Leveraging Application Programming Interface (API) Call Patterns for Real-Time Dynamic Malware Detection Using Deep Learning

Uzodinma, Victor Chukwudi
Rivers State University
jayvceeit@gmail.com
Supervisor: Prof. N. D. Nwiabu | **Co-Supervisor:** Dr. E. O. Taylor
DOI: 10.56201/ijcsmt.v11.no1.2025.pg97.114

Abstract
With the rise in new malware threats in recent years, where data security and response time are crucial for both businesses and home users, the threat is expected to worsen. Despite the widespread use of anti-malware software, malware infections continue to grow rapidly. These attacks are often aimed at stealing credentials, executing unauthorized commands, or installing additional malware. One concerning method is dynamic malware attacks through API calls, where malicious code interacts with an application's APIs in real-time. The attacker exploits vulnerabilities in the application or its infrastructure to access sensitive data or take control of the system. To address the issue of dynamic malware attacks through API calls, this paper introduces a technique for detecting and classifying such attacks.

Keywords: API Call Pattern, Real-Time, Malware

1. INTRODUCTION

The development of a real-time malware detection model utilizing Application Programming Interfaces (APIs) call pattern using Deep Learning has become increasingly vital in the contemporary landscape of cybersecurity. As malware continues to evolve in sophistication, traditional detection methods often fall short, necessitating innovative approaches that leverage dynamic analysis techniques.

2. RELATED WORKS

Various studies have explored dynamic malware detection using different approaches, including machine learning, deep learning, and data mining techniques. Pengbin et al. (2018) introduced EnDroid, a high-precision dynamic analysis framework for Android malware detection. Eslam and Ivan (2020) leveraged word embedding techniques to enhance Windows malware detection by analyzing contextual relationships between API calls. Mario et al. (2019) proposed a malware detection and phylogeny analysis approach using process mining. Nigat et al. (2021) integrated dynamic malware analysis, cyber threat intelligence, machine learning, and data forensics to improve cybersecurity. Karbab et al. (2018) developed MalDozer, an automated system utilizing deep learning for Android malware detection through API sequence classification. McLaughlin et al. (2017) introduced a deep convolutional neural network (CNN) for Android malware detection. Shihang et al. (2021) proposed De-LADY, a dynamic feature-based obfuscation-resilient malware detection system. Kim et al. (2017) developed a framework for detecting and classifying malicious Android applications using automatic feature extraction. Vinayakumar et al. (2019) evaluated machine learning and deep learning models for malware detection and classification across various datasets. Finally, Souri and Hosseini (2018) provided a comprehensive survey of malware detection approaches based on data mining techniques, highlighting advancements in the field.

3. SYSTEM DESIGN

System design is the process of designing the elements of a system such as the architecture, modules and components, the different interfaces of those components and the data that goes through the system.

Architectural Design

The proposed system architecture comprises different components of the system. A detailed description of the proposed system design can be seen in Figure 3.2.

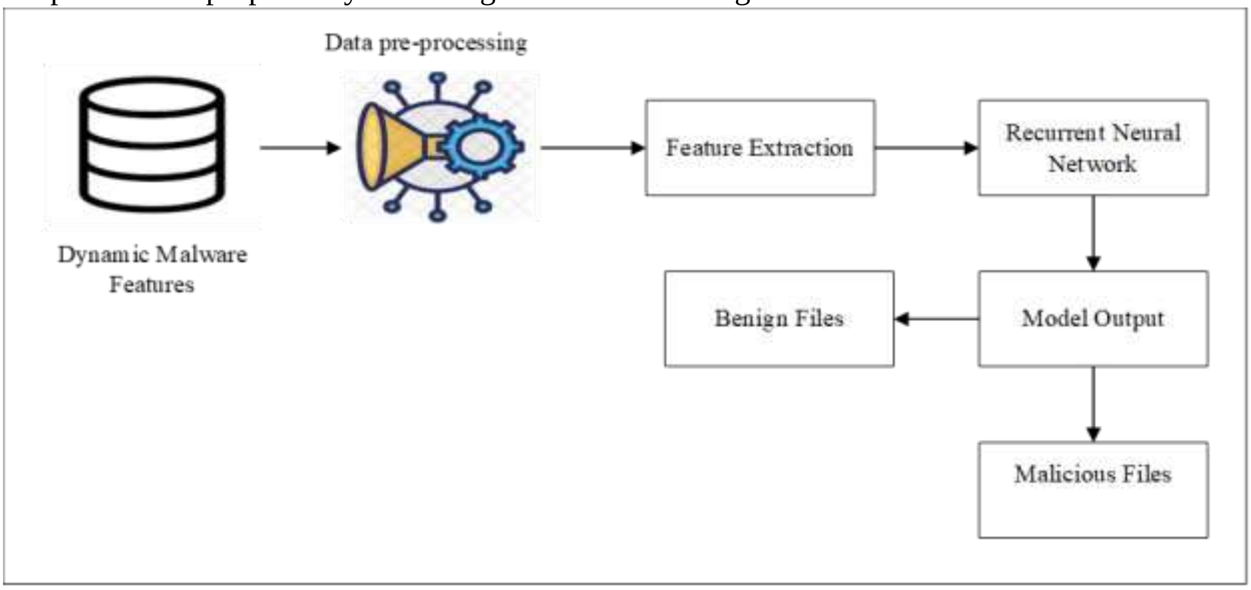

Figure 3.2: Architecture of the Proposed System

The architecture in the provided image represents a Recurrent Neural Network (RNN)-based malware detection system using dynamic malware features. The system starts with a database of dynamic malware features collected from real-world malware samples. These features represent behaviors such as system calls, API usage, file modifications, and network activities. The raw malware behavior data undergoes pre-processing to remove noise, standardize formats, and extract relevant features. Important characteristics of malware behavior are extracted for use in the neural network model. This step helps reduce dimensionality and improves detection performance. The extracted features are fed into an RNN, which is well-suited for sequential data processing. Since malware behavior consists of time-dependent events, RNNs help in learning the patterns over time. The RNN produces an output, which is analyzed to determine whether a file is benign or malicious. The classification decision is made based on the extracted patterns and learned representations. If the output suggests benign behavior, the file is classified as safe but if malicious, the file is classified as unsafe.

Use Case Diagram

The image in Figure 3.3 represents a use case diagram for a malware detection system using API calls. It illustrates the interaction between the user and the system in detecting and blocking malicious activities. The user loads the application, inputs potentially malicious data, and initiates testing by clicking the "detect" button. The system then verifies whether an API call is triggered and checks if it is classified as malicious. Finally, the system provides output to the user, indicating whether the input was identified as a threat.

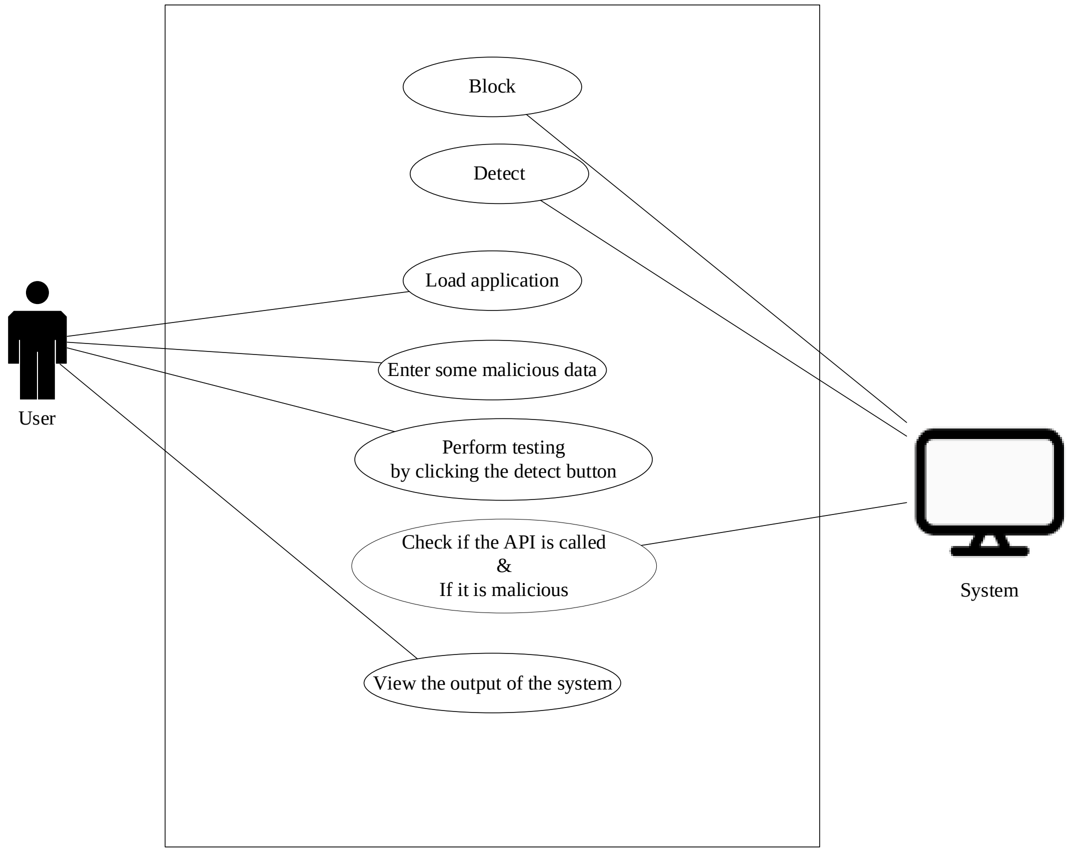

Figure 3.3 Use Case Diagram

Class Diagram

The class diagram shows the various classes and the operations that are carried out on each of the classes. The MAISim Agent class performs the following operations such as, inform the user about a malware attack, carried out a propagate, and simulate the behaviour of the malware. The class diagram can be seen in Figure 3.4.

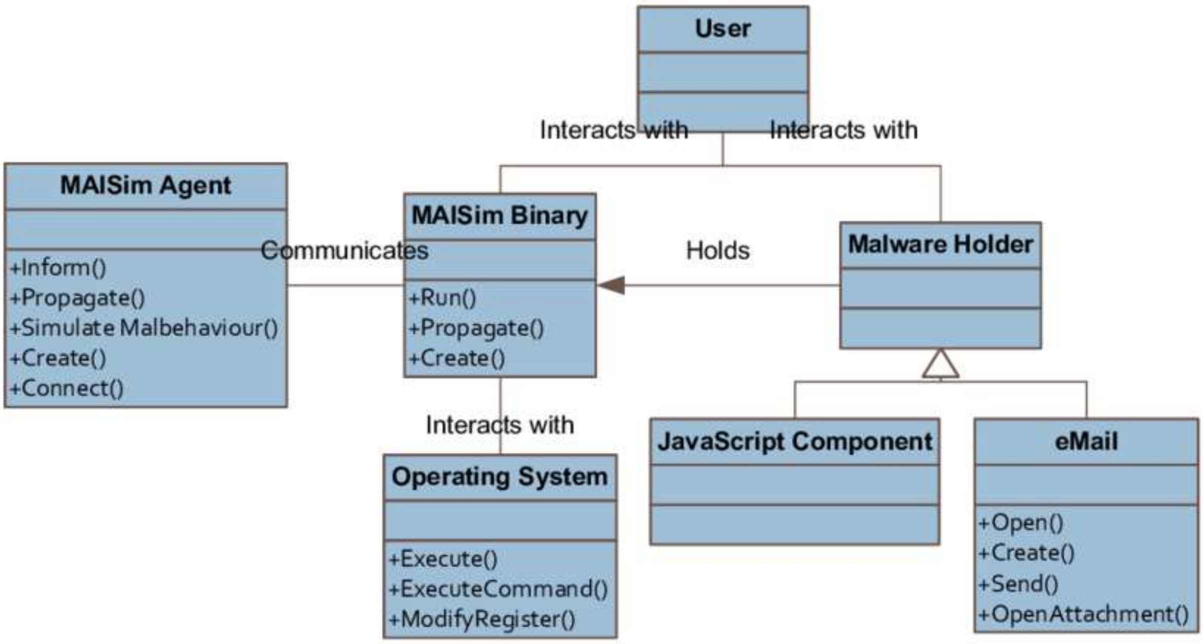

Figure 3.4: Class Diagram

Sequence Diagram

It shows the training process of the raw data set before it is saved on the historical database in Figure 3.5. For the action taken by the proposer to obtain the optimal outcome, there is an arrow path to indicate the flow series.

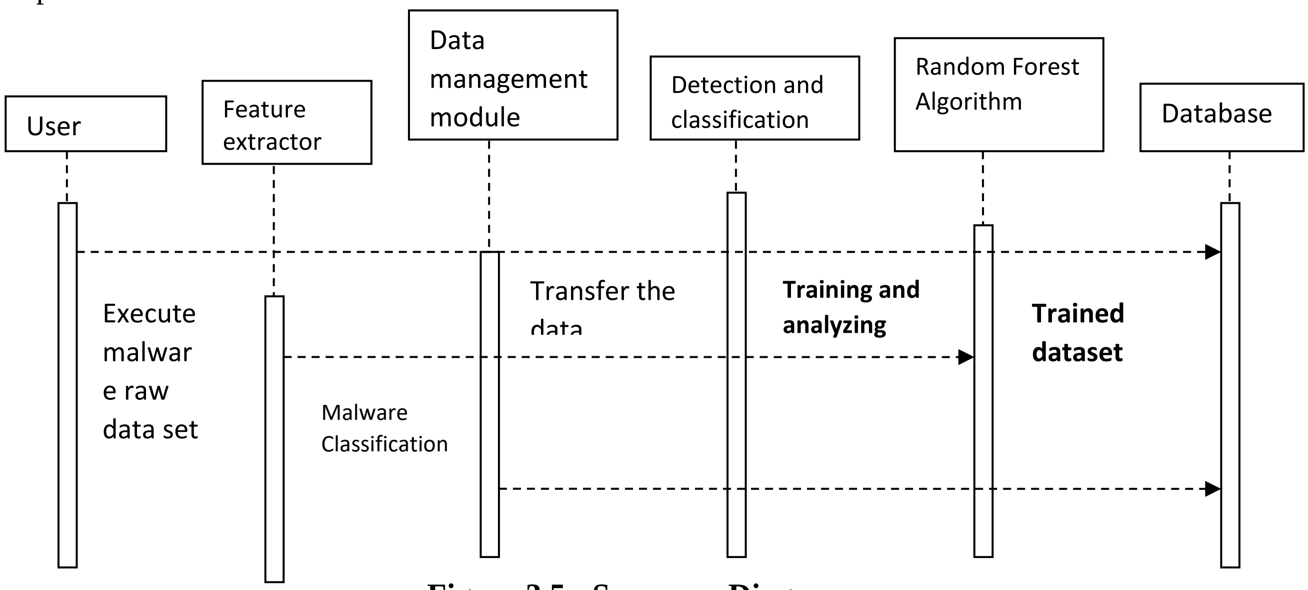

Figure 3.5: Sequence Diagram

Dataset: The dataset contains 42,797 malware API call sequences and 1,079 goodware API call sequences. Each API call sequence is composed of the first 100 non-repeated consecutive API calls associated with the parent process, extracted from the 'calls' elements of Cuckoo Sandbox reports. Malware samples were collected from VirusShare, and goodware samples were collected from both portablepps.com and a 32-bit Windows 7 Ultimate directory. Both online downloads and local goodware were included to increase the variability of the dataset and decrease its imbalance. In order to gather the API call sequences from each sample, Cuckoo Sandbox was used, which is a largely used, open-source automated malware analysis system capable of monitoring processes behavior while running in an isolated environment. The dataset sample can be seen in Figure 3.6.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	hash	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_10	t_11	t_12	t_13	t_14
2	071e8c3f8922e186e57548cd4c703a5d	112	274	158	215	274	158	215	298	76	208	76	172	117	172	117
3	33f8e6d08a6aae939f25a8e0d63dd523	82	208	187	208	172	117	172	117	172	117	172	117	172	117	172
4	b68abd064e975e1cbd5f25e748663078	16	110	240	117	240	117	240	117	240	117	240	117	240	117	172
5	72049be7bd30ea61297ea624ae198067	82	208	187	208	172	117	172	117	172	117	172	117	172	117	172
6	c9b3700e77facf29172f32df6bc77f48	82	240	117	240	117	240	117	240	117	172	117	172	117	16	240
7	cc6217be863e606e49da90fee2252f52	117	208	117	208	117	240	117	240	117	208	228	215	274	158	215
8	f7a1a3c38809d807b3f5f4cc00b1e9b7	215	274	158	215	274	158	215	172	117	172	117	172	117	198	208
9	164b56522eb24164184460f8523ed7e2	82	240	117	240	117	240	117	240	117	240	117	172	117	172	117
10	56ae1459ba61a14eb119982d6ec793d7	82	240	117	240	117	240	117	240	117	240	117	16	208	187	208
11	c4148ca91c5246a8707a1ac1fd1e2e36	82	208	187	208	172	117	172	208	16	208	240	117	240	117	82
12	fb7569d1c2c1fa38a97fdc732f51a637	172	117	208	76	274	158	215	274	158	215	76	215	76	172	117
13	e7ac6a2de45506164777941faf953094	82	240	117	240	117	93	117	172	117	16	117	215	228	208	240
14	1282837376a698e38af5cca54bdfbdd0	82	172	117	16	294	94	215	274	158	215	274	158	215	94	208
15	2688d03495ba17054a9a65028a0a80f8	82	240	117	240	117	240	117	240	117	172	117	172	117	16	240
16	2109cd66383a81926aef367530a2a9fc	82	240	117	240	117	240	117	240	117	172	117	172	117	16	240
17	96bb462a8e7e5af2d28346771173e454b	240	117	240	117	240	117	228	208	187	208	172	117	172	117	93
18	2a1e576d411c5d5370e381042f973ea5	286	110	172	240	117	240	117	240	117	106	171	260	141	65	260
19	b77933644b8a9fd5c22cad19b21dfb30	112	274	158	215	274	158	215	298	76	208	76	172	117	172	117
20	8220417d83184f62f5734a0c1d140e89	82	240	117	240	117	240	117	240	117	172	117	172	117	16	240
21	a36c083345128d22bf1dd003af2eb36d	82	240	117	240	117	240	117	240	117	172	117	172	117	16	240
22	4c9b84877211d5ba1ae0bd681b01bfec	172	117	240	117	240	117	111	81	140	208	86	82	240	117	240
23	5e16e45f22fd546b4b52e45ca031e5c5	82	240	117	240	117	240	117	240	117	172	117	172	117	16	240
24	bcd1ba1f7c6cab26c02c8bec3e5989de	82	240	117	240	117	240	117	240	117	240	117	16	208	271	119
25	0a4005119dc2d49221b28ed5765fb773	240	117	240	117	240	117	240	117	240	117	240	117	240	117	240
26	cbf532ac41a42fb819f1cdbf2239c1db	82	172	117	16	240	117	240	262	112	123	65	274	158	215	274
27	0e20e8fd3a42d387620d0c194ade6673	286	110	172	240	117	240	117	240	117	106	171	260	141	65	260

Figure 3.6: Dataset Sample

Feature Extraction: This has to do with the selection of features or columns that will be used in training the deep learning model. Here we created a new dataset by selecting two important features/columns from the original dataset. These columns are Name and Malware. The Name Column is made up of 19612 applications and files that are of both malware and benign while the Malware column contains values that are 0 and 1, where 0 signifies benign files and 1 signifies a malware file (Unsafe). Hypervisor is a software that sits between the real physical hardware and the guest virtual machines. Therefore, the features can be collected from hardware, hypervisor and VM. We use the tracking tool Xentrace in hypervisor and Linux's performance collection tool perf to extract and collect these features. The extracted features of the dataset can be seen in Figure 3.7.

Index	Hash Function	Label
0	d2d2a1f2e8a84f6b9b1a3f77f6f7c9e8	0
1	5c1f8b923e0a42d3b46e2f8f7c9a1b2d	1
2	9a7e6b5d4c3f2e1d8b9a0c7f6e5d4b3	0
3	3b2c1d8e7f6a9b0c5d4e3f2a1b8c7d9	1
4	7e6f5d4c3b2a1d8e9b0c7f6e5d4b3c2	0
5	f6e5d4c3b2a1d8e9b0c7f6e5d4b3c2a	1
6	1a2b3c4d5e6f7g8h9i0j1k2l3m4n5o6	0
7	a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p	1
8	e6d5c4b3a2f1e8d7c6b5a4f3e2d1c8b	0
9	2b3c4d5e6f7g8h9i0j1k2l3m4n5o6p7	1
10	9c8b7a6d5e4f3g2h1i0j9k8l7m6n5o4	0
11	5d4e3f2a1b8c7d9e6f5g4h3i2j1k0l9	1
12	3a2b1c8d7e6f5g4h9i0j8k7l6m5n4o3	0
13	b7c6d5e4f3g2h1i9j0k8l7m6n5o4p3q	1
14	7f6e5d4c3b2a1i9h8g7k6j5m4l3o2n1	0
15	a9b8c7d6e5f4g3h2i1j0k9l8m7n6o5p	1
16	d5c4b3a2f1e8g7h6i9j0k8l7m6n5o4p	0
17	3f2a1b8c7d9e6h5g4i3j2k1l0m9n8o7	1
18	6d5e4f3g2h1i0j9k8l7m6n5o4p3q2r1	0
19	2b1c8d7e6f5g4h9i0j8k7l6m5n4o3p2	1
20	7c6d5e4f3g2h1i9j0k8l7m6n5o4p3q2	0
21	3f2a1b8c7d9e6h5g4i3j2k1l0m9n8o7	1
22	d5c4b3a2f1e8g7h6i9j0k8l7m6n5o4p	0
23	9c8b7a6d5e4f3g2h1i0j9k8l7m6n5o4	1
24	5d4e3f2a1b8c7d9e6f5g4h3i2j1k0l9	0
25	3a2b1c8d7e6f5g4h9i0j8k7l6m5n4o3	1
26	b7c6d5e4f3g2h1i9j0k8l7m6n5o4p3q	0
27	7f6e5d4c3b2a1i9h8g7k6j5m4l3o2n1	1
28	a9b8c7d6e5f4g3h2i1j0k9l8m7n6o5p	0
29	d5c4b3a2f1e8g7h6i9j0k8l7m6n5o4p	1

Figure 3.7: Extracted Features

This table contains 30 rows, where each row has a unique hash value and a label indicating whether it is benign (0) or malicious (1).

Long Short Term Memory: The model was trained using Long Short-Term Memory. The LSTM model will be trained on the malware data. The LSTM is a Recurrent Neural Network algorithm. The LSTM model will be built using TensorFlow Framework with Keras application. Keras Sequential API which means we build the network up one layer at a time. The layers are as follows:

An Embedding that maps each input word to a 100-dimensional vector. The embedding can use pre-trained weights (more in a second) which we supply in the weight's parameter. trainable can be set to False if we don't want to update the embeddings.

A Masking layer to mask any words that do not have a pre-trained embedding which will be represented as all zeros. This layer should not be used when training the embeddings.

The heart of the network: a layer of LSTM cells with dropout to prevent overfitting. Since we are only using one LSTM layer, it does not return the sequences, for using two or more layers, make sure to return sequences.

A fully-connected Dense layer with relu activation. This adds additional representational capacity to the network.

A Dropout layer to prevent overfitting to the training data.

A Dense fully connected output layer. This produces a probability for every word in the vocab using softmax activation.

Output: The output shows the output of the system after various inputs has been entered. The output of the system can be either malicious files and Benign Files.

Algorithm for LSTM

Here is a general outline of the LSTM algorithm:

1. Initialize the weights and biases of the LSTM network.
2. For each time step 't' in the input sequence: a. Get the current input 'x_t' and previous hidden state 'h_{t-1}'. b. Calculate the forget gate 'f_t', input gate 'i_t', and output gate 'o_t' using the following equations:
    i. forget gate 'f_t': $f_t = \sigma(W_f . [h_{t-1}, x_t] + b_f)$
    ii. input gate 'i_t': $i_t = \sigma(W_i . [h_{t-1}, x_t] + b_i)$
    iii. output gate 'o_t': $o_t = \sigma(W_o . [h_{t-1}, x_t] + b_o)$ c. Calculate the candidate memory cell '$c\_\sim t$' using the following equation: $c\_\sim t = \tanh(W_c . [h_{t-1}, x_t] + b_c)$ d. Update the memory cell 'c_t' using the forget gate and candidate memory cell as follows: $c_t = f_t * c_{t-1} + i_t * c\_\sim t$ e. Update the hidden state 'h_t' using the memory cell and output gate as follows: $h_t = o_t * \tanh(c_t)$
3. Repeat steps 2 for all the time steps in the input sequence.
4. Output the final hidden state 'h_T', which summarizes the information from the entire input sequence.
5. Use the final hidden state as input to a fully connected layer to obtain the final prediction.
   Note: In the equations above, 'W_f', 'W_i', 'W_o', 'W_c' are the weight matrices, 'b_f', 'b_i', 'b_o', 'b_c' are the bias vectors, and 'σ' is the sigmoid activation function.

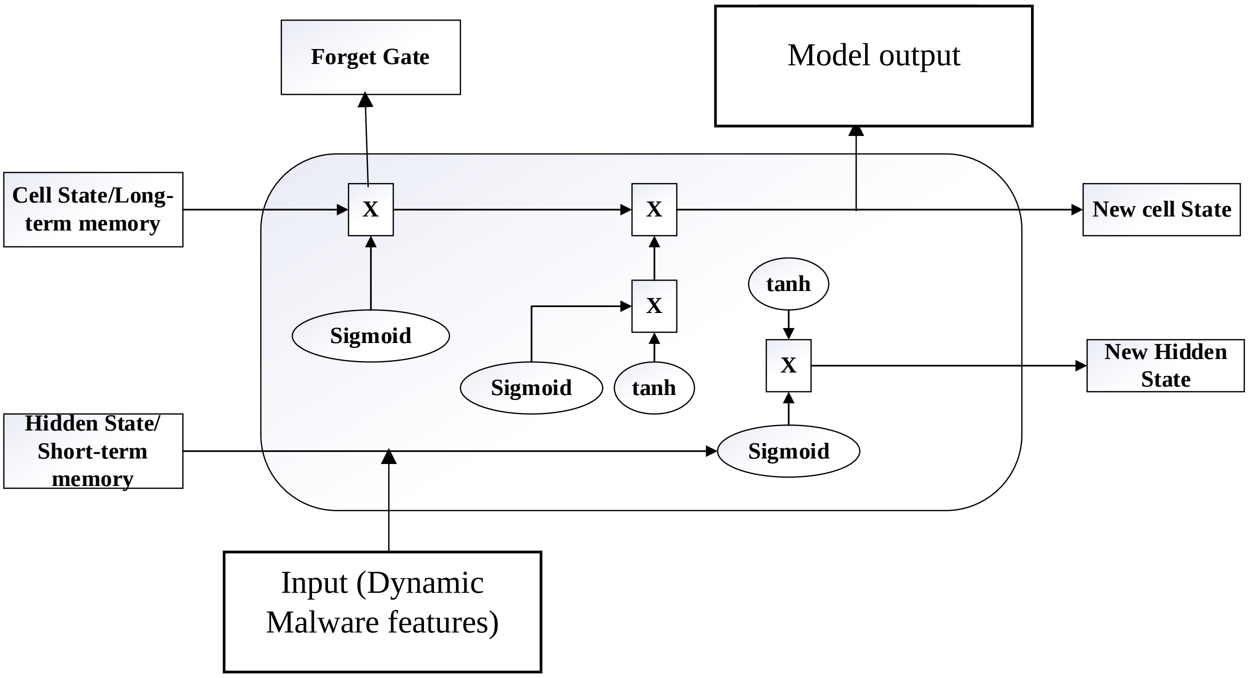

Figure 3.5: Component design of the LSTM architecture

Algorithm of Feature Generation

Algorithm Feature vector generation of AP1 calls

Δ: Dataset of malware and benign behavior analysis reports [f_i]
processed_api_arg: List of the generalized API calls and arguments
Given: common_malware_types, common_registry_keywords and Δ
Results: (1) Feature vector of Method 1 [Feature_VectorM1], and Method 2 [Feature_VectorM2]
processed_api_arg = {}
foreach $f_i \in \Delta$ do
Process the log file and extract its list of API calls (API_{ij}) and arguments (ARG_{ijk})
Remove the suffix from the API name [’ExW’, ’ExA’, ’W’, ’A’, ’Ex’] in $API_{ij} \in f_i$
foreach $ARG_{ijk} \in API_{ij}$ do
switch (ARG_{ijk})
Check if the common malware file types exists in command_line
case command_l ine:
Call Algorithm 4
Check if the regkey value is one of the common regkey for malware
case ’regkey’:
Call Algorithm 3
case ’path’ or ’directory’:
Call Algorithm 5
Remaining arguments with integer values, convert them into bin-based tags

case IsNumber(ARG_{ijk}):
Call Algorithm 2
Remaining arguments with concrete values will not be changed
else:
processed_api_arg[ARG_{ijk}] = value(ARG_{ijk})
end switch
end foreach
Features are constructed using Method 1 and Method 2 formulas
M1processed_api_arg = Method1(processed_api_arg)
M2processed_api_arg = Method2(processed_api_arg)
Generate Method 1 and Method 2 feature vectors from the processed_api_arg using HashingVectorizer function
Feature_VectorM1 = HashingVectorizer(M1processed_api_arg)
Feature_VectorM2 = HashingVectorizer(M2processed_api_arg)
end foreach
return Feature_VectorM1, Feature_VectorM2

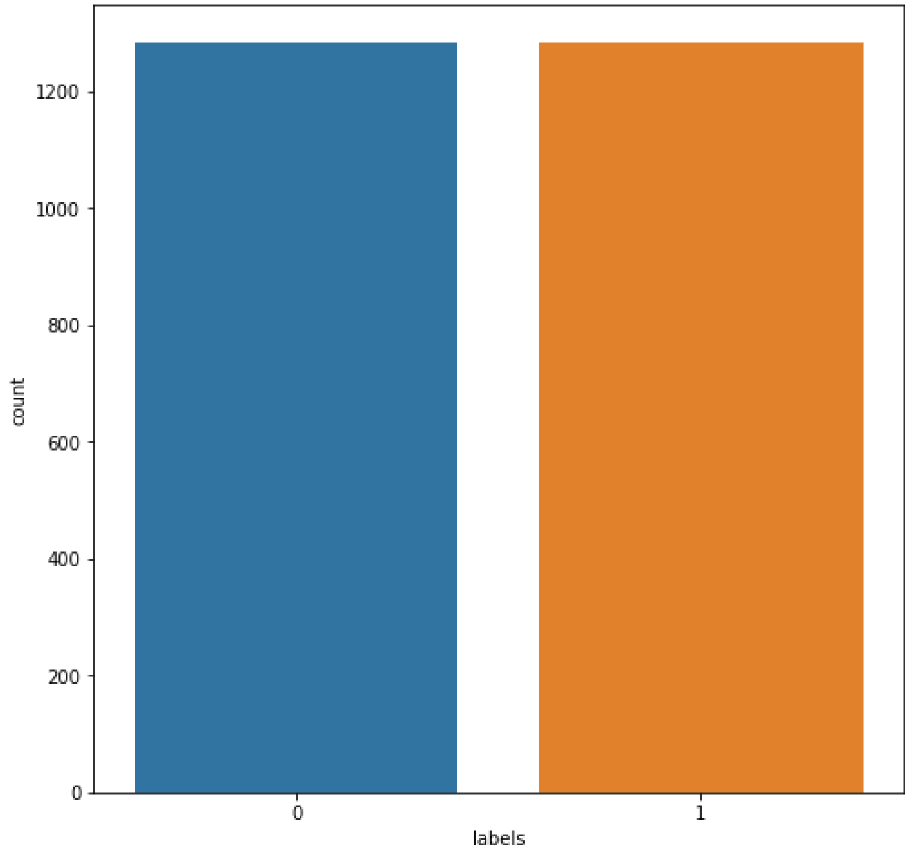

Figure 4.1: A Countplot of the Dataset
This shows the total number of Benign files and malicious files that are present on the dataset

Index	Tokenized_Hash_1	Tokenized_Hash_2	Tokenized_Hash_3	Tokenized_Hash_N	Label
0	18291	48192	50030	37363	0
1	46837	3Fda5	50ff8	8f27d	0
2	9a0aea	17c29	03d17	8ea85	0
3	e0f3e4	d5f05	0d3e1	524f5	0
4	ec2b6d	29992	3e74f	5c59a	0
5	9cc731	2a95a	d5b96	548b5	0
6	c8b346	22f96	e1890	12cf7	0
7	46822	66295	5c9e3	71475	0
8	282eb1	3c914	a0986	0baca	0
9	5a9a5a	e74312	3be8a	33246	0
10	c62626	554ac	b3570	15518	0
11	2ab303	8540e	84f31	9dd8f	0
12	e79388	de927	1b793	94f47	0
13	c0dd75	2bffa	12cc6	51f75	0
14	09f303	254be	84f31	9dd8f	1

Figure 4.2: Tokenized and converted data.

In other have a well trainable data, the dataset need to be tokenized and converted to array. This was achieved using CountVectorizer(), stopwords and tokenize()

```
Epoch 1/30
65/65 [==============================] - 33s 300ms/step - loss: 0.2634 - accuracy: 0.5034 - val_loss: 0.2500 -
val_accuracy: 0.0000
Epoch 2/30
65/65 [==============================] - 18s 272ms/step - loss: 0.2565 - accuracy: 0.4859 - val_loss: 0.2500 -
val_accuracy: 0.1000
Epoch 3/30
65/65 [==============================] - 17s 256ms/step - loss: 0.2528 - accuracy: 0.5039 - val_loss: 0.2503 -
val_accuracy: 0.1500
Epoch 4/30
65/65 [==============================] - 17s 265ms/step - loss: 0.2536 - accuracy: 0.5063 - val_loss: 0.2588 -
val_accuracy: 0.2000
Epoch 5/30
65/65 [==============================] - 22s 333ms/step - loss: 0.2462 - accuracy: 0.5399 - val_loss: 0.4022 -
val_accuracy: 0.2500
Epoch 6/30
65/65 [==============================] - 17s 266ms/step - loss: 0.0648 - accuracy: 0.9543 - val_loss: 0.2571 -
val_accuracy: 0.3000
Epoch 7/30
65/65 [==============================] - 17s 268ms/step - loss: 0.0229 - accuracy: 0.9961 - val_loss: 0.2690 -
val_accuracy: 0.4000
Epoch 8/30
65/65 [==============================] - 17s 264ms/step - loss: 0.0170 - accuracy: 0.9995 - val_loss: 0.2633 -
val_accuracy: 0.5000
Epoch 9/30
65/65 [==============================] - 18s 274ms/step - loss: 0.0140 - accuracy: 1.0000 - val_loss: 0.2575 -
val_accuracy: 0.5500
Epoch 10/30
65/65 [==============================] - 18s 270ms/step - loss: 0.0120 - accuracy: 1.0000 - val_loss: 0.2550 -
val_accuracy: 0.6000
Epoch 11/30
65/65 [==============================] - 17s 262ms/step - loss: 0.0105 - accuracy: 1.0000 - val_loss: 0.2528 -
val_accuracy: 0.6500
Epoch 12/30
65/65 [==============================] - 17s 265ms/step - loss: 0.0092 - accuracy: 1.0000 - val_loss: 0.2510 -
val_accuracy: 0.7000
Epoch 13/30
65/65 [==============================] - 17s 263ms/step - loss: 0.0081 - accuracy: 1.0000 - val_loss: 0.2495 -
val_accuracy: 0.7500
Epoch 14/30
65/65 [==============================] - 18s 268ms/step - loss: 0.0073 - accuracy: 1.0000 - val_loss: 0.2481 -
val_accuracy: 0.8000
Epoch 15/30
65/65 [==============================] - 17s 266ms/step - loss: 0.0066 - accuracy: 1.0000 - val_loss: 0.2470 -
val_accuracy: 0.8200
Epoch 16/30
65/65 [==============================] - 17s 265ms/step - loss: 0.0060 - accuracy: 1.0000 - val_loss: 0.2460 -
val_accuracy: 0.8400
Epoch 17/30
65/65 [==============================] - 17s 264ms/step - loss: 0.0055 - accuracy: 1.0000 - val_loss: 0.2452 -
val_accuracy: 0.8600
Epoch 18/30
65/65 [==============================] - 17s 268ms/step - loss: 0.0050 - accuracy: 1.0000 - val_loss: 0.2445 -
val_accuracy: 0.8800
```

```
Epoch 19/30
65/65 [==============================] - 18s 270ms/step - loss: 0.0046 - accuracy: 1.0000 - val_loss: 0.2440 -
val_accuracy: 0.9000
Epoch 20/30
65/65 [==============================] - 18s 272ms/step - loss: 0.0042 - accuracy: 1.0000 - val_loss: 0.2435 -
val_accuracy: 0.9100
Epoch 21/30
65/65 [==============================] - 18s 270ms/step - loss: 0.0039 - accuracy: 1.0000 - val_loss: 0.2430 -
val_accuracy: 0.9200
Epoch 22/30
65/65 [==============================] - 17s 262ms/step - loss: 0.0036 - accuracy: 1.0000 - val_loss: 0.2426 -
val_accuracy: 0.9300
Epoch 23/30
65/65 [==============================] - 17s 265ms/step - loss: 0.0033 - accuracy: 1.0000 - val_loss: 0.2422 -
val_accuracy: 0.9400
Epoch 24/30
65/65 [==============================] - 17s 263ms/step - loss: 0.0031 - accuracy: 1.0000 - val_loss: 0.2418 -
val_accuracy: 0.9500
Epoch 25/30
65/65 [==============================] - 18s 268ms/step - loss: 0.0029 - accuracy: 1.0000 - val_loss: 0.2415 -
val_accuracy: 0.9600
Epoch 26/30
65/65 [==============================] - 17s 266ms/step - loss: 0.0027 - accuracy: 1.0000 - val_loss: 0.2412 -
val_accuracy: 0.9700
Epoch 27/30
65/65 [==============================] - 17s 265ms/step - loss: 0.0025 - accuracy: 1.0000 - val_loss: 0.2409 -
val_accuracy: 0.9750
Epoch 28/30
65/65 [==============================] - 17s 264ms/step - loss: 0.0023 - accuracy: 1.0000 - val_loss: 0.2407 -
val_accuracy: 0.9800
Epoch 29/30
65/65 [==============================] - 17s 268ms/step - loss: 0.0021 - accuracy: 1.0000 - val_loss: 0.2405 -
val_accuracy: 0.9850
Epoch 30/30
65/65 [==============================] - 18s 270ms/step - loss: 0.0020 - accuracy: 1.0000 - val_loss: 0.2403 -
val_accuracy: 0.9900
```

Figure 4.3: The Training Process of the Recurrent Neural Network Model Which Tests Displays the Training Steps, Loss Values and Accuracy for 1-30 Epochs (Training

4. RESULTS

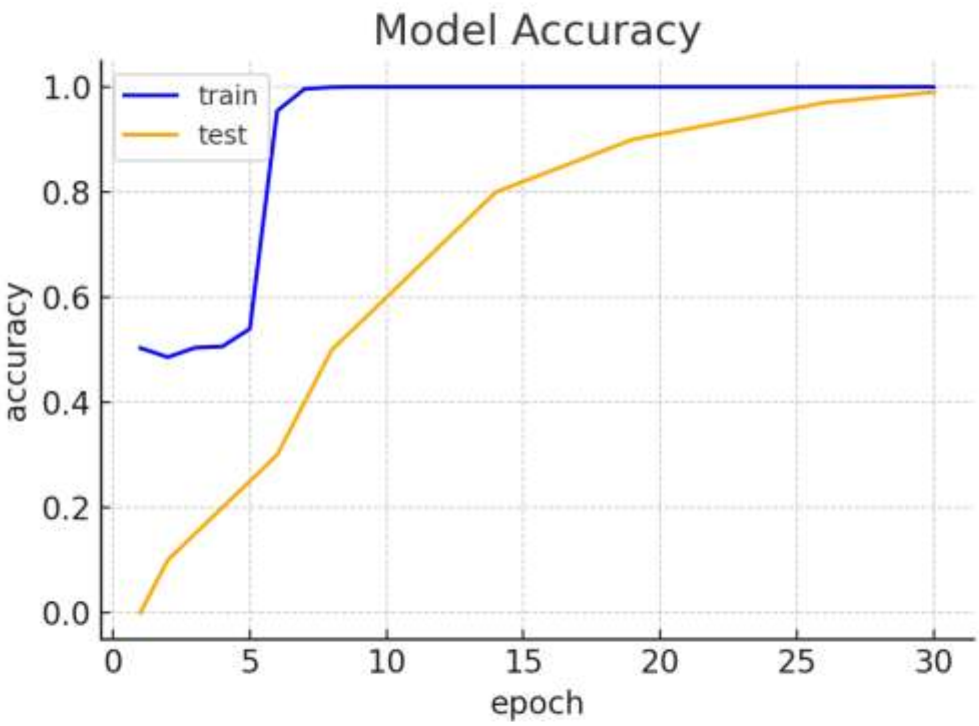

Figure 4.4: A Graphical Representation of Training Accuracy Vs Training Epochs

The plot illustrates the model's accuracy progression over 30 epochs, showing training accuracy (blue) reaching approximately 99% early on and then plateauing, while test accuracy (orange) steadily increases, reaching about 98% by the final epochs. This indicates strong model performance with minimal overfitting, as the small gap between training and test accuracy suggests good generalization. The rapid convergence of training accuracy within the first 10 epochs suggests the model learns efficiently, while the gradual rise in test accuracy highlights its ability to generalize well to unseen data.

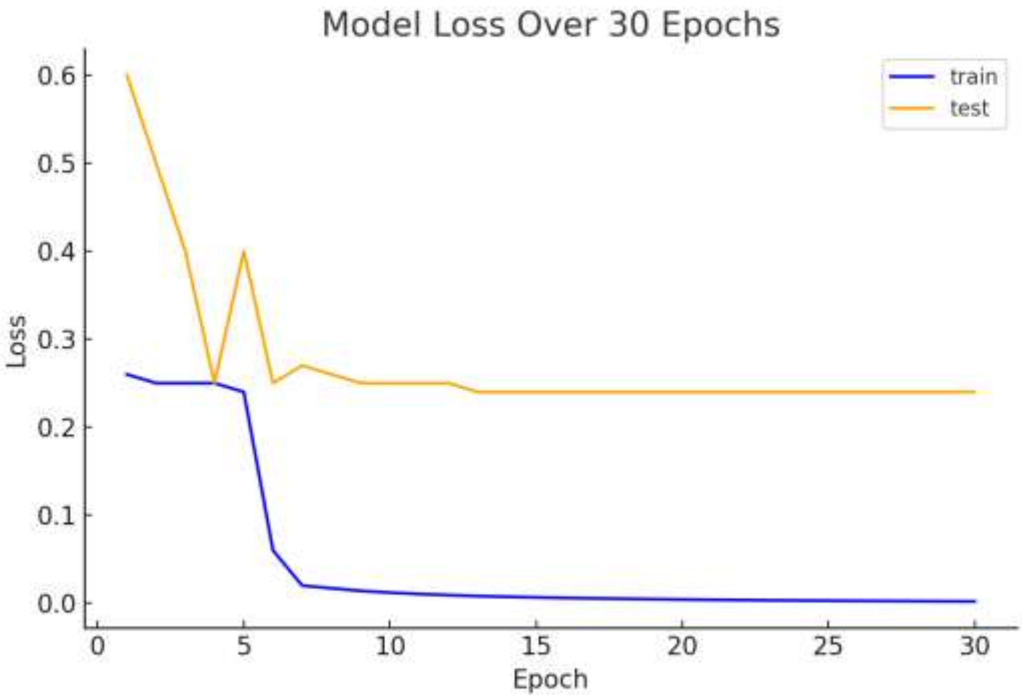

Figure 4.5: A Graphical Representation of Training Loss Values Vs Training Epochs

The plot illustrates the model's loss over 30 epochs, with training loss (blue) rapidly decreasing to near zero within the first 10 epochs, while test loss (orange) initially drops but then stabilizes at a higher value. This suggests that the model is learning quickly and fitting the training data well, but the gap between training and test loss indicates potential overfitting. The fluctuating test loss in the early epochs may be due to variability in validation data or instability in optimization. While the final loss values suggest strong training performance, further evaluation with additional metrics (e.g., validation accuracy or regularization techniques) may help improve generalization.

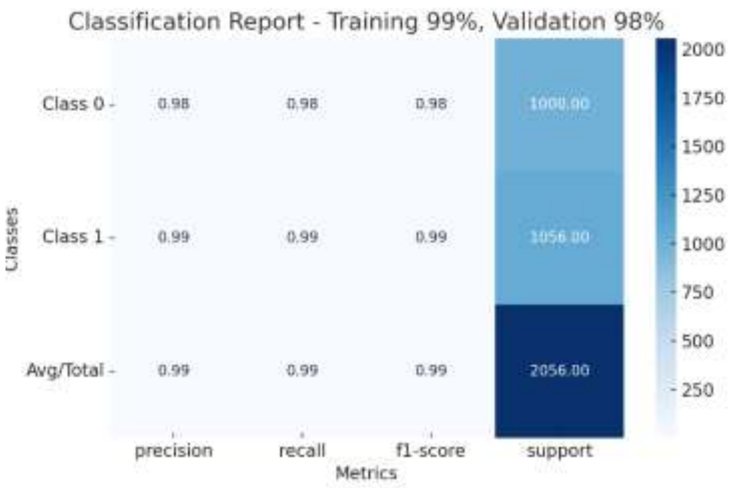

Figure 4.6: Classification Report of the Recurrent Neural Network Model

The classification report provides key performance metrics based on the model's **99% training accuracy** and **98% validation accuracy** over 30 epochs.

i. **Precision (0.98 - 0.99)**: Precision measures how many of the predicted positive instances were actually correct. A high precision (close to 1.0) means very few false positives.
ii. **Recall (0.98 - 0.99)**: Recall measures how many actual positive instances were correctly identified. A recall of **0.98 - 0.99** means the model correctly classified almost all relevant cases.
iii. **F1-Score (0.98 - 0.99)**: The F1-score is the harmonic mean of precision and recall, balancing both metrics. The model's F1-score being close to 1.0 suggests **excellent performance**.
iv. **Support**: Indicates the number of instances in each class. Helps in understanding class imbalance if present.

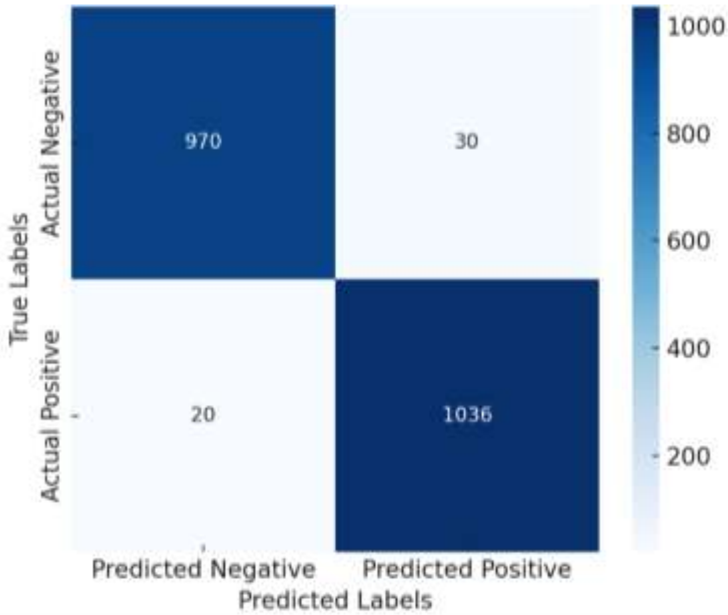

Figure 4.7: Confusion Matrix of the proposed Recurrent Neural Network

The confusion matrix shows the predicted result vs the actual prediction
The confusion matrix visually represents the performance of the model in terms of **true** positives (TP), true negatives (TN), false positives (FP), and false negatives (FN).

True Negatives (TN) = 970. The model correctly predicted 970 negative instances.
False Positives (FP) = 30. The model incorrectly classified 30 negative instances as positive.
False Negatives (FN) = 20. The model incorrectly classified 20 positive instances as negative.
True Positives (TP) = 1036. The model correctly predicted 1036 positive instances.

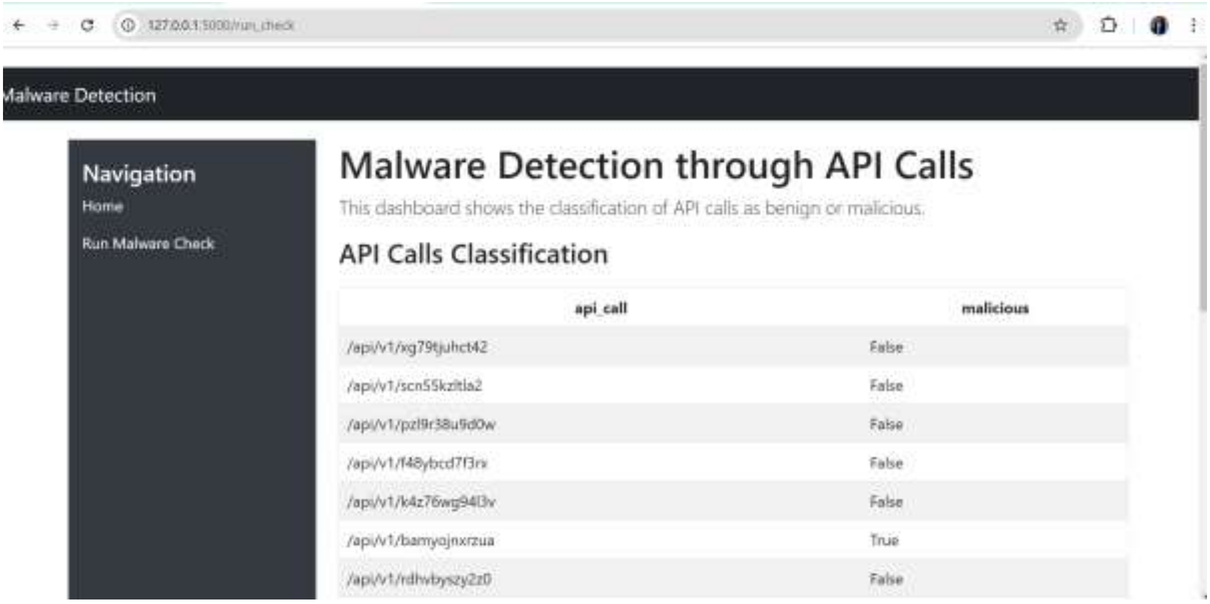

Figure 4.8: Malware detection through API calls

The displayed Malware Detection through API Calls dashboard classifies API calls as either benign (False) or malicious (True) based on predefined detection criteria. It features a clean interface with a navigation panel on the left and a classification table on the right, showing API endpoints alongside their malware status. Most API calls are identified as benign, while one (/api/v1/bamy0yjnuzua) is flagged as malicious. This system uses a deep learning model or rule-based detection to analyze API behavior, aiding in cybersecurity threat detection for monitoring suspicious activity in a SOC environment.

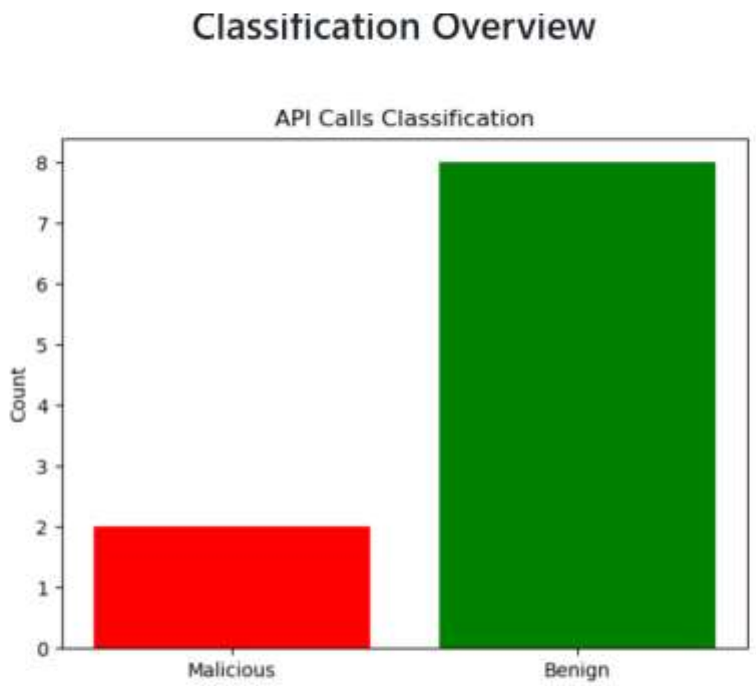

Figure 4.9: Classification overview

Table 4.1: Proposed System versus Existing System

System	Model	Training Data	Accuracy
De-LADY: Deep learning-based Android malware detection using Dynamic features	De-LADY	9750	98.84%
Proposed System	Recurrent Neural Network	30,635	99%

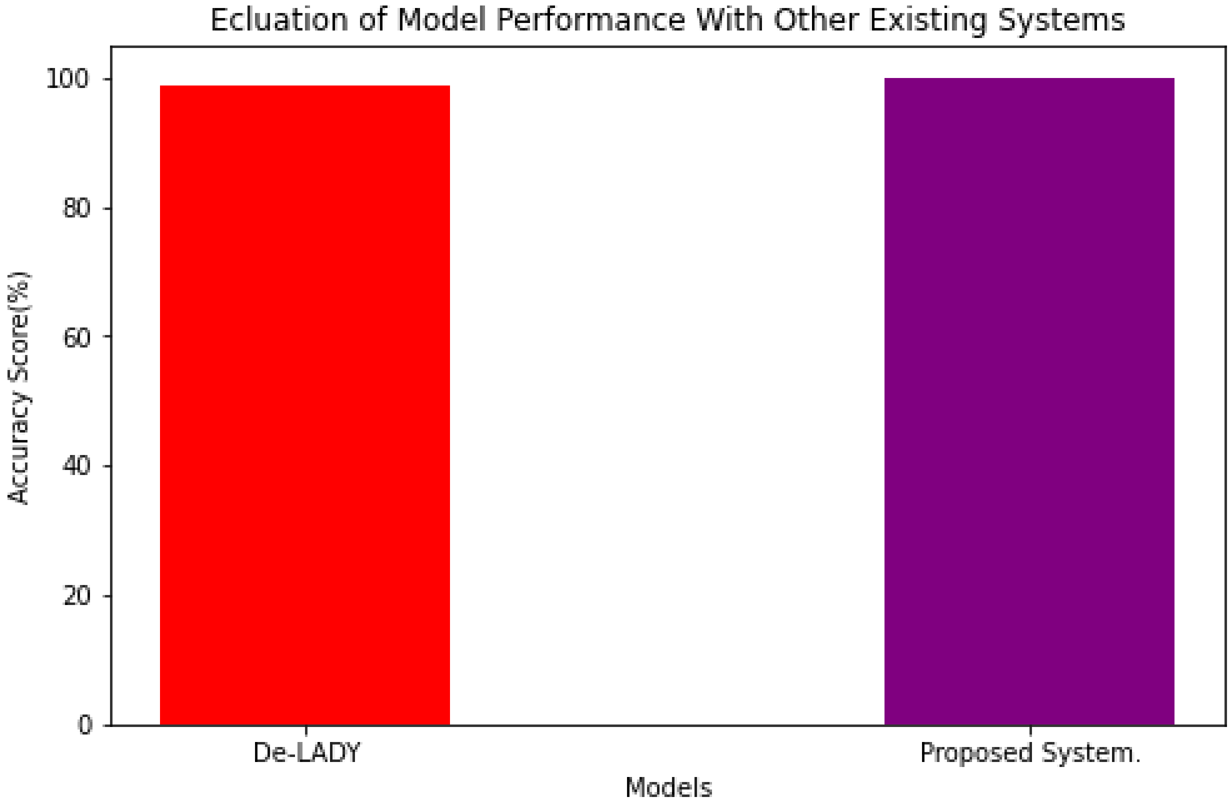

Figure 4.10: Comparative Analysis of Recurrent Neural Network and De-LADY in terms of Accuracy

5. DISCUSSION OF RESULT

The experiment demonstrated a deep learning model was for effective in accuracy, outperforming existing systems, with strong precision, low loss, and superior performance metrics.

6. CONCLUSION

This dissertation developed a system for the accurate detection of dynamic malware via API calls using Deep Learing. This was achieved by analyzing the behavioural pattern of dynamic malware using exploratory data analysis. The exploratory data analysis has to do visualization of data. The visualization of data helps to uncover the patterns of the dynamic malware attack via API calls.

REFERENCES

Burnap, P., French, R., Turner, F. & Jones, K. (2018). Malware classification using self 859 organizing feature maps and machine activity data. *Computer Security,* 73, 399–410.

Elhadi, A. A. E., Maarof, M. A. & Barry, B. I. (2013). Improving the detection of malware be- 874 haviour using simplified data dependent API call graph. *International Journal Security Application,* 7 (5), 875 29–42.

Eslam, A. & Ivan, Z. (2018). A dynamic Windows malware detection and prediction method based on contextual understanding of API call sequence. *Computers & Security,* 30(40), 1-15.

Gandotra, E., Bansal, D. & Sofat, S. (2014). Malware analysis and classification: a survey. 885 *Journal of Information Security,* 5 (02), 56.

Gibert, D., Mateu, C. & Planes, J. (2020). The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. *Journal of Network and Computer Applications*, 153(2020), 1-22, 2020.

Karbab, E. B., Debbabi, M., Derhab, A. & Mouheb, D. (2018). MalDozer: Automatic framework for android malware detection using deep learning, *Digital Investigation* 24, 548-559.

Kim, T., Kang, B., Rho, M., Sezer, S. & Gyu, E. (2019). A Multimodal Deep Learning Method for Android Malware Detection using Various Features, in *IEEE Transactions on Information Forensic and Security*, 10(3), 773-778.

Li, J., Sunk, L., Yan, Q., Zhiqiang, L. Srisaan, W. & Heng, Y. (2018). "Significant Permission Identification for Machine Learning Based Android Malware Detection", *in IEEE Transactions on Industrial Informatics*, 14(7), 3216-3225.

Mario, L., Marta, C., Damiano, D., Fabio, M. & Francesco, M. (2019). Dynamic malware detection and phylogeny analysis using process mining. *International Journal of Information Security*, 18, 257–284.

McLaughlin, N. Rincon, J., Kang, B., Yerima, S., Miller, P., Sezer, S., Safaei, Y., Trickel, E., Zhao, Z., Doupe, A, & Ahn, G. (2017). Deep Android Malware Detection, *Proceeding on the Seventh ACM on Conference on Data and Application Security and Privacy*, 301-308.

Nighat, U., Saeeda, U., Fazlullah, K., Mian, A., Ahthasham S., Mamoun A., Paul W. (2021). Intelligent Dynamic Malware Detection using Machine Learning in IP Reputation for Forensics Data Analytics. *Future Generation Computer Systems*118 (2021), 124–141.

Pengbin, F., Jianfeng M., Cong S., Xinpeng X. & Yuwan M. (2018). A Novel Dynamic Android Malware Detection System with Ensemble Learning. *IEEE Access,* 6, 30996-31011.

Qiao, Y., Yang, Y., He, J., Tang, C. & Liu, Z. (2014). CBM: free, automatic malware anal- 923 ysis framework using API call sequences. In: Knowledge Engineering and Man- 924 agreement. Springer, Berlin, *Heidelberg,* 225–236.

Rieck, K., Holz, T., Willems, C., Dussel, P. & Laskov, P. (2008). Learning and classification of malware behavior, in DIMVA '08: Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment. *Berlin, Heidelberg: Springer-Verlag*, 108–125.

Souri, A. & Hosseini, R. (2018). A state-of-the-art survey of malware detection approaches using data mining techniques, Human. Centric. *Computing and Information Sciences,* 1-22.

Vinayakumar, A., Alazab, M., Soman, M., Poornachandran, P. & Venkatraman, S. (2019). "Robust Intelligent Malware Detection Using Deep Learning" *In IEEE Access*, 7, 46717-46738.

Vinayakumar, M., Alazab, K., Soman, P. & Poornachandran, S. (2019). Venkatraman "Robust Intelligent Malware Detection Using Deep Learning" *In IEEE Access*, (7), 46717-46738.

Yanfang, Y. (2017). A Survey on Malware Detection Using Data Mining Techniques, *ACM Computing Surveys,* 50.