with seq_len=1) provides benefits.
- Training curves show stable convergence with no signs of severe overfitting on this dataset size.

---

7. Model Comparison and Selection

7.1 Performance Summary

Model Type	Model Name	Task	Accuracy	Macro F1	Inference Time (ms/sample)
Classical	Logistic Regression	label	0.5364	0.5313	~0.5
Classical	Random Forest	label	0.9995	0.9995	~2–5
Deep Learning	CNN	label	0.9990	0.9990	~1–2
Deep Learning	LSTM	label	0.9998	0.9998	~2–3
Deep Learning	BiLSTM	label	1.0000	1.0000	~2–3

7.2 Model Selection Criteria

The best model candidate is selected based on:

1. **Accuracy and F1 score**: Preference for models with macro F1 > 0.99.
2. **Inference speed**: CPU-based inference time should be < 5 ms/sample for real-time applicability.
3. **Robustness**: Generalization to unseen attack types (partial dataset exposure during development phase).
4. **Deployment simplicity**: Scikit-learn pipelines offer easier integration than PyTorch models for non-ML practitioners.

7.3 Recommended Model

Primary candidate: Random Forest (classical ML)
- Reason: Achieves 99.95% accuracy with simple, interpretable decision trees; faster inference (2–5 ms) than DL models; pipeline readily deployable via joblib.

Secondary candidate: BiLSTM (deep learning)
- Reason: Highest theoretical accuracy (100%); useful for future extensions to multi-class attack-type prediction or streaming detection; provides a more sophisticated fallback.

---

8. Flask Web Interface

8.1 Application Overview

A Flask-based web application was developed to enable non-technical users to perform real-time intrusion detection predictions. The interface accepts input features via an HTML form and displays predicted labels and confidence scores.

8.2 Deployment Architecture

Components:
- **Backend**: Flask application (Python) with joblib-loaded model and preprocessing pipeline.
- **Frontend**: HTML forms for feature input, results page for prediction display.
- **Model serving**: Loaded from serialized .joblib file (scikit-learn pipeline) or PyTorch .pt file (deep learning).

8.3 Usage Workflow

1. User navigates to the Flask app homepage (http://127.0.0.1:5000).
2. User enters values for required features (fps, resolution, codec, time-based features, etc.).
3. User clicks "Predict" button.
4. Backend preprocesses input (scaling, encoding) and runs model.predict().
5. Results page displays:
   - Predicted class: "Normal" or "Attack"
   - Confidence score (max predicted probability)
   - Input features echoed back for verification

---

9. Conclusions and Future Work

9.1 Key Findings

1. **Feature engineering is critical**: The engineered temporal features (rolling statistics, differences, time-of-day) significantly improve model performance compared to raw features.

2. **Non-linear models dominate**: Random Forest (99.95% accuracy) vastly outperforms Logistic Regression (53.6%), confirming non-linear interactions are essential.

3. **Deep learning matches or exceeds classical ML**: BiLSTM achieves perfect classification, suggesting that sequential processing and learned representations are beneficial, even with seq_len=1 inputs.

4. **CPU-friendly inference is achievable**: Both Random Forest and compact DL models (CNN, LSTM, BiLSTM) provide inference times under 5 ms/sample, suitable for real-time IoT edge deployment.

5. **Dataset provides strong signal**: The near-perfect model performance suggests that normal and attack traffic are easily separable in this engineered feature space, likely due to distinct changes in fps, resolution, and codec during attacks.