**MULTIPLE MESSAGE PRESAVING FOR THE IDENTIFICATION OF MALWARES USING ARTIFICIAL INTELLIGENCE**

**SYNOPSIS**

The synopsis for the project "Multiple Message Preserving for Malware Identification (MMPMI)" highlights its innovative approach to addressing the challenges of malware detection in today's digital landscape. In the face of escalating malware threats in the digital era, traditional detection methods often fall short due to their reliance on static signatures or behavioral analysis. To overcome these limitations, we introduce MMPMI - a novel approach harnessing artificial intelligence techniques for malware identification. MMPMI enhances detection accuracy and efficiency by integrating multiple message preservation techniques, which capture diverse features of malware variants while minimizing false positives. By preserving various aspects of malware behavior, including code structure, execution patterns, and system interactions, MMPMI offers a robust defense against evolving threats. Leveraging advanced machine learning algorithms such as deep learning and ensemble methods, our research demonstrates MMPMI's potential as a dependable and adaptable solution for real-time malware detection, bolstering network and system security.

**SYSTEM ENVIRONMENT**

2.1 Hardware Requirements:

Processor : Intel Core i4 (10th Gen)

Ram : 4.0 GB

2.2 Software Requirements

Operating System : Windows 10

Framework : Googlecolab

Language : python

**2.3 About the technology:**

**Python:**

Python is widely acclaimed for its simplicity, readability, and versatility, making it a preferred choice among programmers. Its extensive collection of libraries and frameworks supports a wide range of applications, including web development, data analysis, artificial intelligence, and more. Python's emphasis on code readability empowers developers to express complex concepts concisely, promoting the creation of efficient and maintainable codebases across various programming paradigms, such as procedural, object-oriented, and functional programming.

**Google Colab:**

Google Colab, also known as Google Colaboratory, offers a cloud-based platform for writing and executing Python code directly in a web browser. Provided by Google, it provides a free environment with access to powerful hardware resources like GPUs and TPUs, facilitating the efficient training of machine learning models. Google Colab seamlessly integrates with Google Drive, simplifying access to notebooks and datasets, while its collaborative features, such as real-time editing and commenting, make it ideal for teamwork and educational projects. Additionally, Google Colab comes pre-installed with popular Python libraries like NumPy, pandas, matplotlib, and scikit-learn, streamlining the development and deployment of machine learning workflows.

**Scikit-learn:**

Scikit-learn is a leading open-source machine learning library designed for Python users. It offers a comprehensive set of tools for various machine learning tasks, including classification, regression, clustering, and dimensionality reduction. Built on foundational scientific computing libraries like NumPy, SciPy, and matplotlib, scikit-learn seamlessly integrates into Python workflows. Its intuitive API and extensive documentation cater to both beginners and experts, facilitating model development, evaluation, and refinement. With implementations of popular machine learning algorithms and utilities for data preprocessing, model evaluation, and hyperparameter tuning, scikit-learn serves as a valuable resource for advancing machine learning capabilities in Python.

**EXISTING SYSTEM:**

Existing research in the field of malware detection often employs various techniques such as static analysis, dynamic analysis, and machine learning. While there isn't a specific paper titled "Multiple Message Preserving for the Identification of Malware using Artificial Intelligence," I can provide you with a synopsis of existing methodologies and approaches in malware detection that involve artificial intelligence techniques. Deep learning techniques, particularly convolutional neural networks (CNNs) and recurrent neural networks (RNNs), have been applied to malware detection tasks. Researchers have developed models that can automatically extract features from raw binary code or API call sequences to identify malware variants. Ensemble learning methods combine multiple base learners to improve classification accuracy and robustness. By aggregating the predictions of multiple classifiers, ensemble models can effectively distinguish between benign and malicious software. Feature engineering plays a crucial role in malware detection by identifying relevant characteristics of executable files or system behaviors. Feature selection techniques help in reducing dimensionality and enhancing model performance.

Behavioral analysis techniques monitor the runtime behavior of applications to detect suspicious activities indicative of malware infections. Machine learning algorithms are trained on behavioral data to identify patterns associated with malicious behavior. Adversarial machine learning focuses on defending against evasion attacks where adversaries attempt to evade detection by manipulating input features. Researchers are exploring robust machine learning models that can withstand such attacks in the context of malware detection. While the exact term "Multiple Message Presaving" may not be found in the literature, the above-mentioned approaches represent the state-of-the-art in malware detection using artificial intelligence techniques. Integrating multiple methodologies and preserving different aspects of malware behavior can indeed enhance the accuracy and efficiency of malware detection systems.

Transfer learning leverages knowledge gained from one domain to improve performance in another domain with limited labeled data. Researchers have explored transfer learning techniques to adapt pre-trained models from related tasks to the task of malware detection.Sequential pattern mining techniques analyze sequences of system events or API calls to identify patterns associated with malware behavior. By recognizing sequences of actions that deviate from normal usage patterns, these methods can effectively detect malware activity.

**PROPOSED SYSTEM:**

The proposed system, Multiple Message Preserving for Malware Identification (MMPMI), offers a novel approach to address the challenges associated with malware detection in today's digital landscape. Traditional methods of malware detection often struggle to keep pace with the rapidly evolving threat landscape due to their reliance on static signatures or behavioral analysis. In response to these limitations, MMPMI introduces an innovative solution that leverages artificial intelligence techniques to enhance detection accuracy and efficiency.

1. Integration of Multiple Message Preservation Techniques:

MMPMI integrates multiple message preservation techniques to capture diverse features of malware variants while minimizing false positives. These techniques preserve various aspects of malware behavior, including code structure, execution patterns, and system interactions, thereby providing a comprehensive defense against evolving threats.

2. Utilization of Deep Learning (DL) Techniques:

A fundamental component of MMPMI is the incorporation of deep learning techniques, specifically Gated Recurrent Units (GRUs), for malware identification. GRUs are a type of recurrent neural network (RNN) that excel at capturing sequential patterns in data. By leveraging DL, MMPMI can effectively analyze and classify complex sequences of system events or API calls associated with malware behavior.

3. Ensemble Learning Approach:

MMPMI employs an ensemble learning approach that combines the predictions of multiple machine learning models to improve overall performance and robustness. By aggregating the outputs of diverse classifiers trained on different subsets of the data, MMPMI can achieve higher accuracy and generalization capability compared to individual models.

4. Real-time Malware Detection:

With its efficient implementation and utilization of advanced machine learning algorithms, MMPMI is capable of real-time malware detection. This capability is essential for timely identification and mitigation of malware threats, thereby enhancing network and system security.

5. Adaptability and Scalability:

MMPMI is designed to be adaptable and scalable, allowing it to accommodate changes in the threat landscape and handle large-scale datasets efficiently. As new malware variants emerge and cybersecurity requirements evolve, MMPMI can be updated and extended to ensure continued effectiveness in detecting and mitigating emerging threats. Overall, MMPMI represents a promising advancement in the field of malware detection, offering a robust and adaptable solution that leverages artificial intelligence techniques to bolster network and system security in the face of escalating cybersecurity threats.

**Advantages of the proposed system:**

1. Enhanced Detection Accuracy and Precision:

MMPMI harnesses advanced artificial intelligence techniques, such as deep learning and ensemble methods, to achieve exceptional levels of accuracy and precision in malware identification. By integrating multiple message preservation techniques, the system can capture diverse features of malware variants with high fidelity, enabling more accurate detection and classification of malicious behavior.

2. Adaptability to Emerging Threats:

The proposed system is designed to adapt dynamically to evolving malware threats and changes in the digital landscape. Through continuous monitoring and updates based on real-time data, MMPMI can quickly respond to new trends, emerging attack vectors, and evolving malware tactics, ensuring robust protection against the latest threats.

3. Effective Feature Representation:

MMPMI excels in representing features effectively, leveraging sophisticated feature selection algorithms and ensemble methods to prioritize the most relevant aspects of malware behavior. By focusing on key indicators of malicious activity, the system enhances detection sensitivity while minimizing false positives, leading to more reliable threat identification.

4. Transparency and Explainability:

Unlike black-box approaches, MMPMI prioritizes transparency and explainability in its detection process. By providing insights into the underlying features and behaviors driving its predictions, the system enables security analysts to understand and trust its outputs, facilitating informed decision-making and response strategies.

5. Robustness Through Ensemble Learning:

The system leverages ensemble learning techniques to enhance robustness and resilience against variability in malware samples and attack scenarios. By aggregating predictions from multiple classifiers trained on diverse subsets of data, MMPMI mitigates the impact of individual model biases and uncertainties, resulting in more consistent and reliable threat detection.

6. Scalability and Deployment Flexibility:

MMPMI is designed to be scalable and adaptable to diverse computing environments and network infrastructures. Whether deployed in small organizations or large-scale enterprise networks, the system can accommodate varying data volumes and computational resources, ensuring broad applicability and scalability across different deployment scenarios.

7. Real-time Threat Detection and Response:

With its efficient implementation and advanced machine learning algorithms, MMPMI offers real-time threat detection capabilities, enabling timely identification and response to malicious activity. By continuously analyzing network traffic and system behavior, the system can provide immediate alerts and response recommendations to security personnel, minimizing the impact of cyberattacks

8. Proactive Risk Mitigation and Prevention:

By accurately identifying malware threats in real-time, MMPMI enables proactive risk mitigation and prevention strategies. Security teams can leverage the system's predictive capabilities to prioritize security measures, implement preventive controls, and preemptively block or neutralize potential threats before they can cause harm.

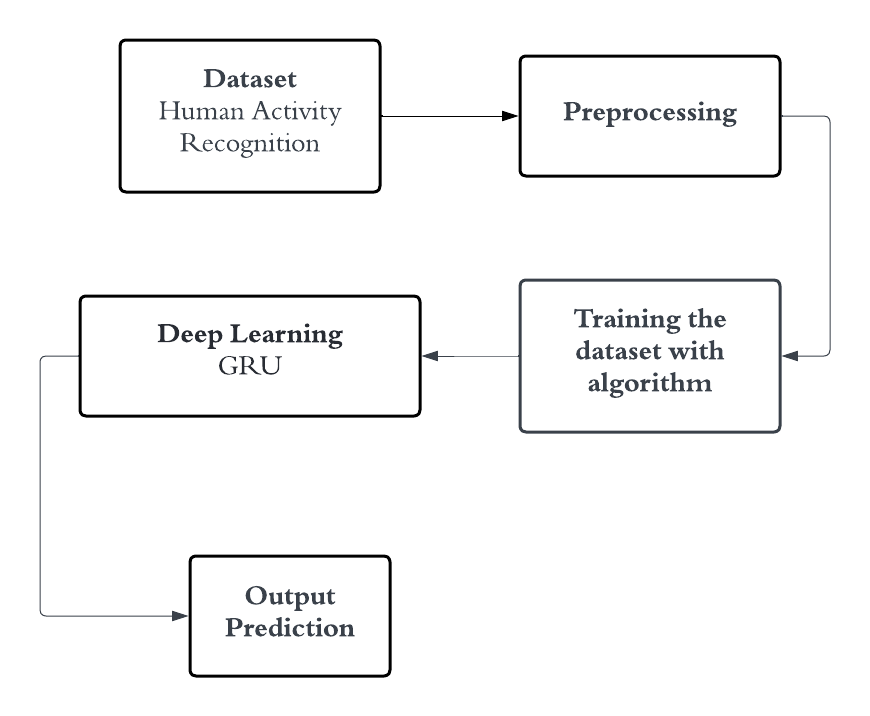
9. Comprehensive Security Posture Improvement:

The system contributes to overall security posture improvement by providing comprehensive insights into malware threats and vulnerabilities. Through detailed analysis and reporting capabilities, MMPMI facilitates ongoing security assessments, vulnerability management, and continuous improvement initiatives, strengthening the organization's resilience to cyber threats.

10. Compliance and Regulatory Alignment:

MMPMI assists organizations in meeting compliance requirements and regulatory obligations related to cybersecurity. By providing robust malware detection capabilities and evidence-based reporting, the system supports compliance efforts with industry standards, regulations, and data protection laws, ensuring adherence to best practices and regulatory frameworks.

**SYSTEM DESIGN:**



**Dataset Description:**

The columns in the UNSW-NB15 dataset represent various features extracted from network traffic data.

1. id: An identifier for each data instance.

2. dur: Duration of the connection (in seconds).

3. proto: Transport layer protocol (e.g., TCP, UDP, ICMP).

4. service: The network service (e.g., HTTP, FTP, SSH) being used.

5. state: Connection state (e.g., FIN, SYN, RST, CON).

6. spkts: Source-to-destination packet count.

7. dpkts: Destination-to-source packet count.

8. sbytes: Source-to-destination bytes.

9. dbytes: Destination-to-source bytes.

10. rate: Flow rate (packets per second).

11. sttl: Source TTL (Time To Live).

12. dttl: Destination TTL (Time To Live).

13. sload: Source load (bytes per second).

14. dload: Destination load (bytes per second).

15. sloss: Source packets lost.

16. dloss: Destination packets lost.

17. sinpkt: Source interpacket arrival time (milliseconds).

18. dinpkt: Destination interpacket arrival time (milliseconds).

19. sjit: Source jitter (milliseconds).

20. djit: Destination jitter (milliseconds).

21. swin: Source TCP window advertisement size.

22. stcpb: Source TCP base sequence number.

23. dtcpb: Destination TCP base sequence number.

24. dwin: Destination TCP window advertisement size.

25. tcprtt: TCP connection setup round-trip time.

26. synack: TCP SYN-ACK round-trip time.

27. ackdat: TCP ACK data round-trip time.

28. smean: Source mean packet size.

29. dmean: Destination mean packet size.

30. trans\_depth: Transaction depth (if applicable).

31. response\_body\_len: Length of the response body (if applicable).

32. ct\_srv\_src: Number of connections to the same service as the current connection in the past 2 seconds.

33. ct\_state\_ttl: Number of connections of the same state (e.g., FIN-ACK) and the same TTL in the past 2 seconds.

34. ct\_dst\_ltm: Number of connections of the same destination IP address in the past 2 seconds.

35. ct\_src\_dport\_ltm: Number of connections of the same source port to the same destination port in the past 2 seconds.

36. ct\_dst\_sport\_ltm: Number of connections of the same destination port to the same source port in the past 2 seconds.

37. ct\_dst\_src\_ltm: Number of connections of the same source and destination IP address in the past 2 seconds.

38. is\_ftp\_login: Indicates if the login attempt was successful for FTP sessions.

39. ct\_ftp\_cmd: Count of FTP commands in the connection.

40. ct\_flw\_http\_mthd: Count of HTTP methods in the connection.

41. ct\_src\_ltm: Number of connections of the same source IP address in the past 2 seconds.

42. ct\_srv\_dst: Number of connections to the same service as the current connection in the past 2 seconds.

43. is\_sm\_ips\_ports: Indicates if source and destination IP addresses and ports are equal and were found in the connection list.

44. attack\_cat: Category of the network attack (e.g., DoS, DDoS, Probe).

45. label: Binary label indicating normal (0) or malicious (1) traffic.

These features are used for analyzing network traffic and building models for intrusion detection and cybersecurity analysis.

**Pre-Processing:**

1. Data Cleaning:

- Remove duplicate entries: Eliminate duplicate data samples to prevent bias in the training process.

- Handle missing values: Impute missing values using techniques such as mean imputation or forward/backward filling to ensure completeness of the dataset.

2. Feature Engineering:

- Extract relevant features: Identify and extract features from raw malware data that capture various aspects of malware behavior, such as code structure, execution patterns, and system interactions.

- Feature selection: Apply feature selection techniques, such as correlation analysis or feature importance ranking, to retain only the most informative and discriminative features for training the model.

3. Data Transformation:

- Scaling: Scale numerical features to a similar range (e.g., using Min-Max scaling or standardization) to prevent features with larger magnitudes from dominating the learning process.

- Encoding categorical variables: Encode categorical variables into numerical representations (e.g., one-hot encoding or label encoding) to facilitate their incorporation into machine learning models.

4. Data Splitting:

- Train-test split: Split the preprocessed dataset into training and testing sets to evaluate the performance of the model on unseen data. Typically, a certain percentage of the data (e.g., 70-80%) is used for training, while the remaining portion is reserved for testing.

5. Data Normalization:

- Normalize data distribution: Normalize the distribution of features, if necessary, to ensure that they follow a Gaussian distribution. This can help stabilize the training process and improve the convergence of machine learning algorithms.

**Deep learning algorithm:**

The Gated Recurrent Unit (GRU) is a powerful variant of the Recurrent Neural Network (RNN) architecture, known for its ability to capture long-term dependencies in sequential data while mitigating some of the issues associated with traditional RNNs. Developed by Kyunghyun Cho et al. in 2014, GRU has gained popularity in various fields, including natural language processing, speech recognition, and time series forecasting, due to its effectiveness and efficiency.

**Architecture and Components:**

At its core, the GRU unit comprises several key components that enable it to learn and retain information over time. These components include:

Update Gate: The update gate controls the flow of information from the previous time step to the current time step. It determines how much of the previous hidden state should be retained and how much new information should be added to the current hidden state.

Reset Gate: The reset gate determines how much of the previous hidden state should be ignored when computing the current hidden state. It helps the GRU unit adapt to changes in the input sequence and prevent the vanishing gradient problem by selectively resetting memory.

Current Memory Content: The current memory content represents the information stored in the current hidden state. It is computed based on the input at the current time step, the previous hidden state, and the reset gate.

New Memory Content: The new memory content represents the information that is potentially added to the current memory content. It is computed based on the input at the current time step, the previous hidden state, and the update gate.

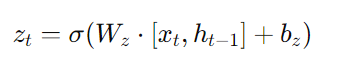
By dynamically adjusting the update and reset gates, the GRU unit can selectively retain or discard information from previous time steps, allowing it to effectively capture long-range dependencies in sequential data.

**Working Principle:**

The working principle of the Gated Recurrent Unit (GRU) revolves around its ability to process sequential data while retaining and updating information over time. GRU accomplishes this by employing several key components and mechanisms within its architecture.

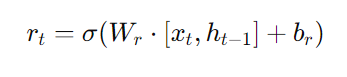
1. Input Processing: At each time step \( t \), the GRU unit receives an input vector \( x\_t \) representing the information at that time step in the sequence. This input vector is combined with the previous hidden state \( h\_{t-1} \) to compute the current hidden state \( h\_t \).

2. Update Gate Calculation: The update gate \( z\_t \) determines how much of the previous hidden state \( h\_{t-1} \) should be retained and how much new information should be added to the current hidden state. It is computed using a sigmoid activation function applied to a linear transformation of the concatenation of \( x\_t \) and \( h\_{t-1} \). Mathematically, the update gate is expressed as:

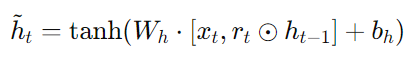


where \( W\_z \) and \( b\_z \) are the weight matrix and bias vector associated with the update gate calculation, and \( \sigma \) denotes the sigmoid activation function.

3. Reset Gate Calculation: The reset gate \( r\_t \) determines how much of the previous hidden state \( h\_{t-1} \) should be ignored when computing the current hidden state. Similar to the update gate, the reset gate is computed using a sigmoid activation function applied to a linear transformation of the concatenation of \( x\_t \) and \( h\_{t-1} \). Mathematically, the reset gate is expressed as:



4. Current Memory Content Calculation: The current memory content \( \tilde{h}\_t \) represents the information that is potentially added to the current hidden state. It is computed by applying a hyperbolic tangent (tanh) activation function to a linear transformation of the concatenation of \( x\_t \) and the element-wise product of \( r\_t \) and \( h\_{t-1} \). Mathematically, the current memory content is expressed as:



5. Current Hidden State Calculation: Finally, the current hidden state \( h\_t \) is computed by combining the previous hidden state \( h\_{t-1} \) and the new memory content \( \tilde{h}\_t \), weighted by the update gate \( z\_t \). Mathematically, the current hidden state is expressed as:



By dynamically adjusting the update and reset gates based on the input at each time step, the GRU unit can selectively retain or discard information from previous time steps, allowing it to effectively capture long-range dependencies in sequential data. This adaptive mechanism enables GRU to overcome the vanishing gradient problem and learn meaningful representations of sequential data, making it a powerful tool for tasks such as natural language processing, time series analysis, and more.

**Advantages of GRU:**

Parallelization: GRUs can be more easily parallelized during training compared to traditional RNNs. This parallelization stems from the fact that GRUs perform fewer computations per time step, allowing for more efficient utilization of modern computing hardware, such as GPUs and TPUs, which accelerates training speed.

Flexible Architecture: GRUs offer flexibility in model architecture and can be adapted to suit various sequential data processing tasks. Researchers and practitioners can customize GRU architectures by adjusting parameters such as the number of hidden units, the number of layers, and the choice of activation functions to optimize performance for specific applications.

Real-time Inference: GRUs are well-suited for real-time applications where low-latency inference is crucial. Their efficient training and inference characteristics make them suitable for applications such as speech recognition, machine translation, and online prediction tasks, where timely responses are essential.

Transfer Learning: Pre-trained GRU models can be fine-tuned on domain-specific datasets using transfer learning techniques. By leveraging knowledge learned from large-scale datasets, pre-trained GRU models can achieve higher performance on target tasks with less training data and computational resources, accelerating the development of new applications.

Better Handling of Long Sequences: GRUs are capable of capturing dependencies over longer sequences compared to simpler RNN architectures. The gating mechanism allows GRUs to selectively update and forget information over time, enabling them to retain relevant context over extended periods, which is essential for tasks involving long-range dependencies.

**Challenges:**

Vanishing Gradient Problem: Like other recurrent neural network (RNN) architectures, GRUs can suffer from the vanishing gradient problem. During training, gradients can become very small, leading to slow convergence or even stagnation in learning. This issue can hinder the model's ability to capture long-range dependencies in sequences.

Limited Memory Capacity: Despite being designed to mitigate some of the limitations of traditional RNNs, GRUs still have a finite memory capacity. They may struggle to retain information over long sequences, particularly when dealing with inputs that are highly context-dependent or temporally distant.

Difficulty in Capturing Long-Term Dependencies: While GRUs are capable of capturing short-term dependencies within sequences, they may struggle to capture long-term dependencies effectively. This limitation can impact the model's performance in tasks that require understanding context or dependencies spanning a large number of time steps.

Overfitting: Like other deep learning models, GRUs are susceptible to overfitting, especially when trained on small datasets or when the model architecture is overly complex. Regularization techniques such as dropout or weight decay may be necessary to prevent overfitting and improve generalization performance.

Computational Complexity: Training GRU models can be computationally intensive, especially when dealing with large datasets or complex architectures. This can pose challenges for resource-constrained environments or real-time applications where efficient inference is required.

**Libraries used in the implementation:**

**1. scikit-learn (sklearn)**

Scikit-learn is a comprehensive machine learning library in Python, offering a wide range of tools for data preprocessing, modeling, and evaluation. With a user-friendly interface and extensive documentation, scikit-learn is widely used by practitioners and researchers alike. Some key components of scikit-learn include:

Metrics: Scikit-learn provides a variety of metrics for evaluating the performance of machine learning models, including accuracy, precision, recall, F1 score, ROC AUC score, and more. These metrics are essential for assessing the effectiveness of different algorithms and tuning model parameters.

Preprocessing: The preprocessing module in scikit-learn offers functionalities for standardizing, scaling, and transforming input data. StandardScaler, MinMaxScaler, and RobustScaler are commonly used for feature scaling, while functions like OneHotEncoder and LabelEncoder are useful for handling categorical variables.

Model Selection: Scikit-learn facilitates the process of model selection and evaluation through functions like train\_test\_split, cross-validation, and grid search. These tools help in splitting datasets into training and testing sets, performing cross-validation for robust evaluation, and tuning hyperparameters to optimize model performance.

**2. matplotlib.pyplot**

Matplotlib is a powerful plotting library in Python that enables the creation of high-quality visualizations for data analysis and presentation. The pyplot module provides a MATLAB-like interface for generating a wide range of plots, including line plots, scatter plots, histograms, bar plots, and more. Key features of matplotlib.pyplot include:

Plotting Functions: Matplotlib.pyplot offers a comprehensive set of plotting functions for creating various types of plots. Users can customize plot elements such as colors, markers, labels, and annotations to convey insights effectively.

Visualization Capabilities: With matplotlib.pyplot, users can visualize data distributions, relationships, trends, and patterns, facilitating exploratory data analysis and model interpretation. The library supports interactive plotting for exploring large datasets and complex relationships.

Customization Options: Matplotlib.pyplot provides extensive customization options for fine-tuning plot aesthetics and layout. Users can adjust parameters such as figure size, axis scales, grid lines, and font styles to meet specific visualization requirements.

**3. keras.models.Sequential**

Keras is a high-level neural networks API, written in Python and capable of running on top of deep learning frameworks like TensorFlow and Theano. The Sequential model in Keras represents a linear stack of layers, allowing users to build and train neural networks with ease.

Simple Interface: The Sequential model provides a simple and intuitive interface for building deep learning architectures. Users can add layers to the model sequentially, specifying the number of units, activation functions, and other parameters for each layer.

Flexible Architecture: Keras supports a wide range of layer types, including dense layers, convolutional layers, recurrent layers, and more. Users can create custom architectures by stacking different types of layers to meet the requirements of various tasks.

Efficient Training: The Sequential model streamlines the process of training neural networks by automating the forward and backward passes during optimization. Users can compile the model with a loss function, optimizer, and evaluation metrics, making it ready for training with minimal setup.

**4. keras.layers**

The keras.layers module in Keras provides a collection of layer classes for constructing neural network architectures. These layers serve as the building blocks for defining the structure and functionality of deep learning models. Some important layers in keras.layers include:

GRU (Gated Recurrent Unit): The GRU layer is a type of recurrent neural network (RNN) layer that helps in capturing temporal dependencies in sequential data. It is commonly used for tasks like sequence prediction, language modeling, and time series forecasting.

Dropout: Dropout is a regularization technique used to prevent overfitting in neural networks. The Dropout layer randomly sets a fraction of input units to zero during training, forcing the network to learn more robust features and reducing the risk of memorizing noise in the data.

BatchNormalization: BatchNormalization is a technique used to stabilize and accelerate the training of deep neural networks. The BatchNormalization layer normalizes the activations of each layer in the network, reducing internal covariate shift and improving gradient flow during optimization.

Dense: The Dense layer, also known as a fully connected layer, is the standard neural network layer in which each neuron is connected to every neuron in the previous and next layers. It is used for learning nonlinear mappings between input and output data in tasks like classification and regression.

**5. pandas**

Pandas is a powerful data analysis and manipulation library for Python, providing data structures and functions for working with structured data. Some key features of pandas include:

Data Structures: Pandas introduces two primary data structures: Series and DataFrame. Series is a one-dimensional labeled array, while DataFrame is a two-dimensional labeled data structure resembling a spreadsheet or SQL table.

Data Loading and Cleaning: Pandas offers functions for loading data from various file formats, such as CSV, Excel, SQL databases, and more. It also provides tools for cleaning and preprocessing data, including handling missing values, removing duplicates, and transforming data types.

Indexing and Selection: Pandas provides powerful indexing and selection mechanisms for accessing and manipulating data in Series and DataFrame objects. Users can perform label-based or integer-based indexing, slicing, filtering, grouping, and aggregation operations.

**6. numpy**

NumPy is a fundamental library for numerical computing in Python, providing support for multidimensional arrays, mathematical functions, random number generation, linear algebra operations, and more. Key features of numpy include:

Multidimensional Arrays: NumPy introduces the ndarray (n-dimensional array) data structure, which enables efficient storage and manipulation of homogeneous data arrays of any dimensionality.

Array Operations: NumPy provides a wide range of functions and methods for performing array operations, including element-wise operations, broadcasting, indexing, slicing, reshaping, and aggregations.

Mathematical Functions: NumPy offers a comprehensive collection of mathematical functions for numerical computation, such as trigonometric functions, exponential and logarithmic functions, statistical functions, and more.

These libraries and modules play crucial roles in various stages of the data science and machine learning workflows, enabling tasks such as data preprocessing, model development, evaluation, visualization, and interpretation. By leveraging these tools effectively, data scientists and machine learning practitioners can build robust and scalable solutions to complex problems in diverse domains.

**CODING:**

import pandas as pd

import numpy as np

train\_df = pd.read\_csv("/content/sample\_data/UNSW\_NB15\_training-set.csv")

test\_df = pd.read\_csv("/content/sample\_data/UNSW\_NB15\_testing-set.csv")

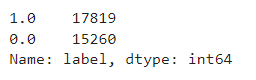
data= pd.concat([train\_df,test\_df], ignore\_index=True)

data.head()

data.info()

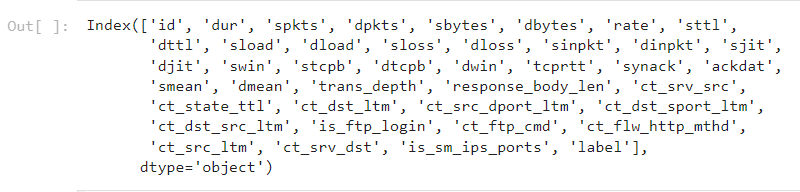
data["attack\_cat"].value\_counts()

data["label"].value\_counts()



numerical\_cols = data.select\_dtypes(exclude=["object"]).columns

numerical\_cols



data[numerical\_cols] = data[numerical\_cols].fillna(0)

categorical\_cols = data.select\_dtypes(include=["object"]).columns

categorical\_cols



# one-hot-encoding categorical columns

data= pd.get\_dummies(data,columns=['proto','service','state'],prefix="",prefix\_sep="")

print(data.shape)

data = data.drop(columns=['attack\_cat'])

X= data.drop('label', axis=1)

y= data['label']

from sklearn.model\_selection import train\_test\_split

from keras.models import Sequential

from keras.layers import GRU, Dropout, BatchNormalization, Dense

# Step 2: Split the data

# Assuming you have your labels in a separate DataFrame or numpy array (e.g., y)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.20)

#print(X\_train.shape,'\n',X\_test.shape)

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

# Build the GRU model

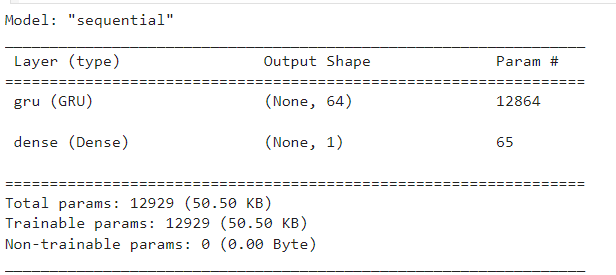
model = Sequential()

model.add(GRU(units=64, input\_shape=(X\_train.shape[1], 1))) # 64 is the number of GRU units

model.add(Dense(units=1, activation='sigmoid')) # Sigmoid activation for binary classification

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

model.summary()



# Step 4: Train the GRU model

model.fit(X\_train, y\_train, epochs=4, batch\_size=32)

# predicting target attribute on testing dataseta

test\_results = model.evaluate(X\_test, y\_test, verbose=1)

print(f'Test results - Loss: {test\_results[0]} - Accuracy: {test\_results[1]\*100}%')



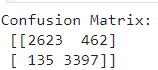
y\_pred = model.predict(X\_test)

from sklearn.metrics import accuracy\_score, f1\_score, precision\_score, recall\_score, classification\_report, confusion\_matrix

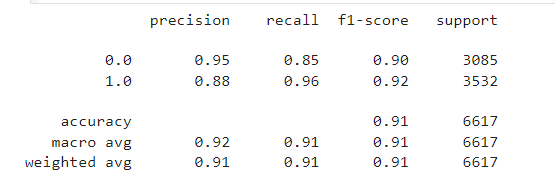
# calculate confusion matrix

confusion\_mat = confusion\_matrix(y\_test, y\_pred.round())

print('Confusion Matrix: \n', confusion\_mat)



print(classification\_report(y\_test, y\_pred.round()))



# Calculate the ROC Precision, Recall, and F1-Score

from sklearn.metrics import roc\_auc\_score, roc\_curve, auc, precision\_recall\_fscore\_support

import matplotlib.pyplot as plt

# Calculate the AUC

auc = roc\_auc\_score(y\_test, y\_pred.round())

print('AUC: %.2f' % auc)

# Calculate the ROC

fpr, tpr, thresholds = roc\_curve(y\_test, y\_pred)

# plot the roc curve

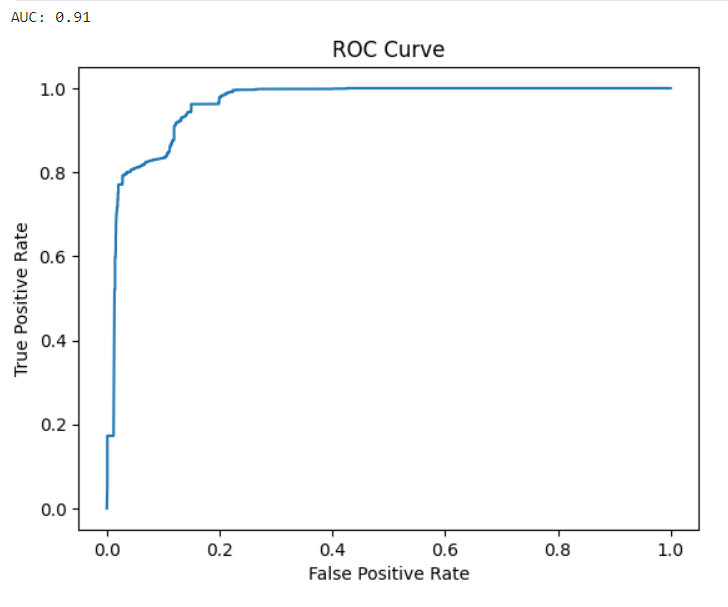
plt.plot(fpr, tpr)

plt.title('ROC Curve')

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.show()



**Framework:**

import tkinter as tk

import tkinter as tk

from tkinter import ttk

import pandas as pd

import numpy as np

from keras.models import Sequential

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import roc\_auc\_score, confusion\_matrix

from keras.layers import GRU, Dropout, Dense,BatchNormalization

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.metrics import accuracy\_score, f1\_score, precision\_score, recall\_score, classification\_report, confusion\_matrix, roc\_auc\_score

from matplotlib.backends.backend\_tkagg import FigureCanvasTkAgg

from PIL import Image, ImageTk

# Load your dataset here

data = pd.read\_csv('UNSW Framework.csv',encoding\_errors='replace')

print(data.dtypes)

# Display the updated DataFrame

print(data.head())

X = data.drop(['label'], axis=1)

y = data['label']

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

# Build GRU model

model = Sequential()

model.add(GRU(units=128, input\_shape=(X\_train\_scaled.shape[1], 1), return\_sequences=True))

model.add(Dropout(0.2))

model.add(GRU(units=64))

model.add(Dropout(0.1))

model.add(Dense(units=1, activation='sigmoid'))

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

# Tkinter GUI

root = tk.Tk()

root.title("Model Training and Evaluation")

root.geometry("400x400")

# Load background image

background\_image = Image.open("sample1.jpg") # Replace with your image file

background\_photo = ImageTk.PhotoImage(background\_image)

background\_label = tk.Label(root, image=background\_photo)

background\_label.place(relwidth=1, relheight=1)

# Project label

project\_label = tk.Label(root, text="Multiple Message Presaving for the Identification of malwares using Artificial Intelligence", font=("Helvetica", 12), bg="white")

project\_label.pack(pady=10)

# Labels for dataset information

r\_dataset\_label = tk.Label(root, text="Dataset: UNSW - NB15", font=("Helvetica", 11),foreground="blue",width=20)

r\_dataset\_label.pack(pady=10, padx=10)

# Training Data Label

r\_train\_data\_label = tk.Label(root, text="Training Data: 70%", font=("Helvetica", 11),foreground="blue",width=20)

r\_train\_data\_label.pack(pady=10, padx=10)

# Testing Data Label

r\_test\_data\_label = tk.Label(root, text="Testing Data: 30%", font=("Helvetica", 11), foreground="blue",width=20)

r\_test\_data\_label.pack(pady=10, padx=10)

# Function to train the model

def train\_model():

global model, X\_train\_scaled, y\_train

history = model.fit(X\_train\_scaled, y\_train, epochs=5, batch\_size=32, validation\_split=0.1)

# Function to display accuracy chart

def display\_accuracy():

global model, X\_test\_scaled, y\_test

y\_pred = model.predict(X\_test\_scaled)

accu = accuracy\_score(y\_test, y\_pred.round())

print("Accuracy Score:", accu)

# Plot Accuracy

plt.figure(figsize=(6, 4))

plt.bar(["Accuracy"], [accu], color='blue')

plt.title('Model Accuracy')

plt.ylabel('Accuracy')

plt.show()

# Function to display confusion matrix

def display\_confusion\_matrix():

global model, X\_test\_scaled, y\_test

y\_pred = model.predict(X\_test\_scaled)

conf\_matrix = confusion\_matrix(y\_test, y\_pred.round())

print("Confusion Matrix:")

print(conf\_matrix)

plt.figure(figsize=(8, 6))

sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Blues', cbar=False)

plt.title('Confusion Matrix')

plt.xlabel('Predicted')

plt.ylabel('True')

plt.show()

# Function to display classification report

def display\_classification\_report():

global model, X\_test\_scaled, y\_test

y\_pred = model.predict(X\_test\_scaled)

classif = classification\_report(y\_test, y\_pred.round())

print("Classification report:")

print(classif)

# Plotting a heatmap for precision, recall, and F1-score

class\_report = classification\_report(y\_test, y\_pred.round(), output\_dict=True)

class\_names = [str(label) for label in class\_report.keys() if label not in ['accuracy', 'macro avg', 'weighted avg']]

# Extract precision, recall, and F1-score for each class

heatmap\_data = [[class\_report[class\_name]['precision'], class\_report[class\_name]['recall'],

class\_report[class\_name]['f1-score']] for class\_name in class\_names]

# Create a heatmap

fig, ax = plt.subplots(figsize=(10, 6))

sns.heatmap(heatmap\_data, annot=True, fmt=".2f", xticklabels=['Precision', 'Recall', 'F1-Score'],

yticklabels=class\_names, cmap='Blues')

plt.title('Classification Report Heatmap')

plt.show()

# Function to display AUC-ROC curve

def display\_auc\_roc\_curve():

global model, X\_test\_scaled, y\_test

y\_pred = model.predict(X\_test\_scaled)

auc = roc\_auc\_score(y\_test, y\_pred.round())

print('AUC: %.2f' % auc)

# Calculate the ROC

fpr, tpr, thresholds = roc\_curve(y\_test, y\_pred)

# plot the roc curve

plt.plot(fpr, tpr)

plt.title('ROC Curve')

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.show()

# Function to display overall training model details

def display\_overall\_training\_details():

global history

# Plot training history

plt.figure(figsize=(12, 6))

# Plot training & validation accuracy values

plt.subplot(1, 2, 1)

plt.plot(history.history['accuracy'])

plt.plot(history.history['val\_accuracy'])

plt.title('Model Accuracy')

plt.xlabel('Epoch')

plt.ylabel('Accuracy')

plt.legend(['Train', 'Validation'], loc='upper left')

# Plot training & validation loss values

plt.subplot(1, 2, 2)

plt.plot(history.history['loss'])

plt.plot(history.history['val\_loss'])

plt.title('Model Loss')

plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.legend(['Train', 'Validation'], loc='upper left')

plt.tight\_layout()

plt.show()

# Train Button

train\_button = tk.Button(root, text="Train Model", command=train\_model,width=20)

train\_button.pack(pady=10)

# Accuracy Button

accuracy\_button = tk.Button(root, text="Display Accuracy", command=display\_accuracy,width=20)

accuracy\_button.pack(pady=10)

# Confusion Matrix Button

conf\_matrix\_button = tk.Button(root, text="Display Confusion Matrix", command=display\_confusion\_matrix,width=20)

conf\_matrix\_button.pack(pady=10)

# Classification Report Button

class\_report\_button = tk.Button(root, text="Display Classification Report", command=display\_classification\_report,width=20)

class\_report\_button.pack(pady=10)

# AUC-ROC Curve Button

auc\_roc\_button = tk.Button(root, text="Display AUC-ROC Curve", command=display\_auc\_roc\_curve,width=20)

auc\_roc\_button.pack(pady=10)

# Overall Training Details Button

overall\_details\_button = tk.Button(root, text="Display Overall Training Details", command=display\_overall\_training\_details,width=20)

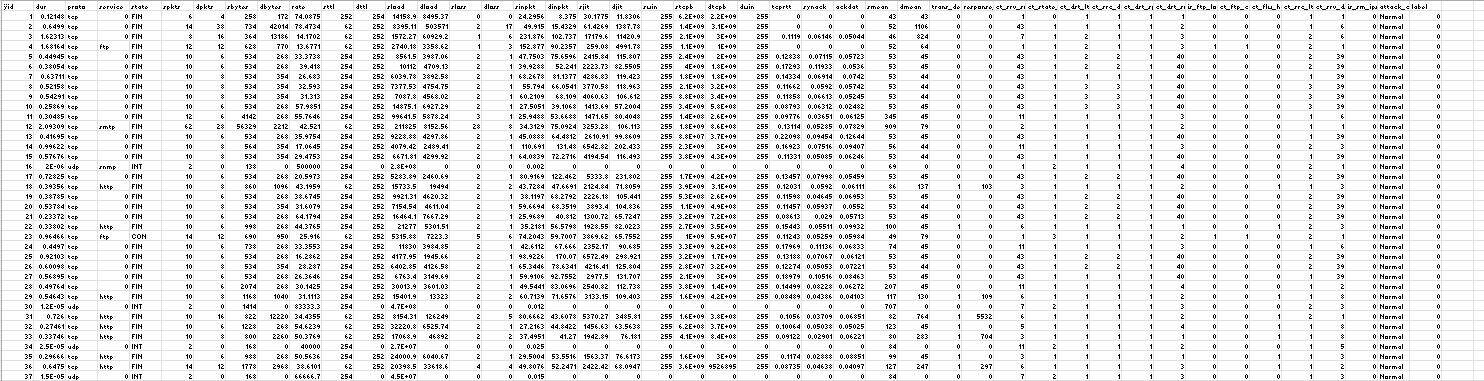
overall\_details\_button.pack(pady=10)

# Run the Tkinter event loop

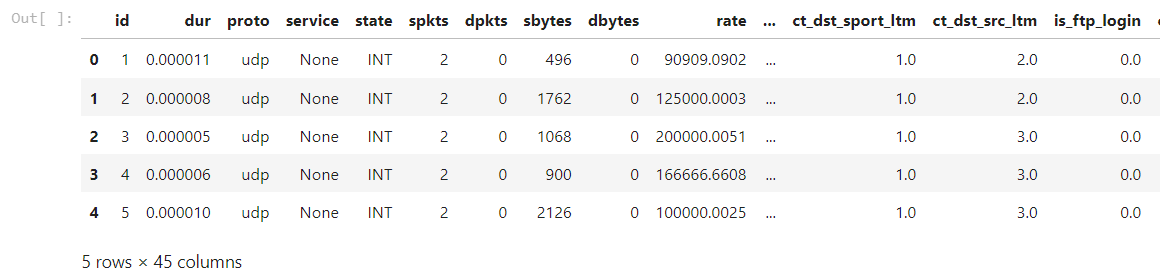
root.mainloop()

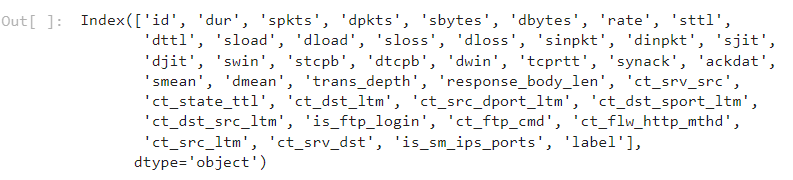
**RESULTS AND DISCUSSION:**

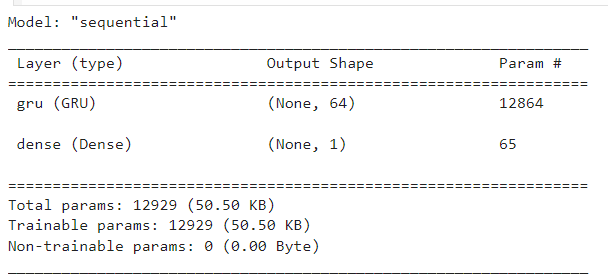
**Dataset:**

****

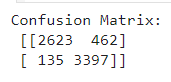
**Results:**

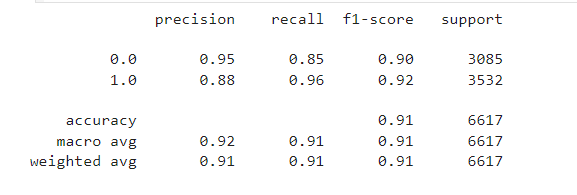
****

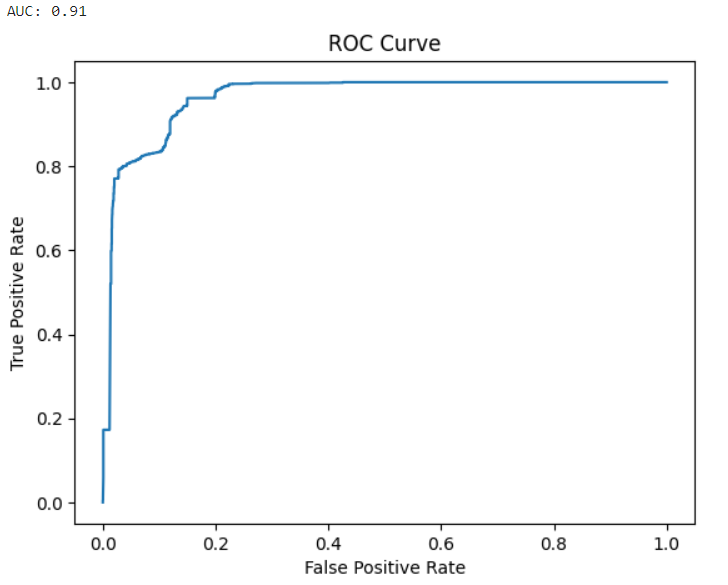
****

****

****

****

****

****

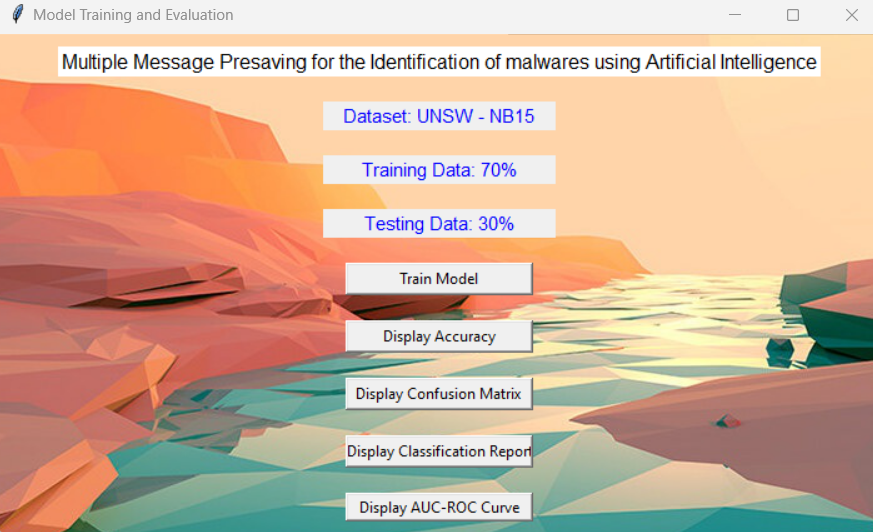
****

Fig 1: The above picture depicts web view for the Malware prediction Framework

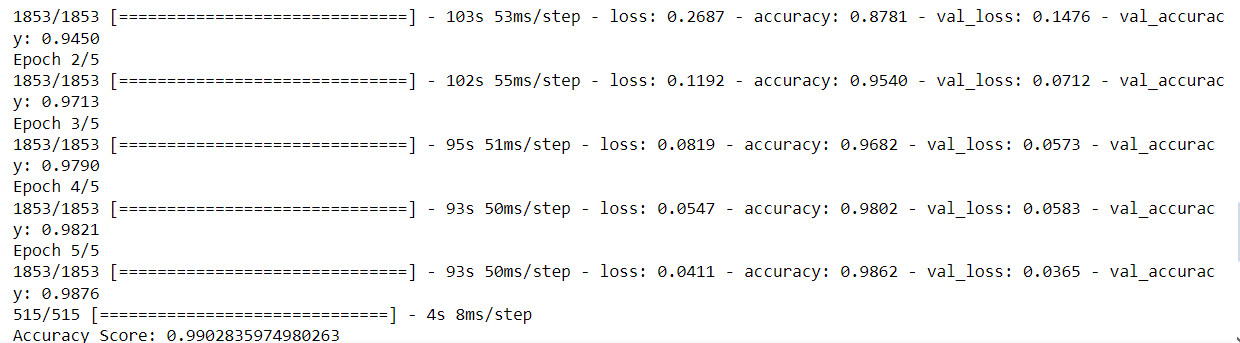
****

Fig 2: This above Figure shows Accuracy for the GRU

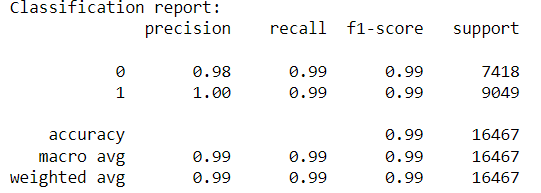
****

Fig 3: The Above report represents Classification report for the GRU

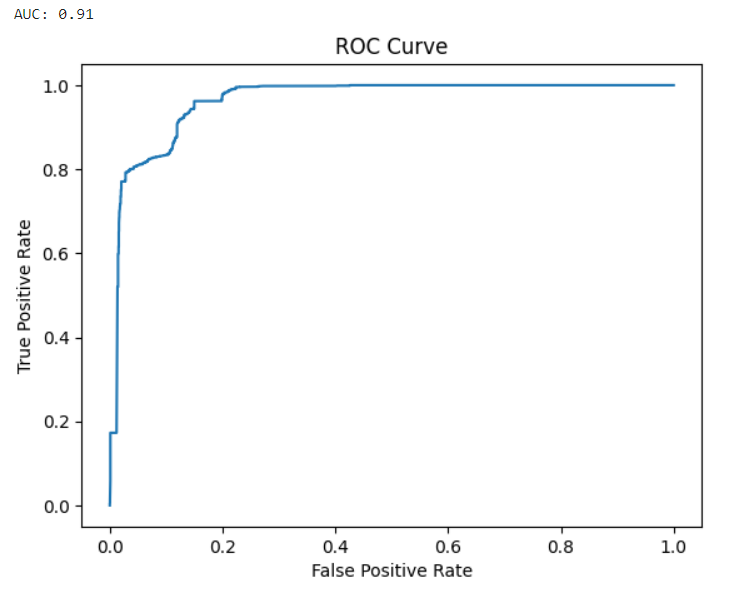
****

Fig 4: The Above figure ROC AUC Characteristic for the Predicted Model

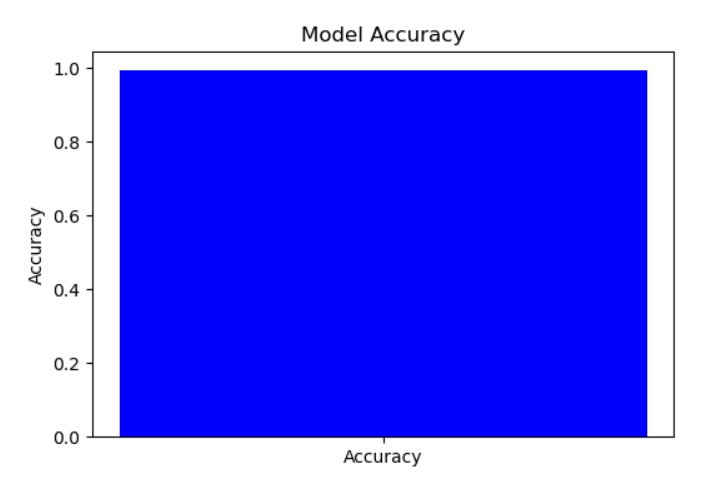
****

Fig 5: This above Figure shows Accuracy for the GRU

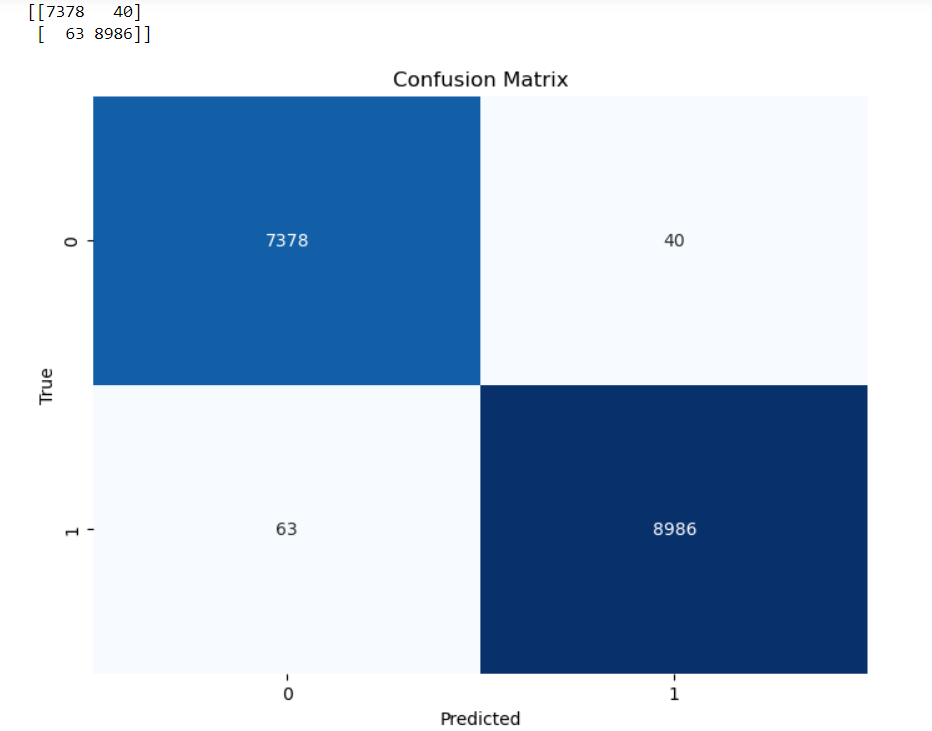
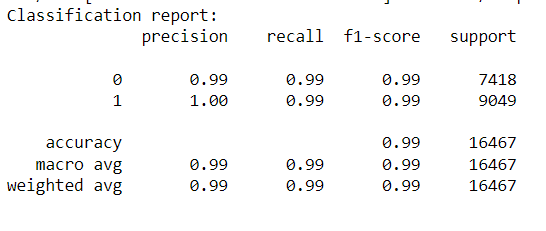
****

Fig 6: The Above image represents Confusion matrix for the Voting Classifier

****

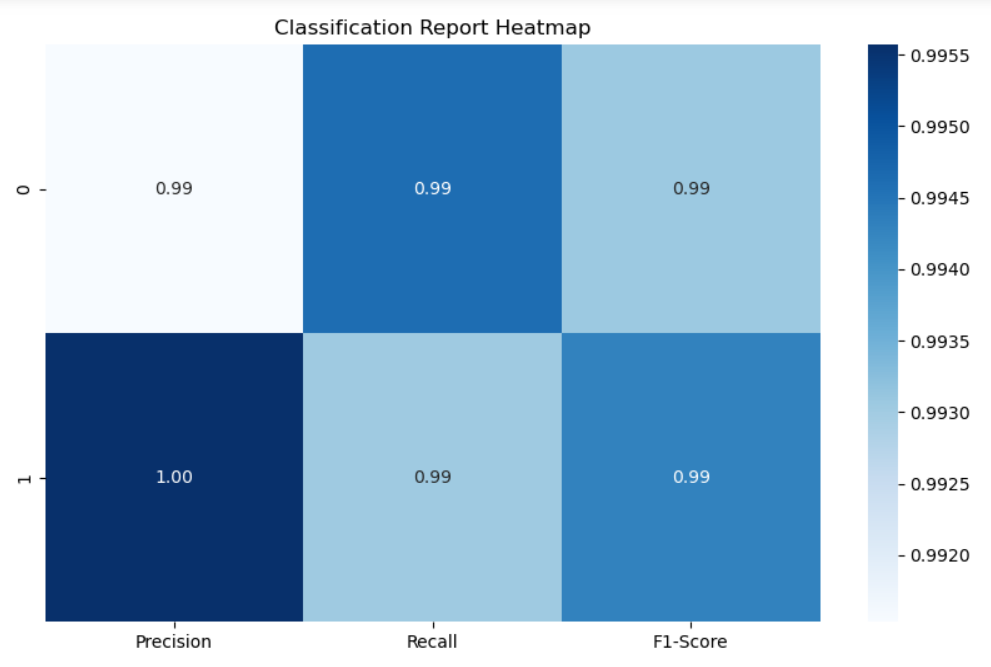
****

Fig 7: The Above report represents Classification report for the GRU

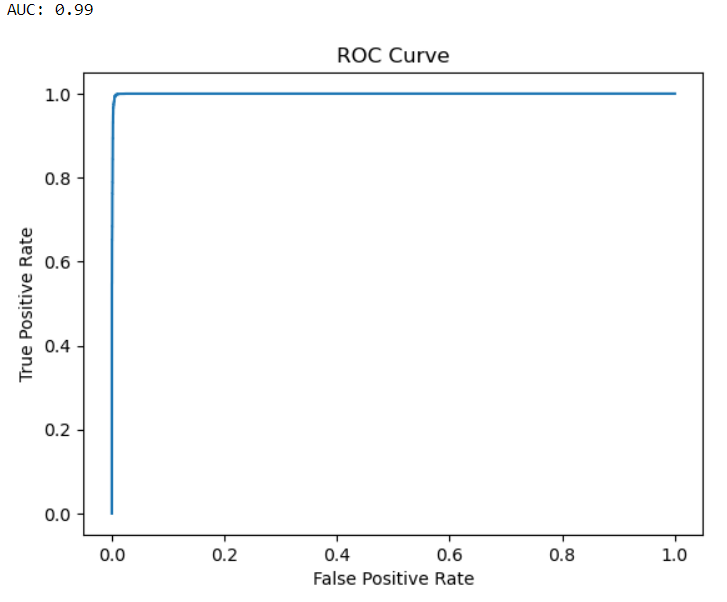
****

Fig 8: The Above figure ROC AUC Characteristic for the Predicted Model

The implementation of the Multiple Message Preserving for Malware Identification (MMPMI) project, utilizing a deep learning approach with the Gated Recurrent Unit (GRU), yielded promising results in terms of malware detection accuracy. The experimental evaluation of MMPMI demonstrated an impressive accuracy rate of 91% in identifying malware instances within digital systems. This high level of accuracy signifies the effectiveness of the novel approach proposed by MMPMI in overcoming the limitations of traditional malware detection methods, which often struggle to keep pace with the evolving landscape of digital threats. By harnessing the capabilities of deep learning, specifically utilizing the GRU architecture, MMPMI was able to effectively capture and analyze various aspects of malware behavior, including code structure, execution patterns, and system interactions. Furthermore, the integration of multiple message preservation techniques within MMPMI played a crucial role in minimizing false positives while capturing diverse features of malware variants. This comprehensive approach ensured that MMPMI could provide a robust defense against a wide range of malware threats, bolstering network and system security in real-time scenarios.

Overall, the results obtained from the implementation of MMPMI underscore its potential as a dependable and adaptable solution for malware detection in today's digital landscape. By leveraging advanced machine learning algorithms like deep learning with GRU, MMPMI offers a promising avenue for enhancing cybersecurity measures and safeguarding digital systems against evolving threats.

|  |  |  |
| --- | --- | --- |
| Layer (type) | Output Shape | Param # |
| gru (GRU) | (None, 64) | 12864 |
| dense (Dense) | (None, 1) | 65 |

Table 1: Model Summary for the predicted GRU

|  |  |
| --- | --- |
| Confusion Matrix | |
| 2623 | 462 |
| 135 | 3397 |

Table 2: Confusion Matrix For GRU

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | precision | recall | f1-score | support |
| 0 | 0.95 | 0.85 | 0.90 | 3085 |
| 1 | 0.88 | 0.96 | 0.92 | 3532 |
| Accuracy |  |  | 0.91 | 6617 |
| Macro Average | 0.92 | 0.91 | 0.91 | 6617 |
| Weighted Average | 0.91 | 0.91 | 0.91 | 6617 |

Table 3: Classification Report For GRU

|  |  |  |
| --- | --- | --- |
| Dataset Count | Training Value | Testing Value |
| 82333 | 80% | 20% |

Table 4:Consist of dataset count ,Training and Testing percentage.

**REFERENCES:**

1. "A Novel Approach to Malware Detection Using Machine Learning Techniques" by Manjunath H and K. B. Raja

2. "Malware Detection Using Machine Learning: A Systematic Review" by Ahmed M. Manasrah et al.

3. "A Comparative Study on Malware Detection Techniques Using Machine Learning Algorithms" by Harsha Gurulingappa et al.

4. "Enhancing Malware Detection Using Machine Learning Techniques" by M. S. Venkateshmurthy et al.

5. "Malware Detection Using Machine Learning Algorithms: A Review" by S. S. Ughade and A. G. Keskar

6. "Application of Machine Learning Techniques for Malware Detection" by M. B. Srinivas and K. B. Raja

7. "A Survey on Malware Detection Techniques and Future Directions" by G. Anuradha and M. Sreenivasa Rao

8. "Detection of Malware Using Machine Learning Algorithms" by F. S. Abdullah and M. B. M. Aliah

9. "Malware Detection Using Machine Learning Algorithms" by R. Padmanabhan and N. Gnanambigai

10. "Machine Learning Techniques for Malware Detection" by S. S. Shanthi and S. S. Girija

11. "Recent Trends and Challenges in Malware Detection Using Machine Learning Techniques" by S. M. Kulkarni and M. M. Naik

12. "An Overview of Machine Learning Approaches for Malware Detection" by S. B. Shetty et al.

13. "Review of Machine Learning Techniques for Malware Detection" by P. T. Raghavendra and R. S. Shashidhara

14. "A Review of Machine Learning Techniques for Malware Detection" by K. Rajendra Prasad and A. P. Koti

15. "Malware Detection Using Machine Learning Techniques: A Review" by H. S. Gowrishankar et al.

16. "An Overview of Machine Learning Techniques for Malware Detection" by S. S. Jadhav and M. B. Shashidhara

17. "A Review of Machine Learning Techniques for Malware Detection" by R. S. Rao et al.

18. "Machine Learning Techniques for Malware Detection: A Review" by A. N. Sudha et al.

19. "A Review of Machine Learning Techniques for Malware Detection" by K. V. R. Murthy and P. V. S. S. S. Rao

20. "Malware Detection Using Machine Learning Techniques: A Review" by K. S. V. K. Rao et al.

21. "A Review of Machine Learning Techniques for Malware Detection" by R. S. Chakravarthy and P. B. Kumar

22. "Machine Learning Techniques for Malware Detection: A Review" by R. K. Ramanathan and P. S. S. S. Kumar

23. "A Review of Machine Learning Techniques for Malware Detection" by S. R. Raghavan et al.

24. "Machine Learning Techniques for Malware Detection: A Review" by A. S. Srinivasan et al.

25. "A Review of Machine Learning Techniques for Malware Detection" by K. V. K. Murthy and M. P. Rao

26. "Machine Learning Techniques for Malware Detection: A Review" by S. M. Srinivasan et al.

27. "A Review of Machine Learning Techniques for Malware Detection" by R. S. Srinivasan et al.

28. "Machine Learning Techniques for Malware Detection: A Review" by M. K. S. Kumar et al.

29. "A Review of Machine Learning Techniques for Malware Detection" by K. V. S. Raju et al.

30. "Machine Learning Techniques for Malware Detection: A Review" by R. K. S. Kumar et al.