End of Semester Report

Doron Aloni

July 11, 2025

RNN - implementation and results

1 Features

a. The system-call ID sequence is a time-ordered series that records, step by step, every interaction that a process makes with the operating system. Each system call (for example, open, read, write) is first converted from its name into a compact integer index. As the process runs, its log becomes a vector of these indices $[s_1, s_2, \ldots s_T]$, where s_t denotes the t-th call in time.

b. The paper defines the features of the KDD Cup, NSL-KDD, Kyoto, and UNSW-NB15 datasets in detail, focusing on structured network-based attributes. However, those feature sets simply don't exist in ADFA-LD, which is a host-based syscall trace corpus. Because our data contains only ordered lists of system calls, we cannot reuse the paper's attributes. Instead, we define our sole input feature as the system call ID sequence: mapping each syscall number to a unique integer, fixing the length to 400 steps, and embedding those IDs into the LSTM. We chose 400 steps as it strikes a good balance between keeping the model size manageable and covering a large portion of the data, approximately 72% of the sequences are shorter than this length. This choice directly matches the format of ADFA-LD and enables the network to learn temporal syscall patterns indicative of host-based anomalies

c. The system-call ID sequence captures the most fine-grained view of process behavior—every request made to the operating system—so any deviation from normal patterns stands out directly in the data. Anomalies often manifest as:

- **Unexpected call types**: for example, a normal process rarely invokes `execve` or `chmod` in quick sequence, whereas an exploit might.
- **Unusual ordering**: malicious processes often perform a tightly ordered setup sequence, such as opening sensitive files, changing permissions, and loading libraries, immediately before executing the exploit. Such a precise chain of calls rarely occurs in normal processes and may signal a potential attack.
- **Burst patterns**: rapid repetition of sensitive calls that can signal automated scanning or privilege escalation.

Although we do not process full traces, analyzing a fixed-length prefix preserves enough temporal structure to capture both short-term anomalies and long-range dependencies. This allows the LSTM to learn meaningful behavioral patterns without relying on handcrafted features or summaries.

d. Each `.txt` file is read, syscalls are tokenized and mapped to integer IDs using a vocabulary built from the training set, then sequences are padded or truncated to a fixed length. Further details are provided in the ML Technique Implementation Details section.

2 ML technique parameters

While the paper outlines the general RNN architecture, comprising an input LSTM layer, a sigmoid-activated hidden layer, and a softmax output, it does not specify numerical values for key hyperparameters such as the embedding size, hidden state dimension, number of layers, dropout rate, optimizer settings, batch size, or number of training epochs. We therefore selected these values based on standard practices in sequence modeling and empirical tuning on the ADFA-LD validation set. The embedding size (256) and hidden state dimension (512) offer a good trade-off between model capacity and computational efficiency. Two LSTM layers are used to allow the model to capture higher-order temporal dependencies, while a dropout rate of 0.3 helps prevent overfitting. The Adam optimizer with an initial learning rate of 10^{-3} provides robust convergence. We also incorporated a learning rate scheduler (ReduceLROnPlateau) to decrease the learning rate when the validation loss stagnates, further improving convergence. Training was performed with a batch size of 64 and used early stopping (patience of 3 epochs, up to a maximum of 30 epochs) to avoid overtraining. In practice, the final model stopped after 23 epochs. The final hyperparameters are summarized in Table 1.

Hyperparameter	**Value**
Embedding dimension	256
Hidden state size	512
Number of LSTM layers	2
Dropout rate	0.3
Optimizer	Adam (learning rate 10^{-3})
Batch size	64
Training epochs	Up to 30 (with early stopping)
Learning rate scheduler	ReduceLROnPlateau (factor=0.5, patience=2)
Early stopping	Patience = 3 (max 30 epochs)

Table 1: Model hyperparameters

3 ML Technique Implementation Details

The intrusion detection model is implemented in PyTorch under Python 3.12. The implementation includes the following components:

1. **Data preprocessing** Each `.txt` file in the ADFA-LD dataset is parsed into an ordered list of system call numbers. Each system call is then mapped to a unique integer index using a vocabulary built from the training set. To ensure compatibility with the `nn.Embedding` layer, which expects indices in the range $[0, V-1]$ where V is the vocabulary size, we maintain a mapping dictionary that assigns each system call a consecutive integer ID starting from 1. Index 0 is reserved for unknown system calls, which may appear during testing but were not seen in training. Sequences longer than 400 are truncated.

2. **Data loading and batching:** A custom PyTorch `Dataset` and `DataLoader` are used, with a `collate_fn` that applies `pad_sequence` to produce uniform-length batches for training and evaluation.

3. **Model architecture:** A custom `nn.Module` defines the model: input syscall IDs are embedded via an `nn.Embedding` layer, passed through a 2-layer unidirectional `nn.LSTM` for sequential encoding, and reduced to the final hidden state at the last time step. This vector is passed through a fully connected `nn.Linear` layer with sigmoid activation and dropout. A second `nn.Linear` layer then projects to two logits for binary classification using `nn.CrossEntropyLoss`.

4 ML Technique Training and Testing Details

The dataset was split into 64% for training, 16% for validation, and 20% for final testing. The model was trained for up to 30 epochs with a batch size of 64, minimizing cross-entropy loss using the

Adam optimizer. A learning rate scheduler (`ReduceLROnPlateau`) adjusted the learning rate based on validation loss. A custom early stopping mechanism halted training if validation loss did not improve for 3 consecutive epochs. After each epoch, validation metrics—including accuracy, precision, recall, F1-score, and ROC–AUC—were computed using `sklearn.metrics`, with predictions obtained by selecting the class with the highest logit via `argmax`. Final performance was evaluated on the held-out test set using the same metrics.

5 Results

Dataset	Training Accuracy (%)	Testing Accuracy (%)
KDD Cup (paper)	98.46	98.48
NSL-KDD (paper)	97.82	97.61
Kyoto (paper)	98.77	98.73
UNSW-NB15 (paper)	96.73	96.78
ADFA-LD (our results)	96.32	96.53

Table 2: Comparison of Training and Testing Accuracy between our model and the paper's model.

The accuracy achieved by our model on the ADFA-LD dataset is comparable to, though slightly lower than, the accuracy reported in the paper for the KDD Cup, NSL-KDD, Kyoto, and UNSW-NB15 datasets. One potential reason for this difference is the fundamental distinction between the datasets and their associated feature extraction methods: whereas the paper utilized structured and explicitly defined features provided in their network-based datasets, our evaluation relied solely on raw system-call sequences from the host-based ADFA-LD dataset. Another contributing factor may be the severe class imbalance in ADFA-LD, as there are far more normal traces than attack traces, which can make it inherently harder for the model to learn rare attack patterns without biasing toward the majority class. Finally, the limited details provided in the original paper regarding model architecture and hyperparameter choices required independent selection, which can negatively impact performance. Nevertheless, the accuracy achieved demonstrates that our model effectively learns meaningful host-based anomaly patterns directly from raw syscall sequences, even without explicit feature engineering (see Table 2).

6 mistakes analysis

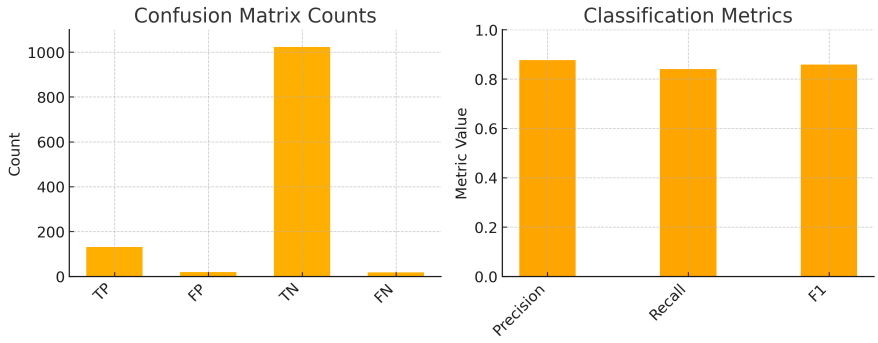

Figure 1: Confusion matrix counts and classification metrics (Precision, Recall, F1) for the test set.

At test time, our model achieved 1023 true negatives and 131 true positives but also produced 19 false positives and 18 false negatives (Figure 1). Because the ADFA-LD test set is heavily imbalanced, 1042 normal traces versus 149 attacks, the vast majority of predictions are true negatives, and the model raises very few false alarms (only about 1.7 % of normal traces are misclassified as attacks). At the same time, it misses roughly 13% of actual attacks, indicating the model is somewhat cautious about flagging sequences as attacks. This results in a precision of 87.7%, meaning that when the

model predicts an attack, it is correct most of the time. The recall of 84.0% indicates that the model successfully identifies the majority of actual attacks, though it still misses some. The F1 score of 85.8%, which balances precision and recall, reflects a strong overall trade-off between avoiding false alarms and detecting real threats. This tendency to miss more attacks than it falsely flags normal traces reflects the data imbalance: with 5205 normal traces and only 746 attacks in the full dataset, the LSTM sees far more examples of normal behavior during training and thus learns to recognize normal patterns more confidently than rare attacks. Nevertheless, these metrics demonstrate that the model effectively discriminates between normal and attack syscall sequences.

7 Improvement ideas

The detection accuracy can be improved by addressing the dataset's limitations—specifically, its small size and class imbalance. Training the model on a larger and more diverse dataset, or combining ADFA-LD with additional datasets, could expose the model to a broader range of attack patterns and improve generalization. Adding statistical features such as syscall frequency, common n-grams, or timing between calls could also provide valuable context beyond the raw sequence. Furthermore, making the LSTM bidirectional may enhance performance, as it enables the model to consider both past and future context—an advantage when suspicious activity depends on surrounding behavior. Finally, training on the full syscall sequences rather than just the first 400 steps could help the model capture deeper dependencies and more complete patterns, though this comes at the cost of increased computation.

Random forest

8 Description

Random Forest is an ensemble learning method that builds multiple decision trees using randomly sampled subsets of the data and features. Each tree votes on the classification outcome, and the majority vote is taken as the final prediction. It is robust to overfitting, interpretable, and effective for classification tasks on structured data.

9 Random forest effectivness

Random Forest is well-suited for the ADFA-LD dataset because it handles small, imbalanced, and noisy datasets effectively. While the raw syscall traces are sequential, many attack behaviors can be captured through global features like syscall frequency patterns, transition counts, and statistical summaries. Random Forest does not require time-based modeling and can still learn complex decision boundaries from these structured inputs.

Additionally, after training, Random Forest provides a measure of feature importance — an estimate of how much each input feature (e.g., a specific syscall or n-gram) contributes to the final classification. This is calculated based on how often and how effectively each feature is used to split decision nodes across all trees in the ensemble. It allows us to gain insight into which patterns or characteristics in the syscall data are most predictive of malicious behavior, making the model both effective and interpretable.

10 Features

a. To enable the Random Forest classifier to effectively detect cyber attacks from syscall traces, we convert each sequence into a fixed-length feature vector composed of multiple types of features. This hybrid approach captures both the global structure and local patterns of the syscall behavior.

   The selected features are:

   - **Syscall histogram:** A count of how many times each syscall ID appears in the trace. This captures the overall usage distribution and helps identify unusually frequent system calls that might be characteristic of specific attack patterns. For example, some exploits repeatedly use certain privileged syscalls, which this feature can highlight.
   - **3-gram frequencies:** Frequencies of consecutive triplets of syscalls (3-grams) extracted from the sequence. These features preserve some ordering information and are especially effective at identifying recurring short sequences of operations that often correspond to typical attack behaviors.
   - **Statistical features:** High-level descriptors of the sequence:
     - `Total sequence length`, indicates the complexity or duration of the process.
     - `Number of unique syscalls`, helps distinguish normal from abnormal behavior, as attack scripts often rely on a limited set of repeated syscalls, whereas legitimate programs typically exhibit greater diversity.
     - `Entropy`, Measures the unpredictability of the syscall sequence. Low entropy may signal scripted or automated activity, while high entropy might point to erratic or probing behavior, both of which could be suspicious.

   By combining these features, the model can detect a wide range of anomalies without requiring deep sequential modeling, making it well-suited for a non-temporal classifier like Random Forest. Since Random Forests make decisions by finding the best split points for each feature independently, they are not sensitive to the absolute scale of the inputs, so no additional feature scaling is needed.

b. For each syscall trace stored in a `.txt` file, the content is read and tokenized into a list of syscall IDs. A histogram is computed by counting how many times each syscall appears in the sequence. A 3-gram sliding window is then applied to extract and count all consecutive triplets of syscalls, and the top k most frequent 3-grams from the training set (with k to be determined based on experiments) are retained to limit dimensionality. In addition, we extract three statistical features from each syscall sequence. The total sequence length is simply the number of syscalls in the sequence. The number of unique syscall IDs is computed by converting the sequence into a set and counting its size. Lastly, for entropy, we calculate the relative frequency of each unique syscall ID in the sequence, and then apply the formula $-\sum p_i \log_2(p_i)$, where p_i is the relative frequency of syscall i. These statistical features are then concatenated with the histogram and 3-gram frequency features to form a single fixed-length feature vector, which is used as input for training the Random Forest classifier.

11 Implementation details

Features will be extracted from raw syscall .txt files and compiled into a fixed-size vector for each trace. The vectors will be used to train a RandomForestClassifier from scikit-learn. We will split the dataset into training, validation, and test sets (e.g., 64/16/20) and evaluate using standard classification metrics. Feature extraction and training will be performed using Python.

12 Parameters

We will use the following parameters for the Random Forest classifier:

- **n_estimators = 100**: A common default that balances performance and training time, ensuring the ensemble is large enough to reduce variance through averaging.
- **max_depth = None**: Trees are allowed to grow fully. Although deep trees may overfit individually, the ensemble nature of the Random Forest mitigates this by averaging over many such trees.
- **min_samples_split = 2**, **min_samples_leaf = 1**: These allow trees to split as much as needed. This flexibility is useful in capturing fine-grained patterns in syscall behavior, especially when combined with the regularizing effect of the ensemble.
- **max_features = 'sqrt'**: At each split, a random subset of features (square root of the total) is considered. This increases diversity among trees and reduces correlation between them, which improves generalization.
- **bootstrap = True**: Each tree is trained on a random sample of the training data drawn *with replacement* (a bootstrap sample). This sampling method introduces variation between trees, encouraging diversity in the forest and reducing the risk of overfitting.
- **class_weight = 'balanced'**: To handle class imbalance in the ADFA-LD dataset, class weights are adjusted inversely to class frequency, helping the model avoid bias toward the majority class. This is especially relevant in our case, as the ADFA-LD dataset contains significantly more normal traces than attack traces.

Non-linear SVM

8 Description

Support Vector Machine (SVM) is a supervised learning algorithm that finds the optimal hyperplane that maximizes the margin between classes. For data that is not linearly separable, SVM uses a non-linear kernel function, such as the Radial Basis Function (RBF), to map the input features into a higher-dimensional space where a clear separation becomes possible. Once trained, the hyperplane acts as a decision boundary that classifies new system-call traces based on which side of the margin they fall on.

9 Non-linear SVM effectiveness

System-call feature vectors often form non-linear clusters because normal and malicious traces can share similar syscall patterns in parts of their histograms or n-grams, yet differ in subtle ways, such as unusual frequencies, rare combinations, or atypical ordering of operations. These small but important variations mean that normal and attack traces overlap in the raw feature space, making a simple linear separation ineffective. This combination of kernel-based separation and margin maximization makes the model sensitive to localized anomalies in syscall behavior (catching rare attacks) while maintaining robustness against noise (keeping false positives low).

10 Features (same as in the Random Forest)

a. To enable the non-linear SVM classifier to effectively detect cyber attacks from syscall traces, each sequence is converted into a fixed-length feature vector that combines different types of information. This hybrid representation captures both the global structure and local patterns in syscall behavior, providing rich input for the kernel to map subtle differences into a more separable space.

   The selected features are:

   - **Syscall histogram:** A count of how many times each syscall ID appears in the trace. This captures the overall usage distribution and helps identify unusually frequent system calls that might be characteristic of specific attack patterns. For example, some exploits repeatedly use certain privileged syscalls, which this feature can highlight.
   - **3-gram frequencies:** Frequencies of consecutive triplets of syscalls (3-grams) extracted from the sequence. These features preserve some ordering information and are especially effective at identifying recurring short sequences of operations that often correspond to typical attack behaviors.
   - **Statistical features:** High-level descriptors of the sequence:
     - `Total sequence length`, indicates the complexity or duration of the process.
     - `Number of unique syscalls`, helps distinguish normal from abnormal behavior, as attack scripts often rely on a limited set of repeated syscalls, whereas legitimate programs typically exhibit greater diversity.
     - `Entropy`, Measures the unpredictability of the syscall sequence. Low entropy may signal scripted or automated activity, while high entropy might point to erratic or probing behavior, both of which could be suspicious.

   By combining these features into a single vector and scaling them appropriately, the non-linear SVM can detect a wide range of anomalies. Because the SVM relies on distance calculations in the kernel space, scaling ensures that no single feature dominates the result. The kernel function then leverages these input patterns to find subtle, non-linear boundaries that separate normal and attack traces more effectively than simple linear models.

b. The extraction of features for the non-linear SVM will follow the same process described for the Random Forest: each syscall trace is read, tokenized, and transformed into a combined feature vector containing the histogram, 3-gram frequencies, and statistical descriptors (sequence length, unique syscalls, entropy). In contrast to the Random Forest, these combined feature vectors will then be scaled using standardization (zero mean, unit variance) to ensure that all features contribute equally to the distance calculations in the RBF kernel. The scaler will be fit only on the training data and applied consistently to the validation and test sets.

11 Implementation details

Features will be extracted from the raw syscall .txt files and compiled into a fixed-size vector. The vectors will then be standardized with StandardScaler. The classifier will be trained using sklearn.svm.SVC with an RBF kernel for non-linear separation. We will split the dataset into training, validation, and test sets (e.g., 64/16/20) and tune C and gamma using grid search, which systematically tests parameter combinations to find the best validation performance. Training and evaluation will be performed using Python.

12 Parameters

We will use the following parameters for the Non-linear SVM:

- **kernel = 'rbf'**: The Radial Basis Function kernel maps the input features into a higher-dimensional space to handle non-linear separation, which is well suited for syscall feature vectors with subtle differences.
- **C = ?**: The regularization parameter controls the trade-off between maximizing the margin and allowing some misclassifications. A well-chosen C helps balance overfitting and underfitting; this will be tuned using grid search.
- **gamma = ?**: The kernel coefficient determines how far the influence of each training sample reaches. Smaller values mean smoother boundaries; larger values create tighter clusters. This will also be tuned using grid search.
- **class_weight = 'balanced'**: To address class imbalance in the ADFA-LD dataset, class weights are adjusted inversely to class frequency, ensuring that rare attack traces receive appropriate importance during training.
- **probability = False**: By default, probability estimates are not required; only the decision function is used for classification. This keeps training efficient.