Application of Machine Learning in Cyber Security

I. ABSTRACT

Cybersecurity has become a critical concern in the current digital age.With increasing use of technology and the internet a new type has also emerged where cyber attacks use the knowledge of these networks to breach the privacy of users and steal important information. Traditional methods of detecting and preventing cyber threats are proving inadequate with the increased frequency and complexity of cyberattacks.
Machine learning (ML) has emerged as a promising approach for enhancing cybersecurity by enabling automated, proactive threat detection and prevention. In this paper, we provide an overview of the use and importance of ML for cyber threat prediction and prevention in an "Autonomous Vehicle System" with in-vehicle networks and sensors controlling it.
We also present a small-scale simulation of a string injection attack, designed to demonstrate the nature of these attacks and provide a basis for further research into mitigation strategies. The simulation involves a client-server model, in which the server generates random messages and sends them to the client, which randomly modifies them with a specified probability.
Overall, this paper provides a comprehensive overview of the use and importance of ML for cyber threat prediction and prevention, highlighting its potential as a key tool for enhancing cybersecurity in the modern digital landscape.

II. INTRODUCTION

ECU and CAN messages

The advancement of innovations like sensors, automatic object recognition and tracking, communication between interconnected devices, distributed and integrated Internet services has led to a surge in the usage of smart devices in daily activities in recent years. Nowadays, modern automobiles have already become a significant part of our lives. Modern automobiles are gradually changing from all-mechanical control technology to software control technology due to the continuous development of network communication technology . With the combination of network, sensors and automotive technology, more and more modern cars are controlled based on electronic control units (ECUs). The electrical control systems of a contemporary automobile are connected by millions of lines of code, making it a sophisticated piece of technology (ECUs). The primary method of correspondence between ECUs is the Controller Area Network (CAN) bus protocol. However, CAN protocol lacks security features. Such in-vehicle networks do not consider potential security issues, making vehicles vulnerable to suffering attacks which might lead to property damage and life threat. It is simple for attackers to collect drivers' private information or even take control of the target vehicle if we cannot defend our in-vehicle networks against attacks.

The proposed model is validated on a representative standard Internet of Vehicles (IoV) dataset Car-Hacking Dataset. In the experiment, the proposed model achieves ~96% accuracy, which demonstrates that the proposed model detects the effectiveness of network intrusions.

Threats Associated With Sensors

Autonomous vehicles use sensors to determine the condition of the environment because they do not have drivers to control them. The use of the sensors is maximized, and if they are maliciously controlled or blinded, the vehicle could cause a catastrophic accident.

1) Autonomous vehicles use GPS to identify the shortest route by analyzing a map, but the GPS system creates vulnerabilities in autonomous cars that attackers can exploit. Hackers can conduct
    - Spoofing is when fabricated signals outwit the GPS receiver, and as a result, the vehicle can be dragged in the wrong direction.
    - Privacy Breach as attackers can take control over the map and know the exact location of the user and his frequent locations of interest or travel.
2) Autonomous vehicles use LiDAR sensors to detect obstacles in their path. Fuzzy attacks modify the data obtained from the vehicle's sensors, making the vehicle believe there are no obstacles, potentially causing a collision.

String Injection Simulation

String injection attacks are a form of cyber-attack that involve injecting malicious code or data into an application or system by manipulating strings. String injection attacks can have serious consequences, ranging from data theft and unauthorized access to complete system compromise. In order to better understand the nature of string injection attacks and develop effective mitigation strategies, it is important to simulate these attacks in a controlled environment. In this paper, we present a small-scale simulation of a string injection attack, designed to provide insight into the mechanics of these attacks and provide a framework for further research.

Dataset Description

The dataset provided by HCRL consists of four sets of data, each representative of one of the four types of attacks - DoS (Denial of Service), Fuzzy, RPM, Spoofing Gear.

Attack Type	**% Injected Messages**
DoS Attack	16.03%
Fuzzy Attack	12.81%
Gear Spoofing	13.44%
RPM Spoofing	14.17%

The experimental dataset used for training the models has been sampled from the combination of all four datasets. The resulting dataset has ~2 million data rows. The features corresponding to the data rows are as shown below:

Field	**Description of CAN Message**
Timestamp	recorded time (s)
CAN ID	identifier of CAN message in HEX (ex. 043f)
DLC	number of data bytes, from 0 to 8
DATA [0~7]	data value (byte)
Flag	T or R, where 'T' represents injected special attack messages while 'R' represents normal messages

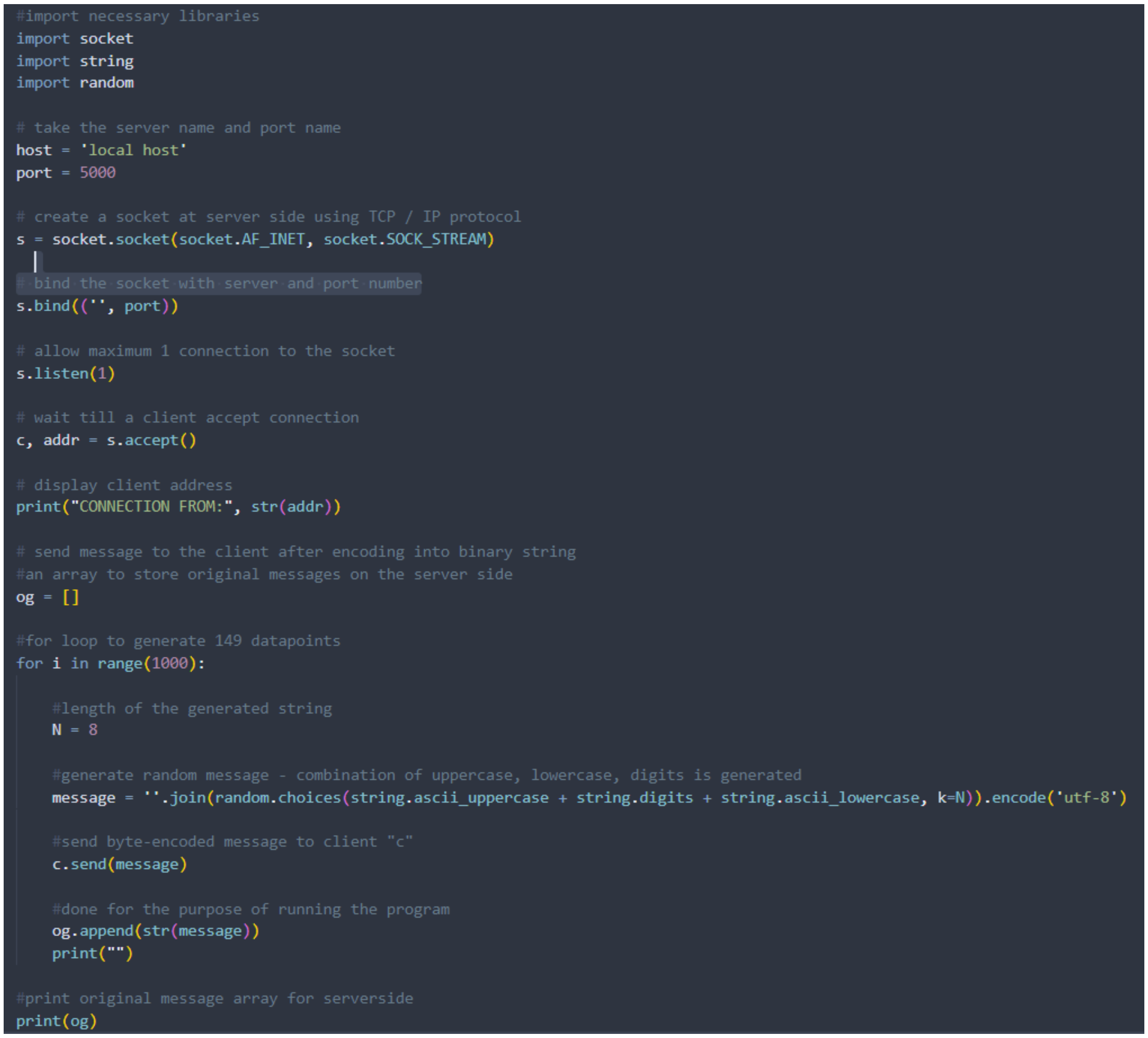
III. METHODOLOGY

Python Simulation

The simulation involves a client-server model, in which the server generates random messages and sends them to the client, which randomly modifies them with a specified probability. The server uses Python libraries to create a socket at the server-side using TCP/IP protocol and to bind it to a specified port number. The server listens to the socket and waits for a client to connect. Once a client connects, the server sends 1000 random messages to the client and stores the original messages in an array.

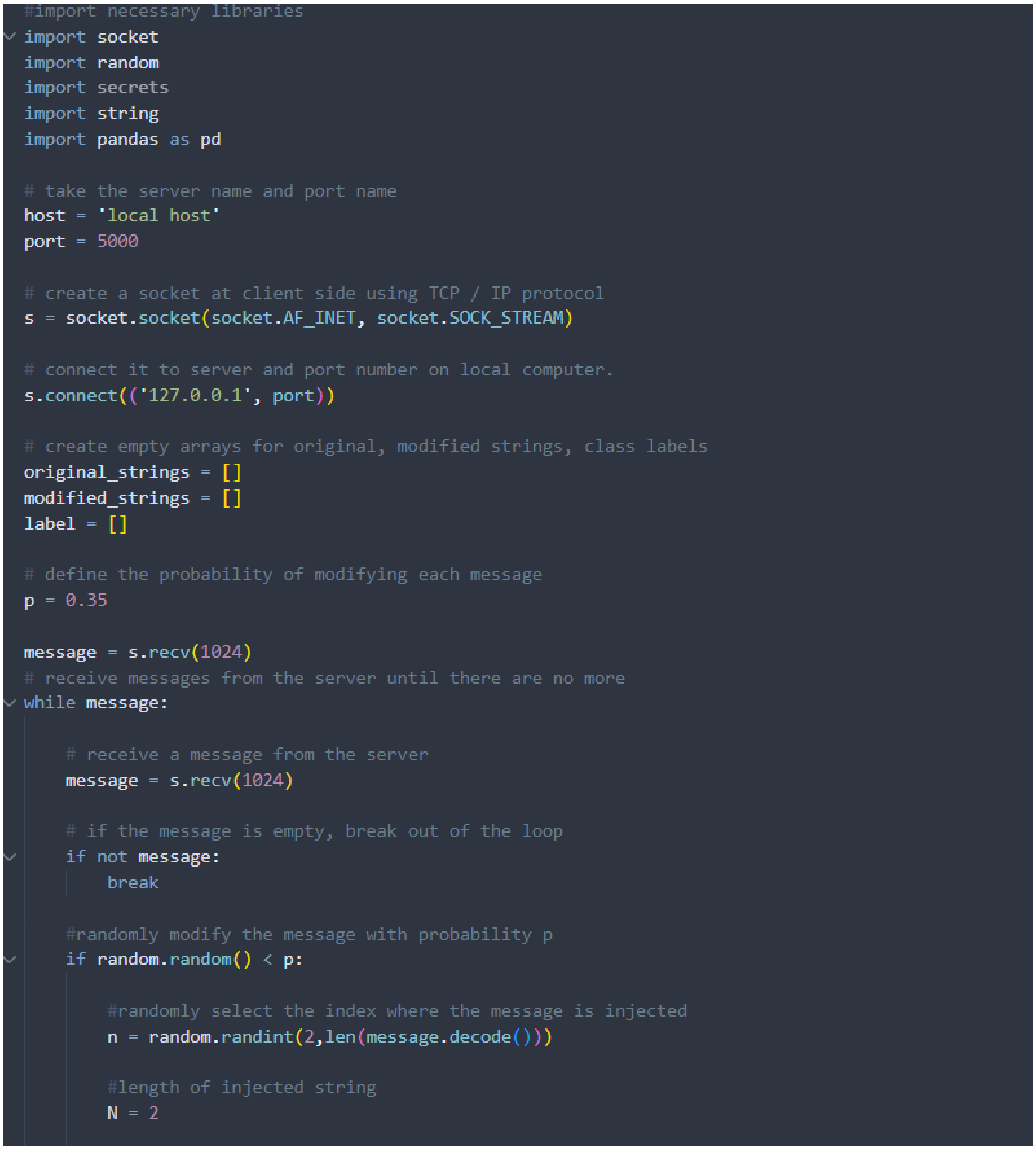
The client uses Python libraries to create a socket at the client-side using TCP/IP protocol and to connect to the server and specified port number. The client receives the random messages from the server and randomly modifies them with a specified probability. The original and modified messages are stored in separate arrays along with a class label indicating whether the message was modified or not. The resulting data is used to analyze the effectiveness of different security measures and to evaluate the impact of string injection attacks.

Server.py

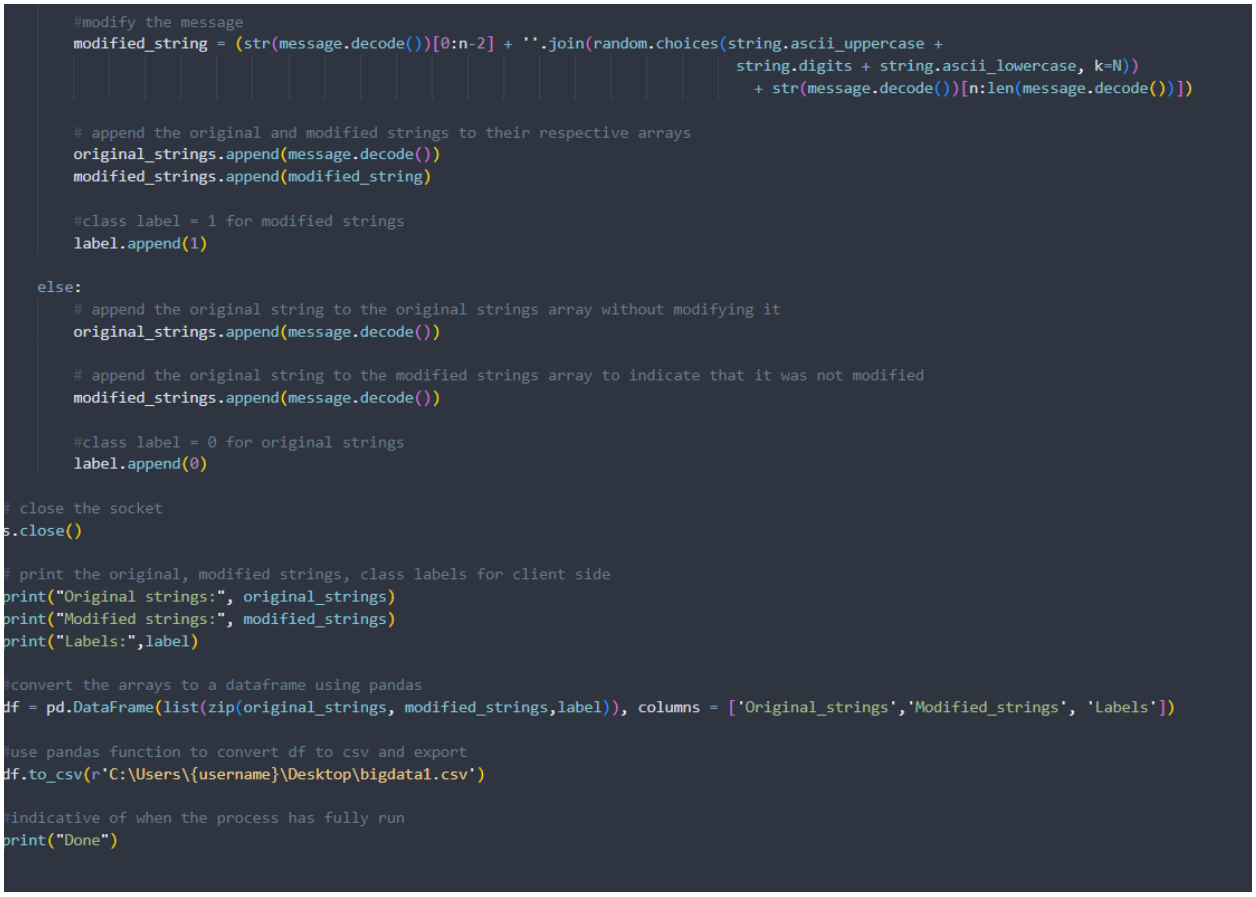


```python
#import necessary libraries
import socket
import string
import random

# take the server name and port name
host = 'local host'
port = 5000

# create a socket at server side using TCP / IP protocol
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# bind the socket with server and port number
s.bind(('', port))

# allow maximum 1 connection to the socket
s.listen(1)

# wait till a client accept connection
c, addr = s.accept()

# display client address
print("CONNECTION FROM:", str(addr))

# send message to the client after encoding into binary string
#an array to store original messages on the server side
og = []

#for loop to generate 149 datapoints
for i in range(1000):

    #length of the generated string
    N = 8

    #generate random message - combination of uppercase, lowercase, digits is generated
    message = ''.join(random.choices(string.ascii_uppercase + string.digits + string.ascii_lowercase, k=N)).encode('utf-8')

    #send byte-encoded message to client "c"
    c.send(message)

    #done for the purpose of running the program
    og.append(str(message))
    print("")

#print original message array for serverside
print(og)
```

Client.py



```python
#import necessary libraries
import socket
import random
import secrets
import string
import pandas as pd

# take the server name and port name
host = 'local host'
port = 5000

# create a socket at client side using TCP / IP protocol
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# connect it to server and port number on local computer.
s.connect(('127.0.0.1', port))

# create empty arrays for original, modified strings, class labels
original_strings = []
modified_strings = []
label = []

# define the probability of modifying each message
p = 0.35

message = s.recv(1024)
# receive messages from the server until there are no more
while message:

    # receive a message from the server
    message = s.recv(1024)

    # if the message is empty, break out of the loop
    if not message:
        break

    #randomly modify the message with probability p
    if random.random() < p:

        #randomly select the index where the message is injected
        n = random.randint(2,len(message.decode()))

        #length of injected string
        N = 2
```



```
        #modify the message
        modified_string = (str(message.decode())[0:n-2] + ''.join(random.choices(string.ascii_uppercase +
                                                                                   string.digits + string.ascii_lowercase, k=N))
                                                                                     + str(message.decode())[n:len(message.decode())])

        # append the original and modified strings to their respective arrays
        original_strings.append(message.decode())
        modified_strings.append(modified_string)

        #class label = 1 for modified strings
        label.append(1)

    else:
        # append the original string to the original strings array without modifying it
        original_strings.append(message.decode())

        # append the original string to the modified strings array to indicate that it was not modified
        modified_strings.append(message.decode())

        #class label = 0 for original strings
        label.append(0)

# close the socket
s.close()

# print the original, modified strings, class labels for client side
print("Original strings:", original_strings)
print("Modified strings:", modified_strings)
print("Labels:",label)

#convert the arrays to a dataframe using pandas
df = pd.DataFrame(list(zip(original_strings, modified_strings,label)), columns = ['Original_strings','Modified_strings', 'Labels'])

#use pandas function to convert df to csv and export
df.to_csv(r'C:\Users\{username}\Desktop\bigdata1.csv')

#indicative of when the process has fully run
print("Done")
```

Pipeline:

1) *Data Pre-processing of HCRL Dataset*

   Firstly, for each individual dataset, a “Type” column has been added that indicates whether the message is “Normal” or attack inflicted specified by the particular “Attack Name”. This is done on the basis of column “Flag” where R represents a normal message and T represents an attack inflicted message.

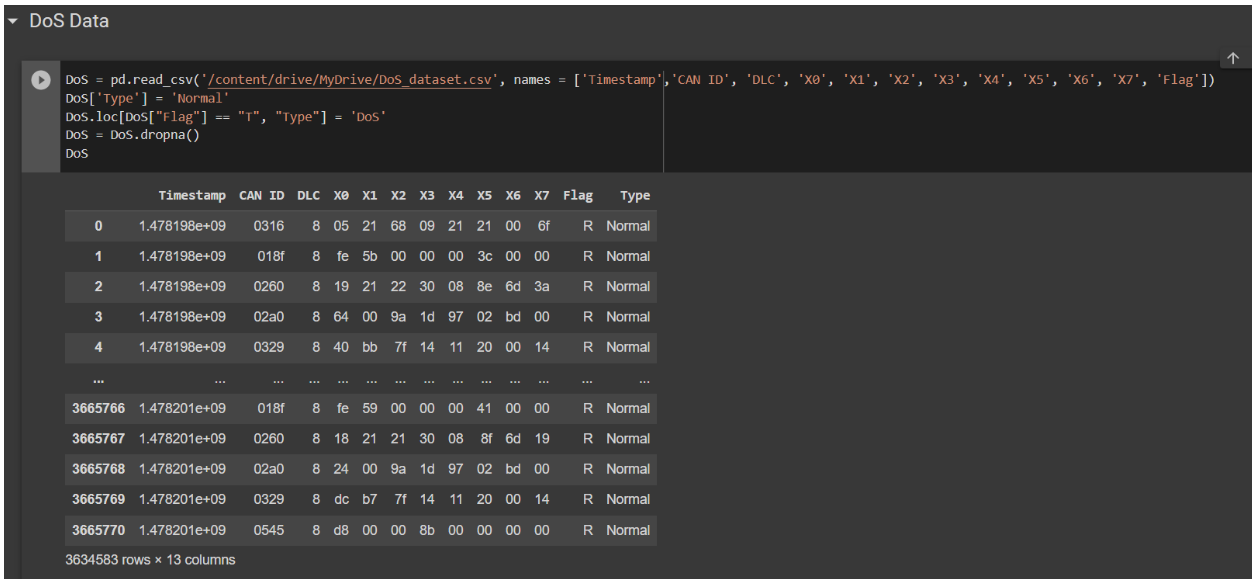
   These individual dataset are then concatenated into one large dataset. The missing values have been handled by dropping the data rows with null values. The resulting dataset is obtained by sampling around ~2 million data rows from the concatenated dataset.

   - The attack types have been label encoded:

     'Normal':0, 'DoS':1, 'Fuzzy':2, 'RPM':3, 'gear':4
   - Columns dropped before training the models:

     'Timestamp', 'CAN ID', 'DLC', 'Flag'
   - Byte data in features X[0~7] have been label encoded

DoS Dataset



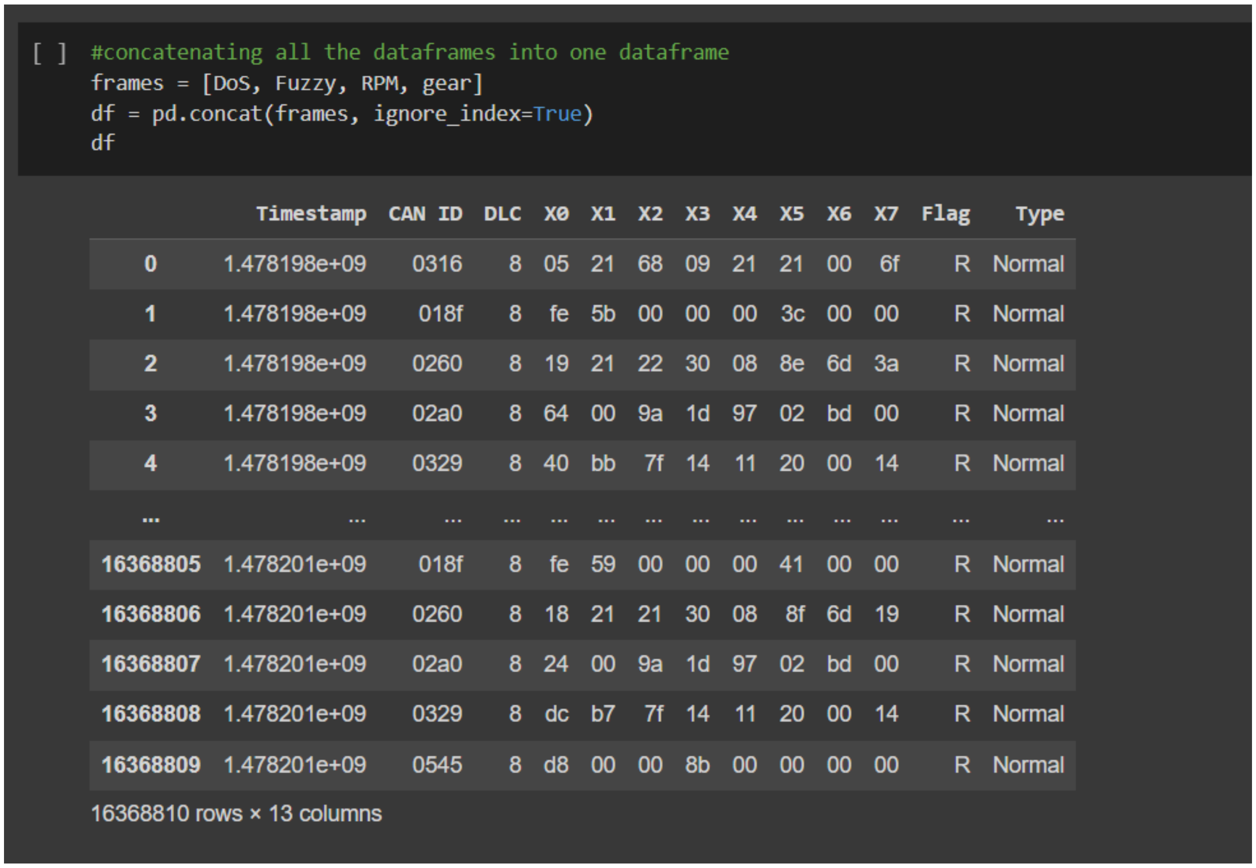
DoS Data

```
DoS = pd.read_csv('/content/drive/MyDrive/DoS_dataset.csv', names = ['Timestamp','CAN ID', 'DLC', 'X0', 'X1', 'X2', 'X3', 'X4', 'X5', 'X6', 'X7', 'Flag'])
DoS['Type'] = 'Normal'
DoS.loc[DoS["Flag"] == "T", "Type"] = 'DoS'
DoS = DoS.dropna()
DoS
```

```
             Timestamp CAN ID DLC  X0  X1  X2  X3  X4  X5  X6  X7 Flag    Type
      0   1.478198e+09   0316   8  05  21  68  09  21  21  00  6f    R  Normal
      1   1.478198e+09   018f   8  fe  5b  00  00  00  3c  00  00    R  Normal
      2   1.478198e+09   0260   8  19  21  22  30  08  8e  6d  3a    R  Normal
      3   1.478198e+09   02a0   8  64  00  9a  1d  97  02  bd  00    R  Normal
      4   1.478198e+09   0329   8  40  bb  7f  14  11  20  00  14    R  Normal
    ...            ...    ... ...  ... ... ... ... ... ... ... ...  ...     ...
3665766   1.478201e+09   018f   8  fe  59  00  00  00  41  00  00    R  Normal
3665767   1.478201e+09   0260   8  18  21  21  30  08  8f  6d  19    R  Normal
3665768   1.478201e+09   02a0   8  24  00  9a  1d  97  02  bd  00    R  Normal
3665769   1.478201e+09   0329   8  dc  b7  7f  14  11  20  00  14    R  Normal
3665770   1.478201e+09   0545   8  d8  00  00  8b  00  00  00  00    R  Normal
3634583 rows × 13 columns
```

The above procedure has been followed with the Fuzzy, RPM and gear datasets as well.

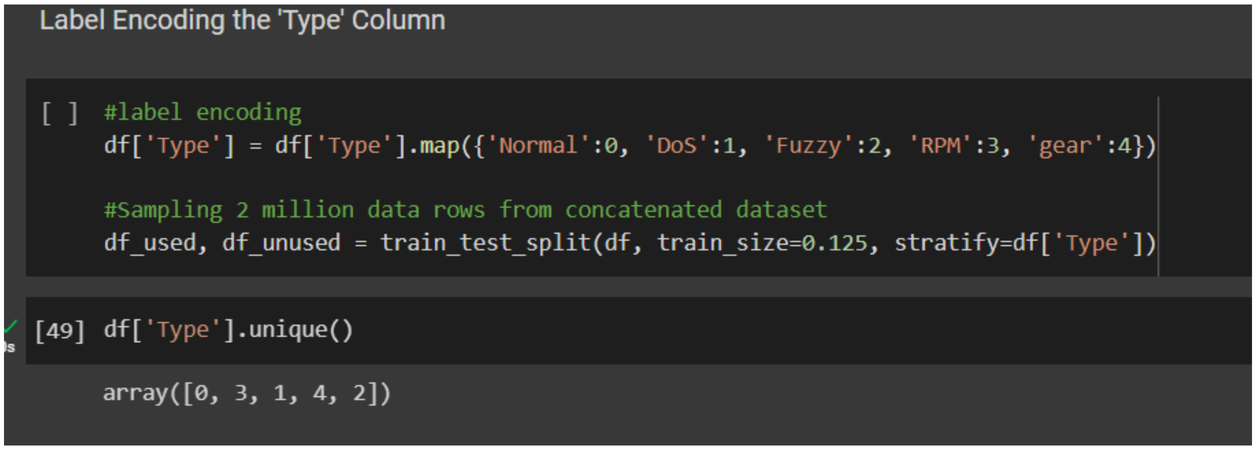
Concatenation



```
[ ] #concatenating all the dataframes into one dataframe
    frames = [DoS, Fuzzy, RPM, gear]
    df = pd.concat(frames, ignore_index=True)
    df
```

```
              Timestamp CAN ID DLC  X0  X1  X2  X3  X4  X5  X6  X7 Flag    Type
       0   1.478198e+09   0316   8  05  21  68  09  21  21  00  6f    R  Normal
       1   1.478198e+09   018f   8  fe  5b  00  00  00  3c  00  00    R  Normal
       2   1.478198e+09   0260   8  19  21  22  30  08  8e  6d  3a    R  Normal
       3   1.478198e+09   02a0   8  64  00  9a  1d  97  02  bd  00    R  Normal
       4   1.478198e+09   0329   8  40  bb  7f  14  11  20  00  14    R  Normal
     ...            ...    ... ...  ... ... ... ... ... ... ... ...  ...     ...
16368805   1.478201e+09   018f   8  fe  59  00  00  00  41  00  00    R  Normal
16368806   1.478201e+09   0260   8  18  21  21  30  08  8f  6d  19    R  Normal
16368807   1.478201e+09   02a0   8  24  00  9a  1d  97  02  bd  00    R  Normal
16368808   1.478201e+09   0329   8  dc  b7  7f  14  11  20  00  14    R  Normal
16368809   1.478201e+09   0545   8  d8  00  00  8b  00  00  00  00    R  Normal
16368810 rows × 13 columns
```

Label Encoding

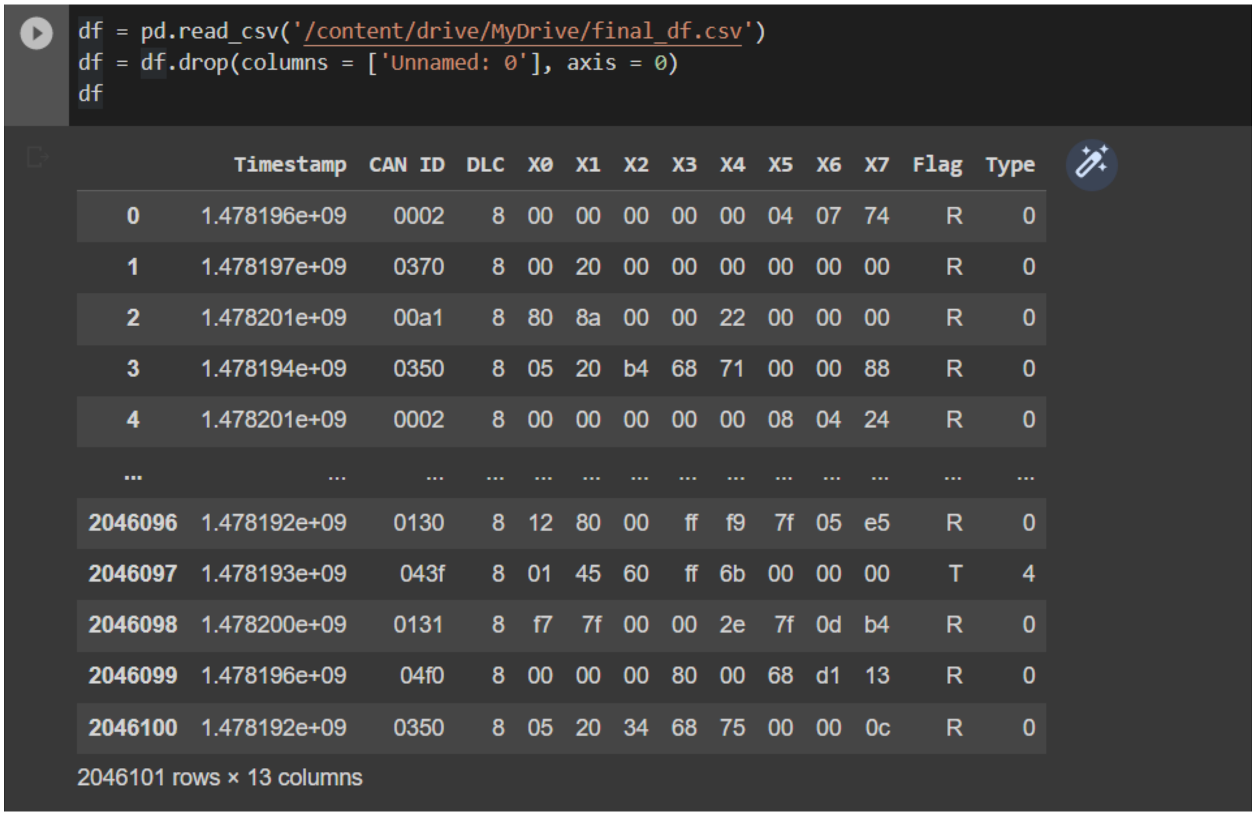


Label Encoding the 'Type' Column

```
[ ]  #label encoding
     df['Type'] = df['Type'].map({'Normal':0, 'DoS':1, 'Fuzzy':2, 'RPM':3, 'gear':4})

     #Sampling 2 million data rows from concatenated dataset
     df_used, df_unused = train_test_split(df, train_size=0.125, stratify=df['Type'])
```

```
[49] df['Type'].unique()

     array([0, 3, 1, 4, 2])
```

Resultant dataframe after pre processing



```
df = pd.read_csv('/content/drive/MyDrive/final_df.csv')
df = df.drop(columns = ['Unnamed: 0'], axis = 0)
df
```

	Timestamp	CAN ID	DLC	X0	X1	X2	X3	X4	X5	X6	X7	Flag	Type
0	1.478196e+09	0002	8	00	00	00	00	00	04	07	74	R	0
1	1.478197e+09	0370	8	00	20	00	00	00	00	00	00	R	0
2	1.478201e+09	00a1	8	80	8a	00	00	22	00	00	00	R	0
3	1.478194e+09	0350	8	05	20	b4	68	71	00	00	88	R	0
4	1.478201e+09	0002	8	00	00	00	00	00	08	04	24	R	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...
2046096	1.478192e+09	0130	8	12	80	00	ff	f9	7f	05	e5	R	0
2046097	1.478193e+09	043f	8	01	45	60	ff	6b	00	00	00	T	4
2046098	1.478200e+09	0131	8	f7	7f	00	00	2e	7f	0d	b4	R	0
2046099	1.478196e+09	04f0	8	00	00	00	80	00	68	d1	13	R	0
2046100	1.478192e+09	0350	8	05	20	34	68	75	00	00	0c	R	0

2046101 rows × 13 columns

The composition of the resultant dataset is as shown:

Message Type	**Number of messages**
Normal	1,754,661
DoS Attack	73,440
Fuzzy Attack	61,481
Gear Spoofing	81,862
RPM Spoofing	74,657
Total	**2,042,101**

Models

1) *Decision Tree Classifier*
    - max_depth: a hyperparameter used to control the maximum depth of the decision tree. In the code, it is set to 11.
    - Criterion: a hyperparameter to decide when a node in the decision tree splits. The default value, which is 'gini' has been used.

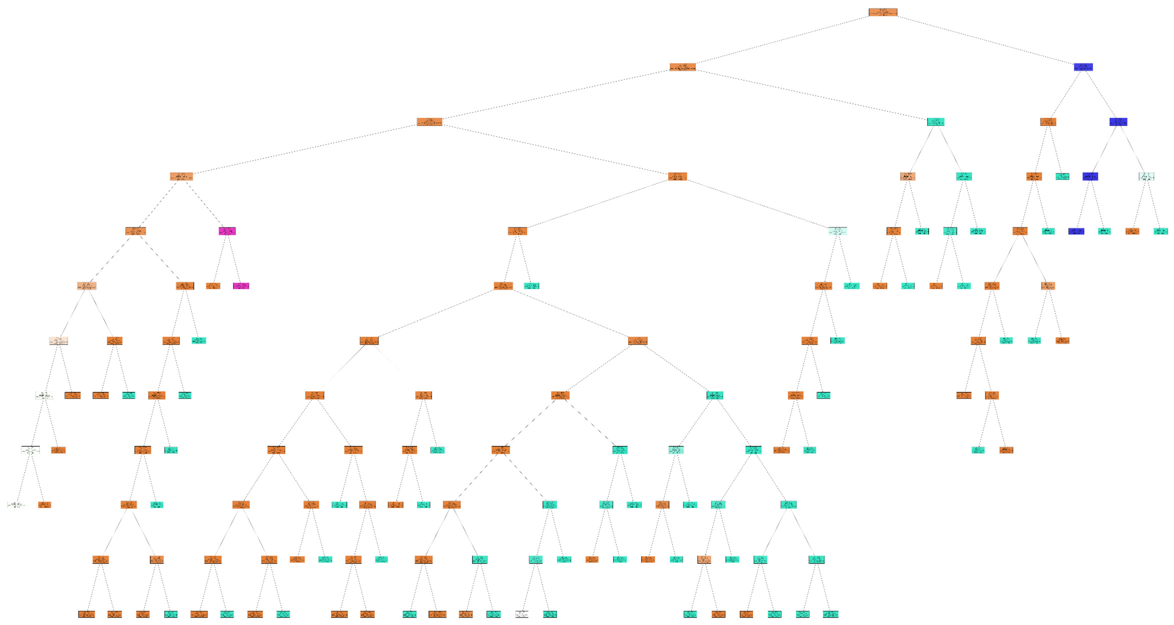
2) *Gaussian Naive Bayes Classifier*
    - No hyperparameter to be set explicitly

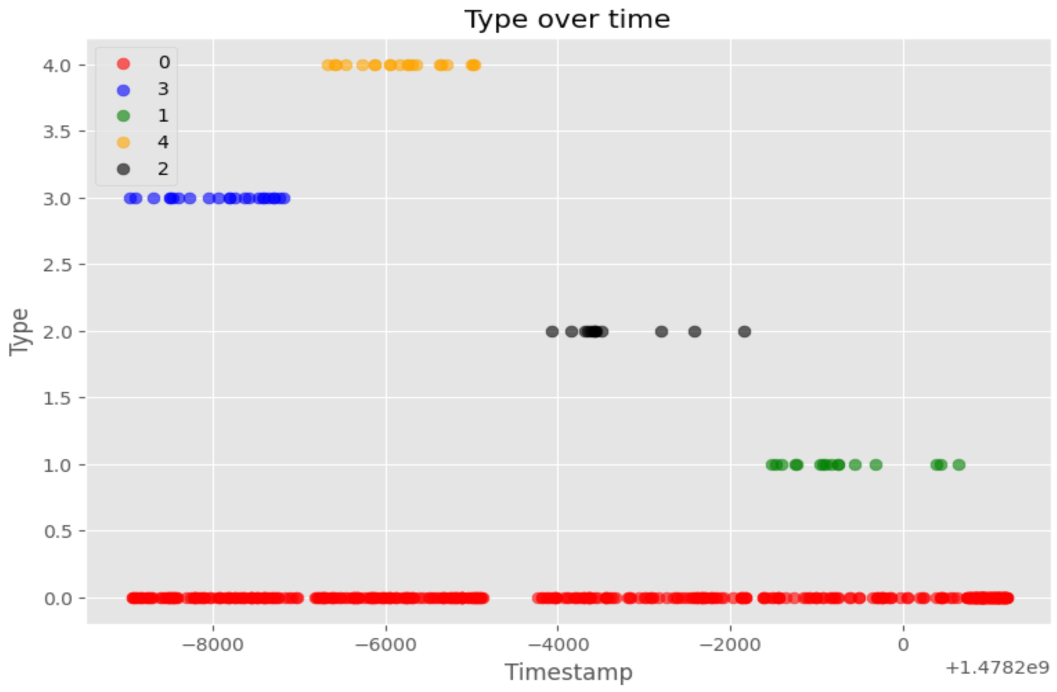
3) *Random Forest Classifier*
    - random_state: a random seed 42 used to ensure reproducibility of the results.
    - n_estimators: the number of decision trees to be used in the random forest. In this code, it is set to 20.
    - max_depth: the maximum depth of each decision tree in the random forest. In this code, it is set to 11.

4) *AdaBoost Classifier*
    - base_estimator: a decision tree classifier is used as the default base estimator.
    - n_estimators: The maximum number of estimators at which boosting is terminated. In the code, it is set to 20.
    - learning_rate: The learning rate shrinks the contribution of each classifier by the specified amount. In the code, the default value of 1.0 is used.
    - algorithm: The algorithm to use for the boosting process. Possible values are "SAMME" and "SAMME.R". In this code, the default value of "SAMME.R" is used.

IV. EVALUATION

Decision Tree Plot





Classification Reports for various models

MODEL	Class	Precision	RECALL	F1 SCORE
Decision Tree Classifier	0	1.00	0.96	0.98
	1	0.51	1.00	0.67
	2	1.00	0.99	0.99
	3	1.00	1.00	1.00
	4	1.00	1.00	1.00

Table 1: Classification report for Decision Tree Classifier

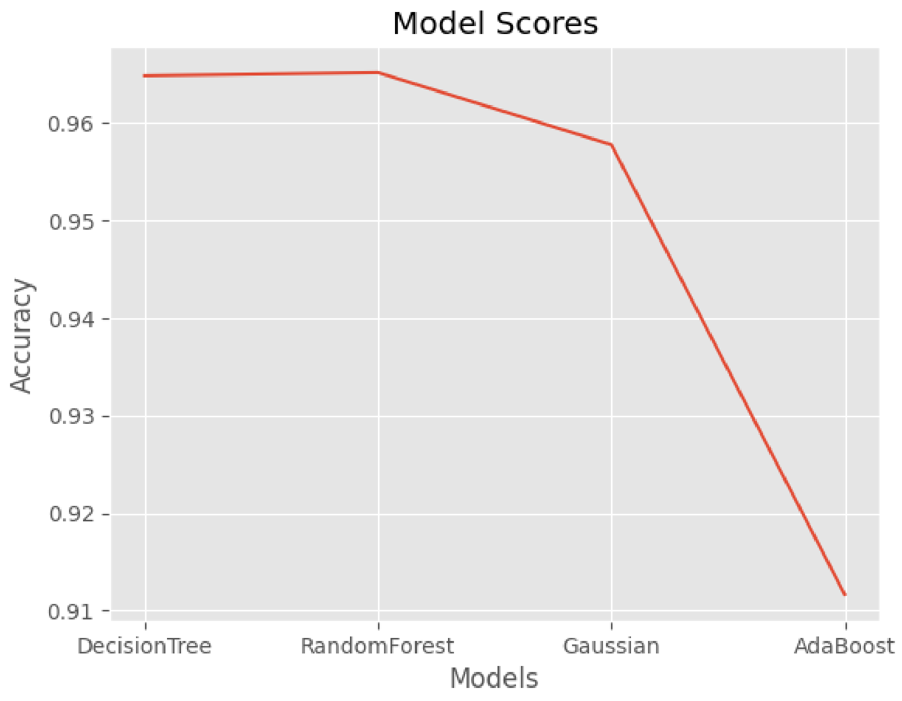
MODEL	Class	Precision	RECALL	F1 SCORE
Random Forest Classifier	0	1.00	0.96	0.98
	1	0.51	1.00	0.67
	2	1.00	1.00	1.00
	3	1.00	1.00	1.00
	4	1.00	1.00	1.00

Table 2: Classification report for Random Forest Classifier

MODEL	**Class**	**Precision**	**RECALL**	**F1 SCORE**
Gaussian Naive Bayes Classifier	0	0.96	0.99	0.97
	1	1.00	0.51	0.67
	2	0.76	1.00	0.86
	3	1.00	1.00	1.00
	4	1.00	1.00	1.00

MODEL	**Class**	**Precision**	**RECALL**	**F1 SCORE**
AdaBoost Classifier	0	0.95	0.95	0.95
	1	0.00	0.00	0.00
	2	0.34	0.82	0.48
	3	1.00	1.00	1.00
	4	1.00	1.00	1.00

MODEL	**Accuracy**
Decision tree Classifier	96.48%
Random Forest Classifier	96.51%
Gaussian Naive Bayes	95.77%
AdaBoost Classifier	91.16%



Observation

As observed from the table above, decision tree classifier and random forest classifier compete closely with both of them showing high accuracies and f1-scores of ~96.5% on the test set. The AdaBoost classifier with Decision Tree as the base classifier reports an accuracy of ~91.16% which is comparatively lower than the base decision tree classifier and the decision tree ensemble (random forest classifier). All models report high precision, recall and f1-scores for most class labels, barring the scores of AdaBoost for ‘class 1’ attack.

V. CONCLUSION

In conclusion, the current digital age has necessitated the use of machine learning in cybersecurity. With cyber threats becoming increasingly complex and frequent, traditional methods of detection and prevention are no longer sufficient. Machine learning offers a promising approach for enhancing cybersecurity by enabling automated, proactive threat detection and prevention. Our focus in this paper has been on the use of ML for cyber threat prediction and prevention in an Autonomous Vehicle System, which is a critical application of ML in cybersecurity. Our experiments have demonstrated that the proposed ML model is effective in detecting network intrusions with a high degree of accuracy, representing an important step forward in the development of automated cybersecurity systems. Our findings could have important implications for other fields, such as critical infrastructure protection and national security. Therefore, the use of machine learning in cybersecurity will continue to be an important area of research and development in the years to come.

VI. REFERENCES

[1]https://mdpi-res.com/d_attachment/sensors/sensors-22-01340/article_deploy/sensors-22-01340.pdf
[2]https://eprint.iacr.org/2022/1700.pdf
[3]https://dl.acm.org/doi/pdf/10.1145/3570954

VII. GROUP MEMBERS

Om Kashyap (B21MT026)
Prashant Tandon (B21AI053)
Shah Keval Bhushanbhai (B21CS069)
Kush Agrawal (B21ME034)
Sushant Chivale (B21MT033)