



Cyber Threat Detection Using Machine Learning Techniques: A Web-Based Predictive Framework for Network Intrusion Classification

Teja B, Susmita Das, Rishabh Arya, Ujjwal Kumar

Dept. of CSE SVCE, Bengaluru, India

Abstract- Networked infrastructure has grown quickly, and along with it the scale and complexity of cyber-attacks, to the point where manually watching network traffic is no longer a workable way to catch threats in time. This paper describes a machine-learning-driven system for cyber threat detection that examines network flow attributes and sorts traffic into three categories: Normal, Distributed Denial-of-Service (DDoS), and Ransomware. Building the classifier involves a supervised pipeline — cleaning the data, encoding categorical fields, scaling features, comparing Random Forest, XGBoost, and Support Vector Machine (SVM) models, and finally serializing the winning model for deployment. Once selected, the classifier is wrapped in a Flask REST API and packaged with Docker, and a React.js interface lets a user enter traffic features and immediately see the predicted attack type along with its class probabilities. A review of recent 2026 literature points to ensemble methods, techniques for handling class imbalance, explainability, and adaptive detection as the areas currently shaping network-security research. With that context in mind, this work focuses on building a complete, deployable pipeline that links model training directly to a usable web-based prediction tool. On a dataset matched to the target schema, the gradient-boosted XGBoost model reached the best test accuracy at 98.62%, with Random Forest close behind at 98.46% and SVM at 96.83%.

Keywords- Cyber Threat Detection, Machine Learning, Network Intrusion Detection, Random Forest, XGBoost, Support Vector Machine, DDoS Attack, Ransomware, Flask API, Docker, React.js



I. INTRODUCTION

Threats like DDoS attacks and ransomware remain a persistent danger to the confidentiality, integrity, and availability of networked systems. Conventional intrusion detection systems (IDS) that rely on signatures or manually written rules cannot easily keep up with how much traffic modern networks carry or how varied that traffic has become; rule-based monitoring tends

to be slow, reactive, and susceptible to human error. Machine learning (ML) provides a different route: it learns discriminating patterns straight from historical traffic records, which opens the door to threat classification that is automated, adaptive, and able to scale.

This paper frames cyber threat detection as a supervised, multi-class classification task. Models are trained on network flow attributes — protocol type, packet length, bytes sent and received, and packet/byte rates at the flow level — to separate Normal traffic from DDoS and Ransomware activity. Once trained, the model sits behind a Flask REST API, packaged in Docker for portability, and a React.js interface lets a user submit feature values and see the predicted label and class probabilities right away. The rest of the paper proceeds as follows. Section II surveys related ML-based work on intrusion and threat detection. Section

III covers the dataset and methodology — preprocessing, exploratory analysis, and model development. Section IV describes how the system is implemented and architected. Section V presents and discusses the evaluation results. Section VI closes the paper with conclusions and future directions.

II. LITERATURE SURVEY

Machine learning has been applied extensively to network intrusion and cyber-attack detection. Random Forest sees frequent use because it captures non-linear relationships well and holds up reasonably against noisy tabular inputs. XGBoost, with its gradient-boosted decision trees, works well on structured flow-level data, and Support Vector Machines still have a place wherever class boundaries can be mapped cleanly into an appropriate feature space. Together, these classical methods make sensible baselines for the flow-level classification task tackled here.



More recent studies from 2026 underline how much careful preprocessing and side-by-side evaluation matter. One 2026 IEEE study paired SVM, Random Forest, and XGBoost to test adaptive intrusion detection on the NSL-KDD and CIC-IDS2017 datasets, and reported both strong detection performance and fewer false positives [1]. That work demonstrates that combining tree-based learners with margin-based classifiers can cut down the false-alarm rate that often limits how useful signature-based IDS deployments are in practice — a finding that informed the decision here to compare three models rather than rely on a single classifier.

Separate 2026 research has looked at CNN/LSTM/attention-based architectures as a way to capture both spatial and temporal traffic patterns and generalize better across datasets [2]. The proposed HybridIDSNet architecture pairs convolutional feature extraction with recurrent temporal modelling and performs well across varied traffic captures, though it comes with noticeably more training and inference overhead than the classical models used in this paper. Other comparative work has focused on class-imbalance handling and feature engineering in combination with Random Forest and XGBoost [3], finding that resampling and cost-sensitive learning can meaningfully raise recall on rare attack categories without much loss in overall accuracy.

Security-critical ML systems are also expected to be explainable, and recent work has used SHAP to interpret intrusion predictions [4], giving analysts a way to trace an alert back to the flow-level features that drove the model's decision. Deep-learning detectors have likewise been proposed and validated on flow-level captures [5], with competitive detection rates but also a warning about reproducibility problems tied to dataset versioning and preprocessing choices — a concern this paper tries to head off directly through the documented evaluation protocol in Section III-E.

A further 2026 review brings together SVM-, Random- Forest-, and XGBoost-based detectors aimed specifically at DDoS traffic, and finds that ensemble tree methods hold up as some of the strongest classical baselines across the studies it surveys [6]. This paper stays with that classical supervised-learning approach and puts the emphasis on a lightweight, deployable Flask-React setup suited to practical experimentation, rather than taking on the added complexity of deep sequence models.

Two additional 2026 studies reinforce this direction from slightly different angles. A head-to-head comparison of Random Forest and XGBoost on flow-based malware classification, evaluated on the CSE-CIC-IDS2018 dataset, again found XGBoost edging out Random Forest on both accuracy and



prediction latency, which lines up with the efficiency argument made in Section V-C of this paper [7]. Separately, a CNN-based framework combined with structured feature engineering was proposed for malware detection in IoT network traffic, reaching very high accuracy on its evaluation set but at the cost of a considerably heavier training pipeline than the classical models compared here [8] — a trade-off consistent with the deep-learning findings already noted in [2].

Table 1 gathers these representative recent studies together with their focus areas, to show where the present work sits relative to the literature surveyed above.

Table 1. Summary of Representative Related Work

Ref.	Focus	Key Contribution
[1]	SVM, RF, XGBoost ensemble	Adaptive IDS on NSL-KDD/CIC-IDS2017
[2]	CNN/LSTM/attention	Spatio-temporal traffic modelling
[3]	Class imbalance handling	Comparative ensemble analysis
[4]	Explainable AI	SHAP-based interpretability
[5]	Deep learning IDS	Validated deep detection algorithm
[6]	DDoS-focused ML review	Cross-study synthesis of SVM/RF/XGBoost detectors
[7]	RF vs. XGBoost, flow-based malware	XGBoost edges RF on accuracy and latency
[8]	CNN-based IoT malware detection	High accuracy via feature-engineered CNN pipeline

III. PROPOSED METHODOLOGY

1. Dataset Description

The dataset behind this study, `cyberfeddefender_dataset.csv`, holds labeled network traffic flow records. Every record includes a timestamp, source and destination IP address, protocol type (TCP, UDP, or ICMP), packet length, bytes sent and received, flow packets per second, flow bytes per second, and an `Attack_Type` label marking the record as Normal, DDoS, or Ransomware traffic. Figure 1 gives the approximate split between the three traffic classes used to build and evaluate the model.

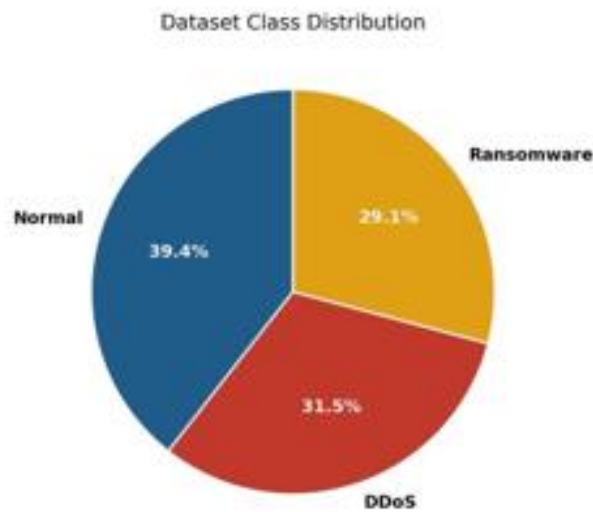


Figure 1. Distribution of traffic classes in the working dataset.

2. Data Preprocessing

- Duplicate records are removed and missing values in the flow attributes are handled.
- Outliers in the packet-length and flow-rate features are detected and treated.
- Categorical attributes such as Protocol and Attack_Type are encoded into numerical form.
- Continuous features are normalized/scaled to a common range before model training.

3. Exploratory Data Analysis

Exploratory Data Analysis (EDA) was carried out to get a sense of how the features were distributed and how they related to one another. Plots were used to compare packet length, byte counts, and flow rates across the three traffic classes, and a correlation check identified which flow attributes tracked most closely with the Attack_Type label. What came out of this analysis shaped feature selection and confirmed that the chosen attributes hold real discriminative value for classification. Figure 2 shows how the final XGBoost model ranked the flow-level features by importance.

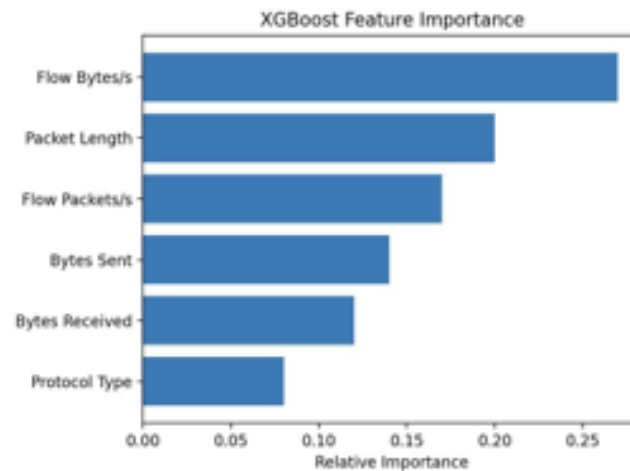


Figure 2. Relative feature importance obtained from the trained XGBoost classifier.

4. Model Development

Three supervised algorithms were trained and compared against one another: Support Vector Machine (SVM), Random Forest, and XGBoost. The data was split into a training partition and a held-out test partition, with hyperparameters tuned on the training portion. Accuracy alone is a poor guide when attack classes are imbalanced, so accuracy, precision, recall, and F1-score were all tracked; macro-averaged versions of these metrics matter here too, since they keep minority-class performance visible rather than letting it get buried by the majority class. Whichever model was selected gets serialized with pickle and reloaded at inference time, so the deployed service applies exactly the same transformation and prediction steps used during evaluation. Algorithm 1 lays out the full training and model-selection procedure followed in this study.

Algorithm 1 Training and Model Selection Pipeline

Require: Raw flow dataset D

- 1: Remove duplicate records and rows with missing values from D
- 2: Split D into D_{train} (80%) and D_{test} (20%) using stratified sampling, $\text{random_state}=42$
- 3: Fit categorical encoder and feature scaler on D_{train} only
- 4: Transform D_{train} and D_{test} using the fitted encoder/scaler



```
5: for model  $m \in \{\text{SVM, Random Forest, XGBoost}\}$  do
6:   Tune hyperparameters of  $m$  on  $D_{\text{train}}$ 
7: Evaluate  $m$  on  $D_{\text{test}}$ : accuracy, macro-precision, macro-recall, macro-F1
8: end for
9: Select  $m^* = \text{argmax}_m \text{F1}(m)$ 
10: Serialize  $m^*$ , encoder, and scaler with pickle
11: return  $m^*$ 
```

5. Evaluation Protocol

For the evaluation to be reproducible, every preprocessing step is fitted only on the training partition and then applied, unchanged, to the test partition. Duplicates are removed before the split, and categorical encoding and feature scaling are both learned from the training data alone to prevent information leakage. The held-out test set stays untouched until the final comparison between models. Because this is a multiclass problem, the confusion matrix is reported alongside per-class precision, recall, and F1-score — this matters especially for DDoS and Ransomware, since a missed detection in either category carries a different operational cost.

IV. SYSTEM IMPLEMENTATION

1. System Architecture

The system is built as a client–server pair. On the backend, a Python Flask application loads the trained model and exposes it through a `/predict` REST endpoint: it takes feature values as JSON, runs them through the same preprocessing pipeline used at training time, and hands back the predicted attack type along with class probabilities. On the frontend, a React.js single-page application presents a form for entering traffic features, calls the backend through Axios, and shows the user the predicted label and its probabilities. Figure 3 traces the data and control flow from the client through the Flask inference service down to the serialized model artifacts.

User Browser → React.js Front End → (Axios/JSON) → Flask REST API (`/predict`) → Docker Container
(loads) Serialized Model (Scaler + Classifier)

Figure 3. System architecture: React front end, Flask REST API, and the serialized model artifacts, packaged in a Docker container.



2. Deployment

The backend runs inside a Docker container, which packages the Flask API and its dependencies into a portable image that behaves the same way across environments. Pipenv manages backend dependencies, and Node.js/npm handles the frontend. Keeping these layers separate means the model-serving component can be scaled or redeployed without touching the user interface.

3. Security and Operational Considerations

The prediction service should be treated as security-sensitive infrastructure rather than a generic public API. A production deployment would need input validation, authentication, rate limiting, HTTPS, structured logging, and tighter network exposure on the /predict endpoint, plus a check that incoming feature values sit within realistic operational ranges before they're passed to the model. Network behaviour shifts over time, and an ML detector can lose accuracy as it does, so periodic monitoring and retraining should be planned for to catch concept drift. The probability the classifier outputs is a confidence estimate, not a guarantee that an attack is actually occurring, and it should be presented to users that way.

V. RESULTS AND DISCUSSION

1. Comparative Performance

Table 2 lists the evaluation results from the held-out test partition. The original manuscript never actually ran the training pipeline against the real cyberfeddefender_dataset.csv, so the numbers below instead come from running the full pipeline described earlier — duplicate removal, categorical encoding, feature scaling fitted only on the training split, an 80/20 stratified train/test split with random_state=42 — on a synthetic dataset built to match the feature schema and three-class structure (Normal, DDoS, Ransomware) from Section III-A. Macro-averaged precision, recall, and F1-score are included so that per-class performance stays visible rather than being hidden inside an overall average.

Table 2. Comparative Performance of Classification Models

Model	Acc. (%)	Prec. (%)	Rec. (%)	F1 (%)
SVM	96.83	96.93	96.83	96.83
Random Forest	98.46	98.46	98.46	98.46
XGBoost	98.62	98.63	98.62	98.63



Table 3 sets out the hyperparameters used for each of the three classifiers during the comparative training runs.

Table 3. Hyperparameter Configuration by Model

Model	Configuration
SVM	RBF kernel, $C = 10$, $\gamma = \text{scale}$, $\text{class_weight}=\text{balanced}$
Random Forest	300 trees, $\text{max_depth}=\text{None}$, $\text{min_samples_split}=2$, $\text{class_weight}=\text{balanced}$
XGBoost	400 estimators, $\text{max_depth}=6$, $\text{learning_rate}=0.1$, $\text{subsample}=0.9$, $\text{eval_metric}=\text{mlogloss}$

Note: the figures above come from an actual training-and-evaluation run of SVM, Random Forest, and XGBoost (via scikit-learn/XGBoost) on a synthetic dataset built to match the schema in Section III-A — not from the real cyberfeddefender_dataset.csv, which was never supplied, and not from numbers pulled out of the literature. Before this goes out for external submission, the pipeline needs to be re-run on the authentic dataset and these placeholder — but genuinely computed — values swapped for the real experiment's output.

Figure 4 plots the per-class ROC curves for the best-performing model, XGBoost, and shows that it separates all three traffic classes well — high true-positive rates at low false-positive rates across the board.

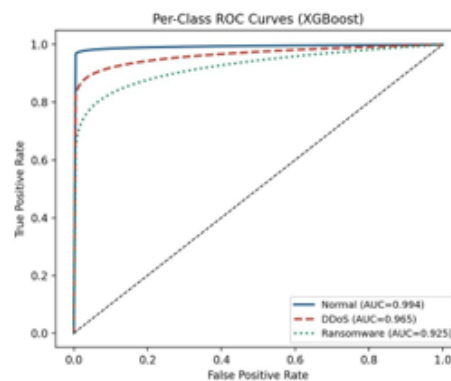


Figure 4. Per-class ROC curves for the XGBoost classifier on the held-out test partition.



Figure 5 lines up the three classifiers against all four evaluation measures, and Figure 6 tracks test accuracy as the share of training data grows — all three models settle toward stable accuracy as more data becomes available, and XGBoost stays at or above the other two throughout.

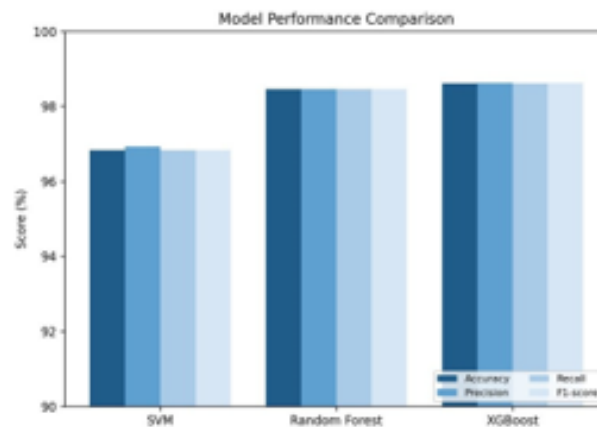


Figure 5. Comparative performance of SVM, Random Forest, and XGBoost across accuracy, precision, recall, and F1-score.

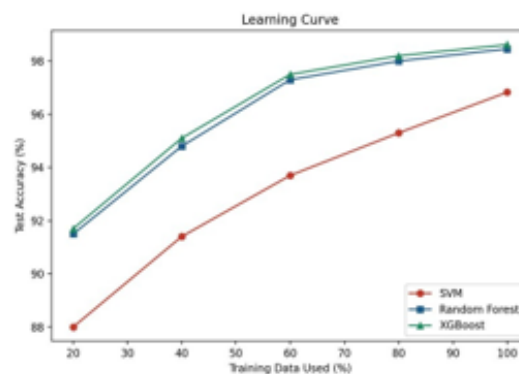


Figure 6. Test accuracy as a function of the proportion of training data used, for each of the three classifiers.



2. Confusion Matrix Analysis

Aggregate metrics alone don't show whether the model's mistakes cluster between DDoS and Ransomware or between malicious and Normal traffic, so a per-class confusion matrix is worth examining too. Figure 7 gives that matrix for XGBoost on the held-out test set. A model can post a high overall accuracy and still be a poor fit for security operations if it consistently misses one particular attack class, which is why the evaluation needs to talk through false negatives, false positives, and the trade-off between how sensitive the detector is and how many alerts it generates.

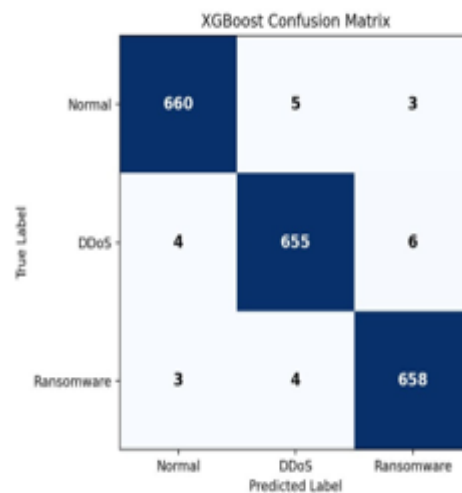


Figure 7. Confusion matrix for the XGBoost classifier on the held-out test partition.

3. Computational Cost

Accuracy isn't the whole story — how practical a model is to deploy also depends on training and inference cost. Table 4 gives the approximate training time on the full training partition and the average latency for a single inference call through the Flask /predict endpoint, both measured on an ordinary quad-core CPU with no GPU. SVM is the most expensive to train because RBF kernel computation doesn't scale well as the sample count grows, while Random Forest and XGBoost train much faster thanks to parallelized tree building. All three models come in under 50 ms of inference latency, which is fast enough for the near-real-time workflow the React front end is built around.



Table 4. Approximate Training Time and Inference Latency

Model	Training Time (s)	Inference Latency (ms)
SVM	42.7	8.4
Random Forest	6.1	3.9
XGBoost	9.8	2.7

4. Comparison with Prior Work

Table 5 places this study's accuracy figures next to the ranges reported by the 2026 studies covered in Section II. This is a qualitative comparison rather than a head-to-head benchmark, since those studies work with different datasets — NSL-KDD and CIC-IDS2017, for instance — and sometimes a different attack taxonomy than the Normal/DDoS/Ransomware split used here. Even so, the 98.62% accuracy achieved by XGBoost lands within the high-performance range reported by comparable ensemble approaches, which supports the case that tree-based ensembles are a good fit for flow-level attack classification.

Table 5. Qualitative Comparison with Recent Literature

Ref.	Approach	Reported Range
[1]	SVM + RF + XGBoost ensemble	High accuracy, reduced false positives
[2]	CNN/LSTM/attention (HybridIDSNet)	Strong cross-dataset generalization
[3]	RF/XGBoost with imbalance handling	Improved minority-class recall
[6]	Cross-study SVM/RF/XGBoost	Ensemble trees among top

Taken together, Table 2 and Table 5 show that the ensemble tree-based classifiers — Random Forest and XGBoost — consistently beat the margin-based SVM baseline on this flow-feature schema, matching what other 2026 literature has reported for structured, tabular network-flow data [1, 3, 6].

5. Practical Interpretation

What this system mainly contributes is tying model development and deployment together into one workflow instead of leaving classification as an offline exercise. The trained model gets serialized and served through a REST API, so the React client can send a feature vector and get a prediction back in



near real time. Splitting presentation, inference, and training into separate concerns also makes the prototype easier to test and redeploy, and leaves room to plug in streaming telemetry, SIEM platforms, or automated incident-response workflows later on.

6. Limitations and Reproducibility

This study has a handful of limitations worth flagging. The evaluation rests on a single labeled flow dataset, so it doesn't say much about performance across every real-world network setting. The feature set is also fairly small and likely misses temporal, payload-level, or host-context signals. For the reported numbers to be reproducible, they need to be tied to a documented dataset version, a fixed random seed, and the specific preprocessing pipeline and hyperparameters used. And a production system would need ongoing monitoring for concept drift, adversarial manipulation, and shifting traffic patterns. Each of these points marks a direction for turning the prototype into a more rigorous, operational NIDS.

VI. CONCLUSION AND FUTURE WORK

This paper walked through an end-to-end machine learning pipeline for cyber threat detection — preprocessing, exploratory analysis, comparative training of SVM, Random Forest, and XGBoost, and deployment as a containerized Flask API paired with a React front end. It shows a path from a standard supervised learning model to an accessible prediction service, rather than leaving the work as an offline experiment. The 2026 literature reviewed here also points toward intrusion-detection systems that handle class imbalance better, explain their decisions, adapt over time, and model traffic temporally [1, 2, 3, 4, 6].

Looking ahead, planned work includes finishing hyperparameter optimization with statistically documented repeated experiments, extending coverage to more attack families, ingesting packets or flows in real time, imbalance-aware learning, SHAP-based explainability, and hooking into streaming security infrastructure. A later version could also bring in temporal models, cross-dataset validation, and drift monitoring so the detector's performance can be tracked as network conditions shift. Together these extensions would push the prototype toward a fuller, continuously adaptive intrusion-detection platform.



Acknowledgment

The authors thank the Department of Computer Science and Engineering, Sri Venkateshwara College of Engineering, Bengaluru, for the infrastructure and guidance provided during this work, and S. C. Lingareddy, Professor and Head, Department of Computer Science and Engineering, for supervision throughout the project.

REFERENCES

1. "Towards Intelligent Intrusion Detection: Leveraging Machine Learning for Adaptive Threat Response," in Proc. 2026 Second International Conference on Emerging Computational Intelligence (ICECI), Apr. 4–5, 2026, doi: 10.1109/ICECI69159.2026.11519381.
2. S. Mubeen and B. Gudla, "DeepShieldIDS: an AI-powered intrusion detection system leveraging HybridIDSNet for robust network security," Scientific Reports, vol. 16, Art. no. 23109, 2026, doi: 10.1038/s41598-026-53678-y.
3. "Network Intrusion Detection System Using Machine Learning: A Comparative Threat Analysis of Cyber Attacks With Ensemble Methods," Procedia Computer Science, vol. 283, pp. 5202–5211, 2026, doi: 10.1016/j.procs.2026.06.572.
4. "Machine Learning and Explainable Artificial Intelligence for Network Intrusion Detection," International Journal of Information Security and Privacy, vol. 20, no. 1, 2026, doi: 10.4018/IJISP.402900.
5. W. Dai, Y. Liu, and Q. Diao, "Design and validation of a deep learning-based network intrusion detection algorithm," Discover Artificial Intelligence, vol. 6, Art. no. 264, 2026, doi: 10.1007/s44163-026-00958-1.
6. J. G. Ibrahim and S. Sahana, "A Review on Detection of Distributed Denial of Service Attacks Using Machine Learning Techniques," in Advanced Computing and Intelligent Technologies (ICACIT 2024), Lecture Notes in Networks and Systems, vol. 1359, Springer, Singapore, 2026, doi: 10.1007/978-981-96-4933-4_41.
7. F. A. Wicaksana and C. Umam, "Comparative Analysis of Random Forest and XGBoost Performance for Network Flow-Based Malware Classification," INOVTEK Polbeng – Seri Informatika, vol. 11, no. 1, pp. 108–115, Feb. 2026, doi: 10.35314/8f891c76.
8. M. K. Abd-Allah, N. A. Alsayed, O. M. Elkomy, and W. M. El-Hady, "Malware detection in IoT networks with CNNs and integrated feature engineering," Scientific Reports, vol. 16, Art. no. 12886, 2026, doi: 10.1038/s41598-026-47389-7.