Abstract
Intrusion Detection Systems (IDS) often require substantial computational resources due to the dynamic and unpredictable nature of network traffic. Efficient feature selection plays a critical role in enhancing both detection accuracy and processing speed. In this study, a Random Forest-based feature selection technique is used to eliminate redundant and irrelevant features. Additionally, Particle Swarm Optimization (PSO) is applied to the NSL-KDD dataset to further refine the selected features, reducing false positives and enhancing overall performance. PSO avoids overfitting by selecting only the most relevant features.

To address the high computational cost of traditional PSO, we develop a parallel multiprocessing framework to accelerate the optimization process. The proposed system achieves a **14× speedup** over the serial version and improves IDS accuracy while selecting as few as **2 features**. Experimental results show that our parallel PSO-based feature selection significantly enhances intrusion detection effectiveness, making the solution viable for real-time, large-scale deployment scenarios.

Keywords: Intrusion Detection System (IDS), Feature Selection, Random Forest, Particle Swarm Optimization (PSO), Parallel Computing, Classification, Network Security, Machine Learning, NSL-KDD Dataset, Computational Efficiency, Real-Time Detection.

Introduction

Intrusion Detection Systems (IDS) play a vital role in safeguarding network infrastructure against malicious threats and cyber-attacks. As a defensive mechanism, IDS continuously monitors system activities and network traffic to detect suspicious behaviors and generate timely alerts. These systems uphold the foundational security principles of confidentiality, integrity, and availability by countering threats such as spoofing, denial-of-service (DoS) attacks, traffic analysis, and zero-day vulnerabilities. Broadly, IDS can be categorized into signature-based and anomaly-based systems. While signature-based IDS detect known threats by matching pre-defined patterns with incoming data, anomaly-based IDS are designed to detect previously unseen attacks by identifying deviations from normal behavior. However, the effectiveness of anomaly-based detection is often hampered by high false-positive rates (FPR), and the performance of both types of IDSs deteriorates as the complexity and dimensionality of data increase.

To mitigate the challenges posed by high-dimensional data, feature selection (FS) has become an essential preprocessing step in modern IDS frameworks. FS not only reduces computational complexity but also improves classification accuracy by eliminating irrelevant and redundant features. However, identifying optimal feature subsets from large-scale datasets remains a challenging task, especially in real-time intrusion detection environments.

Particle Swarm Optimization (PSO), a nature-inspired metaheuristic algorithm, has been widely applied in feature selection due to its simplicity, efficiency, and ability to converge towards optimal solutions. Nevertheless, the standard PSO algorithm can be computationally intensive and slow when applied to high-dimensional datasets such as NSL-KDD, which is a refined benchmark dataset derived from the KDD Cup 99. To address this limitation, this research proposes a parallel

implementation of the PSO algorithm, leveraging modern parallel computing techniques to accelerate the feature selection process without compromising detection accuracy.

The primary objective of this study is to enhance the performance of IDS by integrating a parallelized PSO-based feature selection mechanism. By distributing the computational load across multiple processors, the proposed approach aims to reduce execution time significantly while maintaining or improving the precision of intrusion detection. Experimental results on the NSL-KDD dataset demonstrate that the proposed parallel PSO method achieves higher accuracy and a lower false alarm rate compared to traditional methods, making it a promising solution for real-time IDS applications.

Dataset:

The NSL-KDD dataset is an enhanced version of the KDD Cup 99 dataset, designed to address its limitations. It comprises 41 features and a 42nd class attribute that categorizes each record into one of five classes: normal or one of four attack types. These attack classes are as follows:

Denial of Service (DoS): Attacks in this category aim to make system resources unavailable to legitimate users. Common examples include smurf, teardrop, SYN flooding, and neptune.

User to Root (U2R): These attacks involve an intruder gaining root or administrative privileges by exploiting vulnerabilities in the local system. Examples include rootkit, buffer-overflow, spy, and SQL attacks.

Remote to Local (R2L): Here, attackers attempt unauthorized access to a system from a remote location. Examples include warezmaster, imap, multihop, and spy.

Probe: These attacks are used to gather information about a network through techniques such as traffic analysis. Examples include port-scan, satan, ping-sweep, and nmap.

The 41 features in the dataset are categorized into four distinct groups:

Basic Features: These are derived from individual TCP connections and consist of 9 attributes, detailed in Table 1.

Content Features (Domain Knowledge): These 13 features are based on the data within a connection and are outlined in Table 2.

Traffic Features: Comprising 9 attributes, these are computed over a 2-second time window and described in Table 3.

Host-Based Features: This group contains 10 attributes used to detect long-duration attacks (lasting over 2 seconds), detailed in Table 4.

Figures 1 and 2 present statistical comparisons of the KDD Cup 99 and NSL-KDD datasets, respectively. The NSL-KDD dataset is structured to include 22 different types of attacks in the training set and 17 additional attack types in the test set. Figure 3 illustrates the frequency distribution of various attacks in the NSL-KDD dataset.

Introduction to Particle Swarm Optimization (PSO)

Particle Swarm Optimization (PSO) is a population-based metaheuristic optimization algorithm inspired by the social behavior of bird flocking or fish schooling. Introduced by Kennedy and Eberhart in 1995, PSO simulates a group of particles (candidate solutions) moving through a multidimensional search space to find the optimal solution to a given problem. Each particle adjusts its position based on its own experience and the experience of neighboring particles, guided by two key components: the best position found by the particle itself (personal best) and the best position found by the entire swarm (global best). Through iterative updates of velocity and position, particles converge toward optimal or near-optimal solutions. Due to its simplicity, low computational cost, and ability to effectively explore complex search spaces, PSO has been widely adopted for solving optimization problems across various domains, including feature selection, neural network training, and engineering design.

4. Methodology

This study presents a hybrid approach for Intrusion Detection using feature selection based on Particle Swarm Optimization (PSO). The methodology comprises five key phases: data preprocessing, feature ranking using Random Forest, feature selection using PSO, classifier training, and parallelization for performance enhancement. Both **serial and parallel** versions of PSO are implemented and compared to evaluate speed and detection performance.

1. **Data Preprocessing**
    - **Dataset:** The NSL-KDD dataset is used as the benchmark for intrusion detection. The dataset contains 41 features along with a class label indicating whether the connection is normal or an attack.
    - **Feature Engineering: Column headers were assigned for easier manipulation. Categorical features (protocol_type, service, flag) were encoded using label encoding. The difficulty column was removed as it was not relevant for the model.**
    - **Label Binarization: Multi-class labels were converted to a binary classification. Labels were transformed such that normal traffic was mapped to class 0 and attack types to class 1.**
    - **Feature Scaling: Standardization was applied to ensure that all features were on the same scale.**

- Restricting to the first 10,000 records for both training and testing for faster iteration and analysis.

2. **Initial Feature Ranking using Random Forest**
    - **Random Forest Model: The Random Forest classifier was trained on the full feature set** to assess feature importance**, and the top 10 most important features were selected** as the candidate features for PSO-based optimization. This step reduces the initial search space and computational overhead, while enhancing PSO convergence.
1. **Particle Swarm Optimization (PSO) for Feature Selection**
    - **Particle Representation: Each particle in the swarm represents a binary vector** of length equal to the number of candidate features (10)**, indicating selected features (1) or unselected features (0).**
    - **PSO Dynamics: Particles are updated based on cognitive, social, and inertia components, guided by their individual and global best positions.**
    - **Fitness Functions:** The **fitness function** evaluates each particle by:
        - Sub-setting the training and test data to the selected features.
        - Training a `MLPClassifier` (Neural Network) with fixed parameters: 1 hidden layer with 50 neurons, alpha = 0.001.
        - Measuring the **accuracy** of the model on the test data.
        - Returning the accuracy as the fitness value.

      This process is repeated for each particle over multiple iterations (default: 30), with standard PSO velocity and position updates used to explore the feature subset space.
    - **Two PSO Variants:**
        - **Serial PSO: Fitness evaluations are performed sequentially for each particle.**
        - **Parallel PSO: Uses Python's multiprocessing library to distribute fitness computations across multiple cores for faster convergence.** The `starmap()` function is used to evaluate particles in parallel with respect to their selected features.
        - Both **serial** and **parallel** versions of PSO are run under the same configuration (same number of particles and iterations) to allow for direct comparison of execution time, speedup factor, selected features and classification accuracy.
3. **Final Model Training and Evaluation**
    - **After PSO convergence, the optimal feature subset or hyperparameters were used to train the final MLP classifier.**
    - **Performance Metrics: Model performance was evaluated using accuracy, precision, recall, and F1-score on the test set.**
4. **Speedup Analysis**

- **Execution Time: The execution time of serial and parallel PSO was measured to calculate speedup, which was defined as the ratio of serial execution time to parallel execution time.**

1. Particle Swarm Optimization (PSO)

PSO is an optimization algorithm inspired by the social behavior of birds flocking or fish schooling. The position and velocity updates for each particle in the swarm are given by the following formulas:

- **Velocity Update Formula**:

$$v_i^{(t+1)} = w \cdot v_i^{(t)} + c_1 \cdot r_1 \cdot (pbest_i - x_i^{(t)}) + c_2 \cdot r_2 \cdot (gbest - x_i^{(t)})$$

Where:

- vi(t)v_i^{(t)}vi(t) is the velocity of particle iii at iteration ttt,
- www is the inertia weight that controls the exploration-exploitation tradeoff,
- c1,c2c_1, c_2c1,c2 are the cognitive and social coefficients (influencing the pull towards the individual best position and global best position, respectively),
- r1,r2r_1, r_2r1,r2 are random numbers between 0 and 1,
- pbestipbest_ipbesti is the personal best position of particle iii,
- gbestgbestgbest is the global best position across all particles.

- **Position Update Formula**:

- **Position Update Formula**:

$$x_i^{(t+1)} = x_i^{(t)} + v_i^{(t+1)}$$

Where:

- $x_i^{(t)}$ is the position of particle i at iteration t.

PSO Algorithm Step-by-Step

Given:

- Objective function f(x) to minimize or maximize.
- Number of particles N, dimensions D.

Steps:

1. **Initialize** positions and velocities randomly for all particles.
2. For each particle:
    a. Evaluate fitness f(xi).
    b. Update personal best pi .
3. Update global best g among all pi .
4. For each particle:
    a. Update velocity vi using the formula above.
    b. Update position xi .
5. Repeat steps 2–4 for a set number of iterations or until convergence.

2. Fitness Function for Feature Selection

In your implementation, the fitness function is used to evaluate the quality of a feature subset selected by a particle. The fitness is based on the accuracy or F1-score of the classification model (MLP classifier). The formula for the fitness function is:

$$\text{Fitness}(p) = \frac{TP}{TP + FP + FN + TN}$$

Where:

- TP = True Positives,
- FP = False Positives,
- FN = False Negatives,
- TN = True Negatives.

Alternatively, if you use **F1-score** as the fitness metric, the formula is:

3. Feature Selection with Random Forest

$$\text{F1-score}(p) = 2 \cdot \frac{Precision(p) \cdot Recall(p)}{Precision(p) + Recall(p)}$$

Where:

- $Precision(p) = \frac{TP}{TP+FP}$,
- $Recall(p) = \frac{TP}{TP+FN}$.

You use **Random Forest** for feature selection by ranking features based on their importance. The importance of each feature iii is computed by:

$$\text{Importance}_i = \frac{1}{T} \sum_{t=1}^{T} \left(\text{Imp}_{i,t} \right)$$

Where:

- Impi,t\text{Imp}_{i,t}Impi,t is the importance of feature iii in tree ttt in the random forest,
- TTT is the total number of trees in the random forest.

4. Model Performance Metrics

For evaluation, you compute the **Accuracy** and **F1-score** of the final model, which can be expressed as:

- **Accuracy**:
- $$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$ **F1-score** (as explained above).

5. Speedup Calculation

To evaluate the performance improvement from parallelization, you compute the speedup as:

$$\text{Speedup} = \frac{\text{Time for Serial PSO}}{\text{Time for Parallel PSO}}$$

This formula measures how much faster the parallel implementation is compared to the serial one.

6. PSO Parameter Settings

You can mention the key parameters used for PSO:

- c1=0.5c_1 = 0.5c1=0.5, c2=0.3c_2 = 0.3c2=0.3, w=0.9w = 0.9w=0.9 (cognitive, social coefficients, and inertia weight),
- Number of particles: N=20N = 20N=20,
- Number of iterations: I=30I = 30I=30.

Algorithm: Intrusion Detection using Serial and Parallel PSO with MLP

Step 1: Data Preprocessing

1. Load KDD dataset (KDDTrain+.txt and KDDTest+.txt) and assign column names.

2. Drop the "difficulty" column from both datasets.
3. Encode categorical features (protocol, service, and flag) using Label Encoding.
4. Convert class labels to binary: normal as 0, others as 1 (anomaly).
5. Use Random Forest to compute feature importances and select the top 10 features.
6. Standardize the selected features using StandardScaler.

Step 2: Define Fitness Function

1. A binary particle represents a subset of features (1 = selected, 0 = not selected).
2. For each particle:
    - Select features based on its binary vector.
    - Train an MLPClassifier on training data.
    - Predict on validation/test data.
    - Return the classification accuracy as the fitness score.

Step 3: Serial PSO Optimization

1. Initialize N particles randomly with binary vectors.
2. Initialize velocities for each particle.
3. Set personal best (pbest) positions and scores for each particle.
4. Determine the global best (gbest) among all particles.
5. For T iterations:
    - Update each particle's velocity and position.
    - Clip position values between 0 and 1.
    - Evaluate fitness and update pbest if improved.
    - Update gbest if a new global best is found.
6. Return the best feature subset found.

Step 4: Parallel PSO Optimization

1. Same initialization as Serial PSO.
2. Instead of evaluating fitness sequentially, use Python’s multiprocessing.Pool to:
    - Evaluate multiple particles in parallel using the same MLP fitness function.
3. Continue for T iterations:
    - Update positions and velocities.
    - Perform parallel fitness evaluation.
    - Update pbest and gbest based on parallel scores.

4. Return the best feature subset found.

Step 5: Evaluation and Comparison

1. Measure and compare time taken for Serial and Parallel PSO.
2. Calculate speedup as Serial Time / Parallel Time.
3. Train an MLPClassifier using best feature subsets from both PSO approaches.
4. Evaluate model performance using:
    - Accuracy
    - Classification report (Precision, Recall, F1-Score)

Results

1. **Time Efficiency Comparison**:
    - **Serial PSO**: The serial PSO took **1798.4249 seconds** to perform the feature selection and optimization process.
    - **Parallel PSO**: The parallel PSO, utilizing multiple processors/threads, reduced the time significantly to **127.5708 seconds**.
    - **Speedup**: The parallel implementation achieved a speedup of **14.10x**, indicating a substantial improvement in computational efficiency when parallelized.
2. **Feature Selection**:
    - **Serial PSO** selected the following features: **[0, 2, 3, 4, 5, 6, 8]**. This set of features was deemed the most relevant for the classification task.
    - **Parallel PSO** selected a smaller subset of features: **[3, 1]**, which could indicate that the parallel approach found a more optimal or focused feature subset, potentially offering better predictive power or more efficient use of resources.
3. **Model Performance**:
    - **Serial PSO**:
        - **Accuracy**: 0.7663
        - The classification report shows the following key metrics for the Serial PSO model:
            - **Precision** for Class 0: 0.66
            - **Precision** for Class 1: 0.95
            - **Recall** for Class 0: 0.95
            - **Recall** for Class 1: 0.63
            - **F1-score** for Class 0: 0.78

        - **F1-score** for Class 1: 0.76
    - The model demonstrated an overall **accuracy of 0.77**. The macro-average F1-score is 0.77, and the weighted average F1-score is 0.76.
- **Parallel PSO**:
    - **Accuracy**: 0.7835
    - The classification report for the Parallel PSO model showed an improvement in precision and recall for both classes:
        - **Precision** for Class 0: 0.74
        - **Precision** for Class 1: 0.82
        - **Recall** for Class 0: 0.76
        - **Recall** for Class 1: 0.80
        - **F1-score** for Class 0: 0.75
        - **F1-score** for Class 1: 0.81
    - The model achieved a slightly better **accuracy of 0.78**. The macro-average F1-score is 0.78, and the weighted average F1-score is 0.78. A notable result is a higher recall for class 1, which is a better result for correctly detecting attacks.

4. **Convergence Warning**:
    - Both the serial and parallel PSO models triggered a convergence warning from the MLP classifier, indicating that the optimization did not fully converge within the default 300 iterations. This could suggest that further tuning of the MLP training parameters might improve performance.

Interpretation of Results

- **Time Efficiency**: The significant speedup of 14.10x using parallel PSO is a key finding. This highlights the benefit of parallel computing, making the PSO optimization more feasible for larger datasets and more complex optimization tasks.
- **Feature Selection**: The parallel PSO method selected fewer features than the serial PSO. This suggests that the parallel version may have been able to converge on a simpler and more relevant subset of features, potentially reducing overfitting and improving generalization.
- **Model Performance**: Although the serial PSO showed decent performance with an accuracy of 0.7663, the parallel PSO slightly outperformed it with an accuracy of 0.7835. The higher accuracy and better F1-scores for both classes in the parallel PSO indicate that the parallelized version was able to fine-tune the model more effectively.

These results suggest that the parallelization of PSO not only improves the computational efficiency but also leads to a better-performing model with a more focused feature selection. The slight

improvement in accuracy and F1-scores indicates that the parallel approach has significant practical value in both time-sensitive and performance-critical applications.