Table of Contents

Network Intrusion Detection System (NIDS)

Technical Documentation

Complete Project Walkthrough: From Data to Deployment

Project Highlights

Metric	Value
Dataset Size	2,520,751 network packets
Model Accuracy	97.47%
F1-Score	98.21%
Attack Types Detected	7 categories
Features Analyzed	52 network flow features
Technologies	Python, Scikit-learn, Streamlit

Table of Contents

---

1. Executive Summary

This document provides comprehensive technical documentation for the Network Intrusion Detection System (NIDS) project. The system leverages machine learning to analyze network traffic in real-time and identify malicious activities with high accuracy.

Project Objectives

- Develop a machine learning-based intrusion detection system
- Analyze 2.5+ million network packets from the CICIDS2017 dataset
- Achieve 95%+ accuracy in detecting network attacks
- Detect multiple attack categories including DoS, DDoS, Port Scanning, Brute Force, Web Attacks, and Bots
- Build an interactive dashboard for real-time monitoring and visualization

Key Achievements

Objective	Target	Achieved	Status
Model Accuracy	95%	97.47%	✅ Exceeded
F1-Score	95%	98.21%	✅ Exceeded
Precision	95%	99.12%	✅ Exceeded
Recall	95%	97.47%	✅ Exceeded
Attack Types	5+	7	✅ Exceeded
Real-time Dashboard	Yes	Yes	✅ Complete

2. Dataset Overview

2.1 Original CICIDS2017 Dataset

The CICIDS2017 (Canadian Institute for Cybersecurity Intrusion Detection System 2017) dataset is a comprehensive benchmark dataset for network intrusion detection research. It was created by the Canadian Institute for Cybersecurity at the University of New Brunswick.

Dataset Characteristics:

- Created over 5 days (Monday to Friday, July 3-7, 2017)
- Contains both benign traffic and modern attack scenarios
- Generated using CICFlowMeter tool for feature extraction
- Originally split across multiple CSV files by day
- Contains 80+ raw features extracted from network flows

Original Attack Types in CICIDS2017:

Day	Attack Types	Description
Monday	Benign only	Normal background traffic
Tuesday	FTP-Patator, SSH-Patator	Brute force attacks
Wednesday	DoS Hulk, DoS GoldenEye, DoS Slowloris, DoS Slowhttptest, Heartbleed	Denial of Service attacks
Thursday	Web Attack (Brute Force, XSS, SQL Injection), Infiltration	Web-based attacks
Friday	Botnet ARES, PortScan, DDoS LOIT	Bot, scanning, and DDoS attacks

2.2 Data Preprocessing Steps (Pre-Project)

Before this project began, the raw CICIDS2017 dataset underwent extensive preprocessing to ensure data quality and reduce noise. The following steps were applied:

Step 1: File Merging

The original CICIDS2017 dataset is distributed across multiple CSV files, one for each day of data collection. These files were merged into a single, unified dataset to facilitate easier analysis and model training.

Rationale: A unified dataset simplifies the data pipeline and ensures consistent preprocessing across all samples.

Step 2: Duplicate Removal

Duplicate rows were identified and removed to improve data integrity and prevent the model from learning redundant patterns. The same process was applied to identify and remove duplicate columns.

Rationale: Duplicates can artificially inflate model performance and lead to overfitting. Removing them ensures the model learns from unique network flow patterns.

Step 3: Infinite Value Handling

Network flow calculations sometimes produce infinite values (e.g., division by zero in rate calculations). These infinite values were replaced with NaN and then handled appropriately.

Rationale: Infinite values can cause numerical instability in machine learning algorithms and must be addressed before training.

Step 4: Missing Value Handling

Rows with missing values were removed from the dataset. This decision was made because missing values represented less than 1% of the total dataset, making removal a safe choice that avoids the complexity and potential bias of imputation methods.

Rationale: With <1% missing data, removal is preferred over imputation as it maintains data authenticity without significantly reducing dataset size.

Decision Point: Imputation methods (mean, median, KNN) were considered but rejected because: - The missing data percentage was negligible (<1%) - Imputation could introduce artificial patterns in network traffic data - Removal maintains the authenticity of real network flows

Step 5: Inconsistent Whitespace

Leading and trailing whitespace in column names were removed for consistency. This prevents issues with column access and ensures reliable feature selection.

Rationale: Whitespace inconsistencies can cause silent failures when accessing columns by name.

Step 6: Data-Driven Feature Selection

Multiple feature selection techniques were applied to reduce dimensionality while preserving predictive power:

6.1 Single-Value Column Removal

Columns with only one unique value were removed as they provide no discriminative power for classification.

Rationale: A feature with no variance cannot help distinguish between classes.

6.2 Correlation Analysis

For pairs of features with near-perfect correlation (≥0.99), one feature was removed to reduce multicollinearity.

Rationale: - Highly correlated features provide redundant information - Multicollinearity can cause numerical instability - Reducing features improves training speed and model interpretability

6.3 H-Statistics and Tree Feature Selection

The Kruskal-Wallis test was combined with Random Forest feature importance to eliminate statistically irrelevant columns.

Rationale: - Kruskal-Wallis test identifies features with no statistical relationship to the target - Random Forest importance confirms predictive value - Combined approach ensures robust feature selection

Decision Point: These techniques reduced the feature space from 80+ columns to 52, significantly reducing computational requirements while maintaining (and often improving) model accuracy.

Step 7: Target Feature Handling

The original 'Label' column contained granular attack labels. These were consolidated into broader categories for improved model generalization:

Original Labels	Grouped As	Rationale
DoS Hulk, DoS GoldenEye, DoS Slowloris, DoS Slowhttptest	DoS	Similar attack mechanism (resource exhaustion)
FTP-Patator, SSH-Patator	Brute Force	Same attack category (credential guessing)
Web Attack - Brute Force, XSS, SQL Injection	Web Attacks	Web-based attack vector
DDoS LOIT	DDoS	Distributed denial of service
Botnet ARES	Bots	Botnet traffic patterns
PortScan	Port Scanning	Reconnaissance activity
BENIGN	Normal Traffic	Legitimate traffic

Removed Attack Types:

- **Infiltration**: Removed due to extremely low sample count (<40 samples)
- **Heartbleed**: Removed due to extremely low sample count (<15 samples)

Rationale: With so few samples, these classes would cause overfitting and poor generalization. The model would memorize these specific samples rather than learning generalizable patterns.

2.3 Final Dataset Characteristics

Property	Value
Total Samples	2,520,751
Total Features	52 (numeric)
Label Column	Attack Type
Missing Values	0
Duplicate Rows	161 (removed during project)
File Size	~685 MB
Memory Usage	~1.12 GB when loaded

Attack Distribution in Final Dataset:

Attack Type	Count	Percentage	Risk Level
Normal Traffic	2,094,896	83.11%	Low (Benign)
DoS	193,745	7.69%	Critical
DDoS	128,014	5.08%	Critical
Port Scanning	90,694	3.60%	Medium
Brute Force	9,150	0.36%	Medium
Web Attacks	2,143	0.09%	Medium
Bots	1,948	0.08%	Medium

Key Observation - Class Imbalance:

The dataset exhibits significant class imbalance, with Normal Traffic comprising 83.11% of all samples. This imbalance is addressed during the feature engineering phase using undersampling techniques to prevent the model from being biased toward predicting the majority class.

---

3. Data Loading & Exploration

3.1 Loading Strategy

The data loading module (`data_loader.py`) was designed with the following considerations:

Memory Efficiency:

- The full dataset (2.5M rows) requires ~1.12 GB of RAM
- The `low_memory=False` parameter ensures consistent data type inference
- Optional sampling functionality allows testing with smaller subsets

Implementation:

```
df = pd.read_csv(file_path, low_memory=False)
```

Decision Point - Why Pandas?

We chose pandas over alternatives like Dask or Vaex because: - The dataset fits comfortably in memory on modern systems (8GB+ RAM) - Pandas provides simpler API and better integration with scikit-learn - No distributed computing infrastructure was required - Faster iteration during development

3.2 Exploratory Data Analysis

The exploration phase revealed several important characteristics:

Data Types:

- 52 numeric features (int64 and float64)
- 1 categorical feature (Attack Type - the target variable)

Feature Categories:

Category	Example Features	Count
Flow Statistics	Flow Duration, Flow Bytes/s, Flow Packets/s	8
Packet Length	Fwd Packet Length Max/Min/Mean/Std	12
Inter-Arrival Time	Flow IAT Mean/Std/Max/Min	8
Flag Counts	FIN Flag Count, PSH Flag Count, ACK Flag Count	3
Header Information	Fwd Header Length, Bwd Header Length	4
Subflow Statistics	Subflow Fwd Bytes, Init_Win_bytes_forward	6
Active/Idle	Active Mean/Max/Min, Idle Mean/Max/Min	6
Port Information	Destination Port	1
Other	Various derived metrics	4

---

4. Feature Engineering

Feature engineering is the most critical phase in building an effective machine learning model. This section details each preprocessing step and the rationale behind it.

4.1 Data Cleaning

Although the dataset was pre-cleaned, additional validation was performed:

Duplicate Removal:

```
df = df.drop_duplicates()  # Removed 161 remaining duplicates
```

Rationale: Even a small number of duplicates can affect model evaluation if they appear in both training and test sets.

Infinite Value Handling:

```
# Replace inf with column max, -inf with column min
df.loc[df[col] == np.inf, col] = col_max
df.loc[df[col] == -np.inf, col] = col_min
```

Rationale: Using column max/min preserves the relative scale of the data while removing problematic values.

Missing Value Handling:

```
df[col] = df[col].fillna(df[col].median())
```

Rationale: Median imputation is robust to outliers, which are common in network traffic data.

4.2 Label Encoding

Machine learning algorithms require numeric inputs. The categorical target variable was encoded using scikit-learn's LabelEncoder:

Original Label	Encoded Value
Bots	0
Brute Force	1
DDoS	2
DoS	3
Normal Traffic	4
Port Scanning	5
Web Attacks	6

Note: The encoding order is alphabetical by default. The `label_mapping` dictionary is preserved for converting predictions back to human-readable labels.

4.3 Train/Test Split

The dataset was split into training (80%) and testing (20%) sets using stratified sampling:

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)
```

Why Stratified Sampling?

- Preserves the class distribution in both sets

- Ensures rare attack types (Bots, Web Attacks) appear in both training and testing
- Provides more reliable evaluation metrics

Set	Samples	Purpose
Training	2,016,472 (80%)	Model learning
Testing	504,118 (20%)	Unbiased evaluation

4.4 Feature Scaling

StandardScaler was applied to normalize all features to zero mean and unit variance:

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)  # Fit on training only!
X_test_scaled = scaler.transform(X_test)  # Transform test data
```

Why StandardScaler?

While Random Forest is not sensitive to feature scaling, scaling helps with: - Numerical stability in calculations - Consistent feature importance interpretation - Compatibility if switching to distance-based algorithms

Critical Decision: Fit on Training Data Only

The scaler is fitted ONLY on training data to prevent data leakage. Fitting on the full dataset would allow information from the test set to influence the transformation, leading to overly optimistic performance estimates.

4.5 Handling Class Imbalance

The dataset is highly imbalanced (83% Normal Traffic). Without addressing this, the model would achieve high accuracy by simply predicting “Normal Traffic” for everything.

Before Resampling:

Class	Training Samples	Percentage
Normal Traffic	1,675,917	83.11%
DoS	154,996	7.69%
DDoS	102,411	5.08%
Port Scanning	72,555	3.60%
Brute Force	7,320	0.36%
Web Attacks	1,714	0.09%
Bots	1,559	0.08%

Resampling Strategy: Random Undersampling

```
undersampler = RandomUnderSampler(random_state=42)
X_train_balanced, y_train_balanced =
undersampler.fit_resample(X_train_scaled, y_train)
```

After Resampling:

All classes reduced to 1,559 samples each (matching the minority class), resulting in **10,913 total training samples**.

Why Undersampling Instead of Oversampling (SMOTE)?

Consideration	Undersampling	SMOTE
Training set size	Smaller (faster training)	Larger
Data authenticity	Real samples only	Synthetic samples
Memory usage	Lower	Higher
Risk of overfitting	Lower	Higher (synthetic patterns)

Decision: With 1,559 samples per class, the model has sufficient data to learn patterns. Undersampling maintains data authenticity and significantly reduces training time.

Important: The test set remains unbalanced to provide realistic evaluation metrics that reflect production performance.

---

5. Model Training

5.1 Algorithm Selection: Random Forest

Random Forest was selected as the primary classification algorithm after considering several alternatives:

Algorithm	Pros	Cons	Decision
Random Forest	Fast, interpretable, handles high dimensions, robust	Memory usage for large forests	✅ Selected
XGBoost	Often higher accuracy, handles imbalance well	More hyperparameters to tune	Alternative
Neural Network	Can capture complex patterns	Requires more data, less interpretable	Not selected
Logistic Regression	Simple, fast, interpretable	Assumes linear relationships	Baseline only
SVM	Effective in high dimensions	Slow on large datasets	Not suitable

Why Random Forest is Ideal for NIDS:

- Handles tabular data with 52 features effectively
- Captures non-linear relationships between network features
- Provides feature importance rankings for security insights
- Resistant to overfitting through ensemble averaging
- Fast training and prediction times

- No assumption about data distribution

5.2 Hyperparameter Configuration

The following hyperparameters were selected for the production model:

Parameter	Value	Rationale
n_estimators	200	More trees = better accuracy, diminishing returns after 200
max_depth	30	Prevents overfitting while allowing complex patterns
min_samples_split	10	Requires at least 10 samples to split a node
min_samples_leaf	5	Each leaf must have at least 5 samples
class_weight	balanced	Additional handling for any residual imbalance
n_jobs	-1	Use all CPU cores for parallel training
random_state	42	Reproducibility

5.3 Training Process

```
model = RandomForestClassifier(
    n_estimators=200,
    max_depth=30,
    min_samples_split=10,
    min_samples_leaf=5,
    class_weight='balanced',
    n_jobs=-1,
    random_state=42
)
model.fit(X_train_balanced, y_train_balanced)
```

Training Statistics:

Metric	Value
Training Samples	10,913 (balanced)
Training Time	~1.1 seconds
Training Accuracy	99.58%
Model File Size	4.6 MB

---

6. Model Evaluation

Model evaluation was performed on the held-out test set (504,118 samples) that was NOT used during training and maintains the original class distribution.

6.1 Overall Performance Metrics

Metric	Score	Description
Accuracy	97.47%	Percentage of correct predictions

Metric	Score	Description
Precision (Weighted)	99.12%	Of predicted attacks, how many were real?
Recall (Weighted)	97.47%	Of real attacks, how many did we catch?
F1-Score (Weighted)	98.21%	Harmonic mean of precision and recall

Prediction Speed:

504,118 samples predicted in 2.74 seconds (~184,000 predictions/second)

6.2 Per-Class Analysis

Attack Type	Precision	Recall	F1-Score	Support
Bots	6%	99%	11%	389
Brute Force	71%	100%	83%	1,830
DDoS	100%	100%	100%	25,603
DoS	95%	99%	97%	38,749
Normal Traffic	100%	97%	98%	418,979
Port Scanning	96%	100%	98%	18,139
Web Attacks	16%	100%	27%	429

Analysis of Results:

Excellent Performance (F1 > 95%): - **DDoS**: Perfect detection (100% precision and recall) - **Normal Traffic**: 98% F1-score with 100% precision - **Port Scanning**: 98% F1-score, nearly perfect detection - **DoS**: 97% F1-score, very reliable detection

Good Performance (F1 70-95%): - **Brute Force**: 83% F1-score, 71% precision with perfect recall

Challenging Classes (F1 < 50%): - **Bots**: 11% F1-score - very few samples (389 in test set) - **Web Attacks**: 27% F1-score - very few samples (429 in test set)

Why Low Precision for Rare Classes?

The low precision for Bots and Web Attacks is due to their extremely small sample sizes. The model achieves 99-100% recall (catching all actual attacks) but generates false positives from similar-looking normal traffic.

Security Perspective: This is actually acceptable behavior for security systems where **missing an attack is worse than a false alarm**. High recall ensures threats are detected, and false positives can be investigated.

6.3 Feature Importance Analysis

Random Forest provides feature importance scores indicating which network characteristics are most predictive of attacks:

Rank	Feature	Importance	Interpretation
1	Destination Port	11.07%	Target port is the strongest indicator of attack type
2	Init_Win_bytes_backward	6.81%	TCP window size reveals connection behavior
3	Packet Length Mean	4.37%	Average packet size differs by attack type
4	Total Length of Fwd Packets	4.34%	Forward data volume indicates attack patterns
5	Subflow Fwd Bytes	4.07%	Subflow statistics capture attack signatures
6	Average Packet Size	4.06%	Overall packet size distribution
7	Fwd Packet Length Max	4.02%	Maximum forward packet reveals anomalies
8	Bwd Header Length	3.72%	Backward header size for response analysis
9	Bwd Packet Length Mean	3.58%	Response packet characteristics
10	Fwd Packet Length Mean	3.04%	Average forward packet size

Security Insights from Feature Importance:

- **Destination Port (11.07%)**: Different attacks target specific ports (e.g., SSH on 22, HTTP on 80, MySQL on 3306)
- **Packet Length Features**: Attack traffic often has distinctive packet size patterns (e.g., DoS attacks may use maximum size packets)
- **TCP Window Size**: Abnormal window sizes can indicate malicious behavior or protocol manipulation
- **Header Lengths**: Protocol anomalies manifest in header statistics

---

7. Interactive Dashboard

A Streamlit-based interactive dashboard was developed to visualize the NIDS capabilities and provide a user-friendly interface for monitoring network traffic.

Dashboard Features

1. Security Overview Dashboard

- Real-time metrics: Total packets, normal traffic, attacks detected, attack rate
- Attack distribution pie chart showing proportion of each attack type
- Bar chart comparing attack counts
- Summary table with risk level indicators

2. Live Network Traffic Monitor

- Simulated real-time packet analysis
- Color-coded alerts: Green for normal, Red for attacks
- Threat level indicator (Low/Medium/High)
- Auto-refresh capability for continuous monitoring

3. Model Performance Page

- Model information (algorithm, parameters)
- Performance metrics display (accuracy, precision, recall, F1)
- Per-class performance visualization
- Grouped bar chart comparing metrics across attack types

4. Feature Importance Analysis

- Horizontal bar chart of top 15 features
- Feature descriptions explaining each metric
- Interactive visualization with hover details

5. Test Prediction

- Select any sample from the dataset
- View input features and predicted class
- Confidence score and probability distribution
- Actual vs predicted comparison

Running the Dashboard

```
streamlit run dashboard/app.py
```

The dashboard will be available at http://localhost:8501

---

8. Conclusions & Future Work

8.1 Project Summary

This project successfully developed a Network Intrusion Detection System that exceeds all target metrics:

Objective	Target	Result
Accuracy	95%	97.47% ✅
F1-Score	95%	98.21% ✅
Attack Detection	5+ types	7 types ✅
Real-time Dashboard	Functional	Complete ✅
Dataset Scale	100K+	2.5M+ ✅

8.2 Key Learnings

1. **Class imbalance handling is critical** for security applications
2. **Feature engineering** (pre-project) significantly impacts model quality
3. **Random Forest** provides excellent balance of accuracy and interpretability
4. **Undersampling** is effective when minority classes have sufficient samples
5. **Feature importance** provides actionable security insights

8.3 Limitations

- Low precision on rare attack types (Bots, Web Attacks) due to small sample sizes
- Model trained on 2017 data may not detect newer attack patterns
- Simulated dashboard does not connect to real network traffic
- Binary (attack/normal) detection may be more practical than multi-class

8.4 Future Improvements

- ☐ Implement SMOTE or other oversampling for rare classes
- ☐ Try ensemble methods combining multiple algorithms
- ☐ Add deep learning models (LSTM) for sequence-based detection
- ☐ Integrate with real network monitoring tools (Wireshark, Snort)
- ☐ Deploy as API service for production use
- ☐ Add automated alerting and notification system
- ☐ Implement model retraining pipeline for new data

8.5 Files and Components

Component	File	Purpose
Data Loader	src/data_loader.py	Load and explore dataset
Feature Engineering	src/feature_engineering.py	Preprocess data for ML
Model Training	src/model_training.py	Train and evaluate models
Full Training	train_full_model.py	Train on complete dataset
Dashboard	dashboard/app.py	Interactive visualization
Notebook	notebooks/NIDS_Complete_Walkthrough.ipynb	Step-by-step documentation
Trained Model	models/random_forest_model.joblib	Serialized model
Scaler	models/scaler.joblib	Feature scaler
Label Mapping	models/label_mapping.joblib	Class name mapping