International Journal of Computer Information Systems and Industrial Management Applications
ISSN 2150-7988 Volume 18(2s) 2026 pp. 495-507
https://doi.org/10.70917/ijcisim-2026-2091
Article

ENHANCING INTRUSION DETECTION THROUGH HYBRID ANOMALY DETECTION: INTEGRATION OF MACHINE LEARNING MODELS WITH RULE-BASED STRUCTURE

Manasi P. Shirurkar[1], Minakshi More[2], Mrunmayee M. Pande[3], Vaishnavi Tapasvi[4], Kanishk Deshpande[5], Atharva Ganesh Kandhare[6]

[1] Department of MCA, MES' IMCC, Pune, India. msu.imcc@mespune.in
[2] Department of MCA, MES' IMCC, Pune, India. mst.imcc@mespune.in
[3] Department of MCA, MES' IMCC, Pune, Indiapandemrunmayee0512@gmail.com
[4] Department of MCA, MES' IMCC, Pune, India.vaishnavi.tapasvi@gmail.com
[5] Department of MCA, MES' IMCC, Pune, India.kanishk.deshpande7@gmail.com
[6] Department of MCA, MES' IMCC, Pune, India atharvakandhare101@gmail.com
Corresponding Author: Manasi P. Shirurkar (msu.imcc@mespune.in)

Abstract: This paper offers a comprehensive hybrid intrusion detection system (IDS) which integrates signature-based threshold detection with machine learning-driven anomaly detection. The system is implemented as a cross-platform desktop application leveraging an Electron-based frontend with a Python backend for real-time monitoring and threat detection. The architecture employs numerous machine learning algorithms containing Random Forest, XG Boost and Support Vector Machines alongside traditional threshold-based detection mechanisms to identify network intrusions and host-based anomalies. The system demonstrates the effectiveness of combining multiple detection methodologies to enhance intrusion detection capabilities while keeping computational competence through optimized threading and alert deduplication mechanisms

Keywords Intrusion Detection Systems (IDS), Hybrid Intrusion Detection System (HIDS), Machine Learning, Artificial Intelligence, Anomaly Based Systems, Signature Based Systems, Rule Based Structure

1. INTRODUCTION

Network anomalies are getting crucial; we need ways to detect them by finding the threats we already know about and the new ones we do not know about yet. The old way of doing things was to use systems that looked for patterns of known attacks which were good at stopping attacks we already know about. [1], [3] Although they are incapable at stopping new attacks that we have failed to witness [2]. There are methods that uses machine learning to find anomalies. Network security threats are a problem and we need to find a better way to deal with them and able to stop them.

This paper describes the architectural execution of a Hybrid Intrusion Detection System that addresses these limitations [7], [9] by combining both approaches. To provide comprehensive threat identification the system monitors network traffic and host system metrics in real-time by applying methodologies of multiple detection simultaneously.

Copyright: © 2026 by the authors

The implementation utilizes an Electron-based GUI for accessibility and cross-platform compatibility, paired with a Python-based backend for efficient data processing and machine learning operations.

The primary offerings of this work are:

A practical architecture for hybrid intrusion detection that embeds threshold-based and machine learning approaches

Implementation of real-time network and host monitoring using cross-platform libraries

Comparative evaluation of various machine learning algorithms for anomaly detection

An efficient alert management system with deduplication and cooldown mechanisms

A user-friendly desktop application for security monitoring and visualization

2. SYSTEM ARCHITECTURE

2.1 Architecture Overview

The Hybrid IDS is consisting of four primary layers visualized in (Figure 1) which makes it as a multi-tier architecture.[10][17]

Presentation layer chains cross-platform operation on Windows, macOS, and Linux through the Electron framework. It is built using Electron v13+ with React, HTML5, and CSS3 (Tailwind CSS), providing a visualization of network statistics, host metrics, alerts, and event logs through responsive graphical interface.

Orchestration layer holds the main IDS monitoring engine implemented in Python 3.8+. The lifecycle of all detection modules is managed with coordination of inter-component communication and handles frontend-backend interaction through a JSON-based protocol via Electron's inter-process communication (IPC) mechanism.

Detection layer includes three primary components: Machine learning based anomaly detection provided by the ML Detector, module network traffic monitoring via the Network Capture module and host system monitoring through the Host Monitor module. Operations are executed independently in separate daemon threads, accessing concurrent data collection and analysis.

Persistence layer utilizes SQLite3 for alert storage and JSON files for statistical data archival, ensuring historical data preservation across system restarts and enabling post-incident analysis.

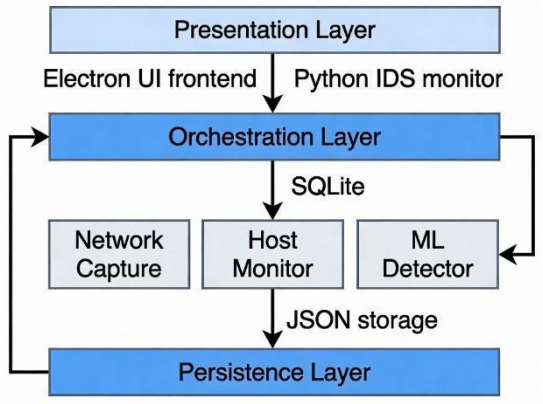

Figure 1: Architecture of the Proposed Hybrid Intrusion Detection System

2.2 Technology Stack

optimizes performance, compatibility, and maintainability of the system.

Frontend:

Electron v13+: cross-platform desktop application development

React: component-based UI architecture

Tailwind CSS: to make design responsive

HTML5 and CSS3: uses semantic markup and styling

JavaScript: logic for client-side and real-time updates

Backend:

Python 3.8+: data processing and machine learning operations

psutil: cross-platform system and network monitoring

scikit-learn: machine learning algorithms [4]

XGBoost: boosting classification [5]

pandas: data manipulation and analysis

NumPy: numerical computations

joblib: efficient model serialization and deserialization

SQLite3: persistent alert storage

Development and Build Tools:

Node.js: runtime execution in Electron

npm: frontend dependency management

pip: Python package management

Python virtual environments: backend dependency isolation

2.3 System Directory Structure

Modular directory structure is used which separates concerns and facilitates maintainability:

Directory/File	**Description**
hybrid-ids-app/	**Root project directory**
backend/	Python backend engine
ids_monitor.py	Main IDS orchestration engine implementation.md
db.py	Database operations and persistence implementation.md
data_collection/	Data collection modules
network_capture.py	Network monitoring implementation.md
host_monitor.py	Host system monitoring implementation.md

Directory/File	Description
ml_detector.py	ML-based anomaly detection implementation.md
ml/	Machine learning pipeline
preprocessing.py	Data preprocessing and feature engineering implementation.md
train_models.py	Model training and evaluation implementation.md
src/	Frontend source code
App.js	Main React application component implementation.md
main.js	Electron main process implementation.md
preload.js	IPC communication bridge implementation.md
data/	Data storage and model repository
raw/	Raw training datasets implementation.md
processed/	Preprocessed datasets implementation.md
models/	Trained ML models implementation.md
alerts/	Alert logs and database implementation.md
network/	Network statistics implementation.md
host/	Host statistics implementation.md
package.json	Node.js dependencies configuration implementation.md

3. DATA COLLECTION METHODOLOGY

3.1 Network Data Collection

Implementation of network monitoring is done through the NetworkCapture module, which operates the psutil library to capture real-time network statistics [2], [3]. Following metrics are monitored:

Per-Interval Network Metrics:

Bytes transmitted across all network interfaces

Bytes received across all network interfaces

Packets transmitted

Packets received

Active network connections count

NetworkCapture class implements the following functions from psutil [14]:

net_io_counters(): Cumulative network I/O statistics is retrieved through bytes and packets sent/received.

net_connections(): Computes active network connections with protocol information

Data Collection Strategy:

Collection of network statistics is done in intervals of one-second and saved to timestamped JSON files in the data/network/ directory. The system tracks the incremental changes done through network counters to calculate per-interval statistics, providing meaningful indicators of network activity rather than cumulative values. This enables detection of anomalous traffic patterns such as:

Sustained high-bandwidth data transfers (potential data exfiltration)

Abnormal packet-to-byte ratios (potential protocol anomalies)

Rapid connection establishment (potential port scanning)

3.2 Host System Data Collection

Implementation of host monitoring is done through the HostMonitor module, which operates the psutil library [14]. Following metrics are monitored:

CPU and Memory Metrics:

CPU usage percentage across all cores

Memory statistics including total, available, used, and percentage utilized

Swap memory information

Disk Metrics:

Disk usage for the root filesystem including total, used, and available space

Process Monitoring:

Process enumeration with PID, process name, CPU percentage, and memory percentage

Top 10 processes sorted by CPU usage for resource anomaly detection

Process count as an indicator of system state

Data Collection Strategy:

Collection of host statistics is done in intervals of one-second and saved to timestamped JSON files in the data/network/ directory. The system maintains a running list of top processes to identify resource-intensive anomalies. This enables detection of:

Recursive high CPU/memory utilization (potential denial-of-service or resource exhaustion attacks)

Abnormal process spawning patterns (potential malware execution)

Unusual process resource consumption ratios

3.3 Data Aggregation and Preprocessing

Preprocessing pipeline, used to convert raw network and host data in a uniform setup appropriate for machine learning implements the ml/preprocessing.py file.

Data Loading and Validation: Raw data is loaded from CSV files in the raw data directory for the comprehensive error handling of malformed files. All required columns are checked in pipeline which consists of : timestamp, src_ip, dest_ip, src_port, dest_port, protocol, packet_size, duration, cpu_usage, memory_usage, process_count, and label.

Data Standardization: The preprocessing pipeline implements a consistent schema across all data, normalizing field names and ensuring data type consistency.[4]

Categorical Encoding: Conversion of string values to numeric representations which is required by machine learning algorithms for protocols such as (TCP, UDP, ICMP) are encoded using scikit-learn's LabelEncoder. [4]

Data Cleaning: Ensuring the dataset has no null entries that could disrupt model training the missing values are handled by filling with zero values.

Feature Scaling: Normalization of numerical features is done using StandardScaler, that focuses each feature around zero with unit variance. This improves model convergence and prediction accuracy for algorithms sensitive to feature magnitude such as Support Vector Machines and K-Nearest Neighbors.

Train/Test Splitting: Training sets (70%) and testing sets (30%) are divided of pre-processed dataset with a fixed random state (42) to ensure reproducible results and allow unswerving model assessment across training iterations.[4]

4. MACHINE LEARNING MODEL DEVELOPMENT

4.1 Model Selection and Comparative Evaluation

Comparative approach to model selection, training and assessing six different machine learning algorithms [12] is conducted by the system to identify optimal detection performance:

Random Forest Classifier: Provides robust anomaly detection with built-in characteristic importance analysis by constructing multiple decision trees and aggregates their forecasts through widely held voting.

Decision Tree Classifier: Recursive partitioning of the characteristic space, offering interpretable decision rules for anomaly identification with tree-based algorithm.

Naive Bayes (GaussianNB): A Bayes' theorem based on probabilistic classifier, effective for high-dimensional feature spaces and provides probability estimates for predictions.

K-Nearest Neighbors (k=5, kd_tree algorithm): Instance categorization for efficient neighbor search using non-parametric algorithm based on the majority label of k nearest neighbors, leveraging kd-tree.

Support Vector Machine (LinearSVC): Finds ideal hyperplanes to maximize margin between classes, effective for high-dimensional data using linear classifier

XGBoost Classifier: Chronological build of weak learners to minimize classification error, reaching state-of-the-art performance on many datasets with gradient boosting framework. [5]

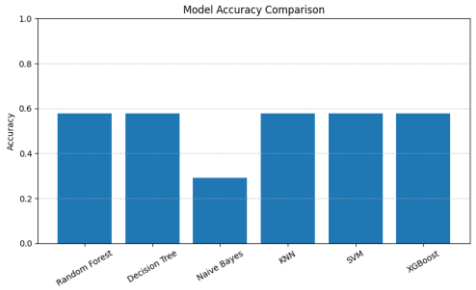

Figure 2: Accuracy Comparison of Machine Learning Models for Intrusion Detection

Model Training Process

:Normalized training dataset using scikit-learn's consistent API for each model (Figure 2) where they are evaluated on the held-out test set using:

Accuracy Score: Overall classification

Precision, Recall, and F1-Score: Performance metrics of each class

Confusion Matrix: Classification performance breakdown

Model-Persistence:
Trained models are serialized using pickle format and stored in the data/models/ directory for deployment in the real-time detection engine.

4.2 Feature Extraction for Real-Time Detection

The MLDetector module extracts features from live-collected network and host data for real-time anomaly prediction [6]:

Network Features:

Bytes sent in the existing interval

Bytes received in the current interval

Packets sent in the present interval

Packets received in the contemporary interval

Number of active network connections

Host Features:

CPU usage percentage

Memory usage percentage

Disk usage percentage

Number of running processes

These features are extracted at regular intervals (typically 3-second processing intervals) and passed through the trained machine learning models to generate real-time anomaly predictions. The feature extraction maintains consistency with the preprocessing pipeline, applying the same categorical encoding and normalization transformations used during model training.

5. INTRUSION DETECTION ENGINE

5.1 Real-Time Monitoring Architecture

The entire intrusion detection process orchestrates IDSMonitor class through multi-threaded concurrent operations:

Component-Initialization:
The IDSMonitor initializes three primary detection components:

NetworkCapture instance for network monitoring

HostMonitor instance for host system monitoring

MLDetector instance for machine learning-based anomaly detection

Threading-Model:
The system employs multi-threading for concurrent operations:

Main processing thread executes the core detection loop

Separate daemon threads for NetworkCapture, HostMonitor, and MLDetector operate independently

Input handling thread processes commands from the frontend application

Processing-Schedule:
The main detection loop executes at the following intervals:

Alert processing: Every 3 seconds

Network statistics processing: Every 2 seconds

Host statistics processing: Every 3 seconds

Event log generation: Every 5 seconds

5.2 Multi-Layer Alert Generation

The system generates alerts through four complementary mechanisms [8] also visualized in Figure 3:

Layer 1: Threshold-Based Detection:

Anomalous metric values with predefined threshold trigger alerts:

Network: >100 KB outbound/inbound per interval

Host: >80% CPU usage, >80% memory usage

Connection: >10 concurrent active connections

Process count anomaly: Baseline unusual deviations

Layer 2: Machine Learning-Based Detection [15]:

Trained ML models predict anomalies on extracted feature vectors. An alert is generated with the equivalent anomaly confidence when any model's prediction equals 1 (indicating anomaly)

Layer 3: Pattern Recognition for Attacks:

Domain-specific patterns are detected:

Port Scanning: High packet counts to multiple destinations with varying ports

Data Exfiltration: High outbound bytes with few active connections

Brute Force Attacks: Many connection attempts with high packet counts

Resource Exhaustion: Sustained CPU/memory utilization above thresholds

Layer 4: Simulation Mode:

For testing and validation when actual data collection is unavailable, the system generates synthetic test alerts following realistic attack patterns.

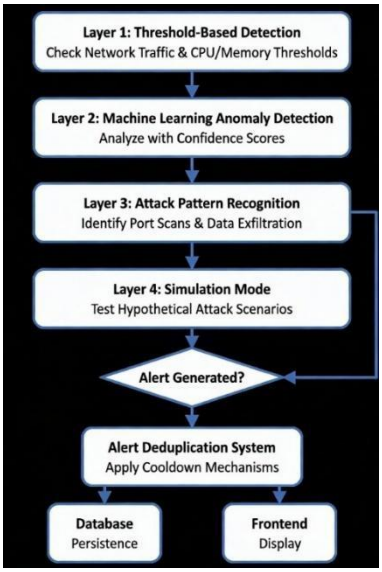

Figure 3: Multi-Layer Intrusion Detection Workflow of the Hybrid IDS

5.3 Alert Management and Deduplication

To prevent alert flooding and maintain system usability, the system implements sophisticated alert management:

Cooldown Mechanism:

Prevents duplicate alerts from overwhelming security personnel to identical alerts are suppressed for a 30-second cooldown period after initial generation.[14]

Message Uniqueness Checking:

History of recent alert messages and suppresses alerts is maintained with identical content, deduplicating alerts [14] from multiple detection layers.

Alert History Management:

100 most recent entries to maintain memory efficiency while preserving sufficient historical context for analysis.

Database Persistence:

To ensure historical alert data survives system restarts and enabling post-incident forensic analysis alerts are persevered to an SQLite database via the db.py module.

5.4 System Performance Optimization

Several optimization strategies ensure responsive operation while managing computational resources:

Resource Limiting:

Top 10 processes monitored by CPU usage

100 recent alert entries in history

In-memory data structures sized to prevent memory exhaustion.

Responsive Threading:

To balance responsiveness and CPU utilization sleep intervals between processing iterations are carefully tuned (0.1-1.0 seconds)

Incremental Computation:

Incremental changes rather than cumulative values, reducing computational overhead while providing meaningful per-interval statistics are used by network counter calculations.

Model Serialization:

Trained models are deserialized using joblib, which provides efficient binary serialization compared to pickle, reducing model loading overhead. [10] [16]

6. FRONTEND IMPLEMENTATION AND USER INTERFACE

6.1 Electron Application Architecture

Main Process (src/main.js):

Creates and manages the main BrowserWindow with security-hardened webPreferences

Spawns the Python backend process using child_process.spawn()

Establishes IPC communication channel between frontend and backend

Implements system tray functionality for minimized background operation

Preload Script (src/preload.js):

Implements secure context isolation using Electron's contextBridge

Exposes safe APIs for frontend-backend communication

Prevents direct access to Node.js APIs from the renderer process

Implements JSON serialization for inter-process messages

Renderer Process (src/App.js):

React component architecture for state management and real-time updates

Implements efficient re-rendering through component lifecycle management

Uses Electron IPC for bi-directional backend communication

Uses Tailwind CSS for cross-platform compatibility with responsive design [13], [15]

6.2 User Interface Components

Real-Time Monitoring Dashboard:

Network statistics visualization with line charts showing bandwidth utilization

Host statistics with pie charts for resource distribution

Alert timeline with color-coded severity levels

Live event log displaying system status and anomalies

Alert Visualization:

Chronological alert listing with timestamp, type, and severity

Alert filtering and search capabilities

Attack pattern categorization with visual indicators

Detailed alert information including triggering metrics

Configuration and Control:

Start/stop controls for IDS operation

Model selection interface for switching detection algorithms

Threshold adjustment controls for fine-tuning detection sensitivity

Data export functionality for external analysis

7. EVALUATION AND PERFORMANCE CONSIDERATIONS

7.1 Machine Learning Model Evaluation Metrics

Classification Metrics:

Accuracy: Overall proportion of correct predictions

Precision: Proportion of affirmative forecasts that are correct

Recall: Proportion of actual positives correctly identified

F1-Score: Harmonic mean of precision and recall [12]

Comparative-Analysis:
Cross-algorithm evaluation enables selection of optimal models based on specific performance priorities in a way such as maximizing recall for sensitive environments versus maximizing precision for reducing false positives.

7.2 System Resource Requirements

The hybrid IDS is designed for minimal resource footprint:

CPU Usage:

Data collection modules: Minimal overhead (network and host monitoring via psutil)

ML detection: Negligible (feature extraction and single prediction per interval)

Frontend: Minimal except during visualization updates

Memory Usage:

Alert history limited to 100 entries

Process list capped at top 10

Model loaded once at startup

Disk Usage:

Model files: Typically, 10-50 MB per model

Alert database: Grows with operational duration

Statistical data: Archived to data/network/ and data/host/ directories

7.3 Detection Latency

Real-time threat detection is achieved through:

Network capture at 1-second intervals

Host monitoring at 1-second intervals

Alert processing every 3 seconds

Machine learning inference latency: Milliseconds for single predictions

Maximum detection latency is approximately 3 seconds from metric collection to alert generation and frontend display.[11]

8. COMPARISON WITH EXISTING IDS APPROACHES

8.1 Advantages of the Hybrid Approach

Over Signature-Based Systems:

Capability to detect novel, previously unseen attack patterns through machine learning

Reduced dependency on signature updates

Ability to identify subtle anomalies [9] that deviate from baseline behaviour

Over Machine Learning-Only Systems:

Lower false positive rates through confirmed threshold violations

Interpretable detection rules through threshold-based mechanisms [15]

Reduced computational requirements through filtered alert generation

Domain expert knowledge integration via pattern recognition

Over Traditional Network-Based IDS:

Host-based metrics detection for internal threats

Cross-platform compatibility through Electron

User-friendly graphical interface for accessibility

Simplified deployment as a single desktop application

9. LIMITATIONS AND FUTURE WORK

9.1 Current Limitations

Data Dependency: ML detection quality depends on training dataset quality and depiction of attack patterns which requires comprehensive labelled training data for optimal performance.

Computational Constraints: Complexity of machine learning models is controlled in real-time processing that can be deployed. More sophisticated ensemble methods may be constrained by latency requirements.

Threshold Tuning: Environment-specific tuning is used for effective threshold-based detection [7] which generates false positives or negatives in different network conditions.

Encrypted Traffic: Network-based detection cannot inspect encrypted payload content, limiting ability to detect application-layer attacks within encrypted tunnels. [7][17]

9.2 Future Enhancement Directions

Advanced Machine Learning:

Integration of deep learning models for complex pattern recognition. [11]

Ensemble methods combining multiple model predictions with weighted voting

Online learning approaches for continuous model adaptation

Detection Capabilities:

Application-layer protocol analysis for encrypted traffic

Behavioral baselining with automatic threshold generation

Integration with threat intelligence feeds for known attack signature updates

Scalability Improvements:

Distributed architecture for enterprise deployment

Cloud-based model training and inference

Horizontal scaling for large network environments. [13][16]

User Experience Enhancement:

Interactive threat playbooks and response recommendations

Integrates external alerting systems (email, Slack, Splunk)

Customizable detection rules and attack pattern definitions

10. CONCLUSION

The study presents a collective Hybrid Intrusion Detection System that effectively combines signature-based threshold detection with machine learning-driven anomaly detection. The system proves that hybrid methods can provide superior threat detection capabilities compared to single-methodology systems [8][17], leveraging the corresponding strengths of both approaches.

The implementation provides a practical, user-friendly platform for real-time network and host-based intrusion detection through an interactive graphical interface. Easy extension and customization for specific organizational requirements is satisfied by modular architecture. The system can be optimized for specific deployment environments and threat profiles by training and evaluating multiple machine learning algorithms.

Threshold-based and machine learning detection mechanisms associates with sophisticated alert deduplication and resource optimization, enables responsive intrusion detection suitable for both personal and small-to-medium enterprise environments. Future work will focus on advanced machine learning integration, scalability for large network deployments, and enhanced user experience features.

Demonstrating practical implementation strategies for hybrid detection systems and providing a reusable foundation for security monitoring applications across different platforms in the field of intrusion detection is the most important contribution achieved by this research.

References:

1. Denning, D. E. (1987). An intrusion detection model. IEEE Transactions on Software Engineering, (2), 222-232.
2. Ansam Khraisat, Iqbal Gondal, Peter Vamplew and Joarder Kamruzzaman (2019). Survey of intrusion detection systems: techniques, datasets and challenges. Journal of Cybersecurity, 2(1), 20.
3. Liao, H. J., Lin, C. H. R., Lin, Y. C., & Tung, K. Y. (2013). Intrusion detection system: A comprehensive review. Journal of Network and Computer Applications, 36(1), 16-24.
4. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Dubourg, V. (2011). Scikit-learn: Machine Learning in Python
5. Chen, T., & Guestrin, C. (2016). XGBoost: A scalable tree boosting system. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (pp. 785-794).
6. Gisung Kim, Seungmin Lee, Sehun Kim (2014). A novel hybrid intrusion detection method integrating anomaly detection with misuse detection.
7. Elijah M. Maseno, Zenghui Wang and Hongyan Xing (2022). A Systematic Review on Hybrid Intrusion Detection System
8. Emma O, Mike Welder (2025). Hybrid Approaches to Intrusion Detection: Combining Machine Learning and Rule-Based Systems
9. Daniel Kofi Odame Bampoe (2025). The Advantages of Hybrid Intrusion Detection Systems: A Comparative Study of Anomaly-Based and Signature-Based Approaches
10. Megha Gupta (2015). Hybrid Intrusion Detection System: Technology and Development
11. Bambang Susilo, Abdul Muis and Riri Fitri Sari (2025). Intelligent Intrusion Detection System Against Various Attacks Based on a Hybrid Deep Learning Algorithm
12. Ashwani Attri, Priyanka Gundeboyena, Vaishnavi Chigurla, Soumika Moluguri and Nithin Kasoju (2025). Network intrusion detection using hybrid approach
13. AMAL MERSNI, NEDŽLA ŠEHOVIĆ, NEJIRA SUBAŠIĆ-HODŽA, NUR RUSTEMPAŠIĆ, MURIS ČELJO (2025) Hybrid Intrusion Detection System for Small and Medium Enterprises
14. Urooj Aslam, Ezzat Batool, S. Nadeem Ahsan and Abdullah Sultan (2017). Hybrid Network Intrusion Detection System Using Machine Learning Classification and Rule Based Learning System
15. Rahul Mohite & Lahcen Ouarbya (2024). Interpretable Anomaly Detection: A Hybrid Approach Using Rule-Based and Machine Learning Techniques
16. M. P. Shirurkar and M. More, “Implementation of the NSL-KDD Dataset to Study the Naive Bayes Algorithm for Intrusion Detection Systems,” Panamerican Mathematical Journal, vol. 35, no. 4s, 2025.
17. “A Guide to Comprehending Cybersecurity,” 2024. [Online]. Available: https://www.researchgate.net/publication/399339089. [Accessed: Mar. 27, 2026].