NEW YORK INSTITUTE OF TECHNOLOGY

DTSC 870 - M01

Final Report: Adversarial Combat

A Deep Dive into Machine and Deep Learning Models Against Adversarial Attacks for Android Malware Detection

Professor Dr. Wenjia Li

Team Members

Zoya Haq	1222440	zhaq01@nyit.edu
Neelam Boywah	1226855	nboywah@nyit.edu
Selina Narain	1261565	snarai01@nyit.edu

Abstract

This research implements Android malware detection using static analysis. Static data analysis is used to investigate the structure and characteristics of Android applications. The analysis also includes looking into and preprocessing two main datasets, the University of New Brunswick: CICMaldroid 2020 and DADA: Debiased Android Datasets. Through the various applications of machine learning algorithms to the datasets, the project analyzes malicious behavior from the dataset samples. This includes algorithms like Random Forest, Naive Bayes Logistic Regression, KNN, and SVM (rbf, linear) models. There is also implementation of a deep learning algorithm specifically, the Dense Neural Network (DNN) model. Furthermore, the research evaluates an adversarial attack to observe the vulnerabilities of the models and develops an adversarial defense mechanism. The FGSM adversarial attack is explored and implemented on the highest performing model from our preliminary findings. Additionally, defense mechanisms are explored further to defend the model against the adversarial attack using adversarial training. This interdisciplinary approach hopes to contribute to the advancement of the fields of machine learning, deep learning and cybersecurity by offering insight for resilient Android Malware detection in an age of evolving threats.

Keywords: Static Data Analysis, Android Malware Detection, Machine Learning, Deep Learning, Adversarial Attacks, Defense Mechanisms

Table of Contents

1.0 Introduction 3

1.1 Motivation 3

1.2 Purpose 3

2.0 Literature Review 3

2.1 Static Analysis 3

2.2 Dynamic Analysis 4

2.3 Machine Learning Approach 5

2.4 Adversarial Attacks and Defense Mechanisms 5

2.6 How is our research different from existing research? 7

3.0 Research Project Timeline 7

3.1 Phase I - Proposal and Key Technologies 8

3.2 Phase II - Interim 8

3.3 Phase III - Final 9

4.0 Android Malware Detection Research Design 10

5.0 Implementation 11

5.1 Datasets 11

5.2 Machine Learning Models 13

5.3 Deep Learning Models 20

5.4 Adversarial Attacks 22

5.5 Defense Mechanisms 23

6.0 Experimentation & Testing 21

7.0 PowerBI Analytics Dashboard 27

8.0 Conclusion 29

8.1 Future Works 30

8.2 Team Contributions 30

9.0 References 31

Introduction

Motivation

The propagation of mobile technology, specifically the use of smartphones, has increased in the modern day and age. At the top of this age in mobile usage is the Android Operating System which had an immense share in the global market. Android has an open source nature as opposed to its counterparts like Apple's Operating System - iOS. The Android Operating system does allow for versatility with billions of users. In this era, users rely on mobile applications and devices to guide their lifestyles and drive daily routine. From communication to social interaction, financial responsibilities and productivity, these technological devices are heavily used. The Google Play Store alone hosts a plethora of applications that suits various needs. Though we are living in a mobile-focused world, there is a challenge which involves the increasing threat of malware that specifically targets Android devices.

Purpose

The developers and users for the Android ecosystem have attracted those involved in cyber crime in order to exploit various vulnerabilities ranging from financial fraud, identity theft and more. This malware can come in many different formats like ransomware, trojans and spyware. This is secure for device health and is a risk to one's privacy. The purpose for this project is to contribute to the cybersecurity field by exploring new methodologies, techniques and models in order to advance the understanding of evolving malicious threats and their respective countermeasures.

Literature Review

Static Analysis

Static analysis, in reference to malware detection within the Android environment, refers to the inspection of the Android apps code and various files without executing the application.It involves the examination of an app's structure to detect security violations and malicious behavior. With the aforementioned code inspection, one can check the permissions requested by the app itself to identify the potential security faults. Examining the API calls made by the app itself can also aid in detecting malicious patterns. Static analysis also involves the analysis of manifest files and by going through the manifest file we can recognize specific attributes like activities, services and receivers.

Drebin, is a method for detecting Android malware by using a comprehensive static analysis approach which involves extracting features from an application's code and manifest[1]. The static analysis facilitates effective detection by providing results as patterns indicative of malware detection and how it can be traced back from the vector space. This method incorporates lightweight analysis which enables the detection of malware on smartphones and the analysis of large sets of apps within a reasonable time frame.

Through statistically analyzing Android applications, Drebin extracts different features that are embedded in a vector space for geometric analysis and this allows malware identification through the use of Support Vector Machines (SVM). [1] Manifest features that are derived from .xml files provide important data that support its installation and execution. Drebin leverages the fact that Android apps developed in Java are compiled into bytecode for Dalvik VM and that provides information on the API calls and data. The system looks at the API calls and the Android permissions system then limits the access to the critical API's even though malware can still slip through and exploit the root access and navigate through the implemented restrictions. Drebin looks at the permissions that are needed and used by analyzing the permissions which are extracted from the restricted API calls. Suspicious API calls are then identified since they allow unauthorized access to sensitive data.

Dynamic Analysis

Dynamic analysis executes a malicious Android application in a controlled environment in order to observe real-time behavior [12]. This allows the user to have insight on how the app functions and allows detection of suspicious activity that static analysis alone may not be able to capture. One of the major components involved with dynamic analysis is behavioral monitoring: network activities file system access, and system calls. Network activities seek communication with unknown servers. The observation of file system interactions allows for the identification of unauthorized access, modification and file creation. The analysis of system calls help the user understand the behavior between the application and the operating system. Tools that are used in Dynamic analysis include TaintDroid and DroidScope and use schemes like AppsPlayground and DroidRanger for scanning and analysis purposes [11]. Subsequently, feature extraction is

performed which focuses on extracting risky permissions, identifying suspicious API calls, and capturing URLs that were accessed during execution. This set of features is then put into a classifier and uses Support Vector Machines to categorize and identify malicious behavior. This dynamic analysis schema has a multi-step involved process that enhances the ability to view app behavior and detest anomalies which contributes to robust malware detection mechanisms.

Machine Learning Approach

The Android malware detection landscape takes on various different approaches with a focus on SVM-based methodologies. A study uncovered a wide range of detection rates (20.2% to 79.6%) of popular security software tools [11]. There are several research efforts that contribute to the field like Drebin as aforementioned. SVM was employed to learn classifiers from existing ground truth datasets. Drebins efficacy is hindered by its time consuming nature due to the high dimensional feature vector involved in the process.

In the context of SVM based approaches, one strategy involves calculating similarity scores between malware and benign applications in terms of suspicious API calls. These similarity scores serve as features in the SVM algorithm and the risky permission requests are incorporated as additional features during the training of the SVM classifier. The results of this approach demonstrated 86% accuracy in detecting the combinations of the dangerous API calls and risky permissions [11].

Adversarial Attacks and Defense Mechanisms

Crafting realistic malicious attacks poses several constraints in order to ensure effectiveness and maintain malicious purposes. Primarily, it is important to preserve the malicious behaviors from the original attack ensuring that the modified malware maintains its intended impact [13]. Furthermore, mutated malware must exhibit robustness, enabling successful installation and execution on the mobile devices. To evade detection by malware detectors, attackers use strategies such as identifying features that are crucial for detection and computing values that enable evasion without compromising the malwares functionality.

In response to these adversarial challenges, defense mechanisms are devised to enhance the resilience of malware detection systems. Adversarial training involves the creation of a new model by combining generated malware variants with the original training data with the goal of improving the model's ability to recognize diverse attack patterns. The Variant Detector represents an additional defense layer which was put in place to see whether an app is a variant derived from existing malware and then providing a secondary line of defense that is beyond the original detector. Another defense strategy involves the restriction of feature weights in the original malware to establish an evenly distributed weight distribution. These defense mechanisms contribute to the fortification of malware detection systems against adversarial attacks.

Android malware detection is a binary classification problem where there are well trained classifiers to distinguish whether an app is malicious or not. There are features utilized for malware detection which fall into some of the aforementioned categories of static, dynamic or hybrid. The extraction of dynamic features requires real time mentoring. When distinguishing between the normal and adversarial examples the normal instances were found to compromise features extracted from benign or malicious Android apps while adversarial examples involve attackers extracting features from a malicious app and then combining them with others to mislead the detection system.

In order to fortify the system against the adversarial attacks a defense strategy is implemented which involves data collection, model training, and adversarial example detection. The detection system collects normal examples by extracting features from both benign and malicious applications and using techniques like Generative Adversarial Networks (GAN) for enhanced data diversity [7]. During model training, both the normal and adversarial examples are combined and binary classifiers are trained collectively. Trained classifiers then serve as firewalls to detect and block adversarial examples during the runtime.

On the adversarial front, the attacks are orchestrated with the preliminary step involving a game between the Generator and Discriminator within the GAN framework. The motivation for these attacks come from the challenging targeted classifiers, and the model and algorithm employed

often take the form of a bi-objective GAN and introduce a dynamic interplay between generating adversarial examples and enhancing the robustness of the classifiers.

In the paper, "*DroidEnemy: Battling adversarial example attacks for Android malware detectio*n", there were three different attack strategies that were explored in the context of Malware Detection in Android. The data poisoning attack, the exploratory attack, and the evasion attack. Two attack models were developed specifically for the data poisoning attack and the evasion attack, which is aimed at generating adversarial examples[9]. These attack models are focused on mutating multiple features of Android applications including API calls, permissions, and the class label with the goal of deceiving the classifier. The Support Vector Machine Algorithm (SVM) employing the kernel functions like linear and radial basis function (RBF), played an important role in the adversarial attack models. An experimental study was conducted through the use of a real Android application dataset to evaluate the attacks effectiveness. The results display significant impacts on the classifiers performance, specifically in the evasion attack where all the instances were not correctly classified with contrasts with the data poisoning attacks mixed outcome where 74.85% were correctly classified instances and 25.14% were incorrectly classified instances. These metrics highlight the importance of having robust defense mechanisms to safeguard against the adversarial attacks in the Android malware detection systems.

<u>How is our research different from existing research?</u>

In our research implementation, we utilized 6 different machine learning and deep learning models. We also used a Deep Neural Network (DNN) as opposed to a Convolutional Neural Network (CNN) for this project. The DNN model was implemented due to the characteristics of the data at hand. The datasets we utilized included sequential data such as the sequences of API calls and permissions. The DNN is designed to be better fit for implementation because of the powerful sequential data analysis. Though there are projects that use CNN models for malware detection, CNNs are more preferred for image-related tasks. Most research papers talk about an adversarial attack implementation and how they are able to deteriorate the accuracy of the model. In our research implementation, we were able to successfully implement an adversarial attack and a defense mechanism that brings back up the accuracy of the model.

Research Project Timeline

Phase I - Proposal and Key Technologies

Throughout the semester, we have conducted research to expand our knowledge on malware detection, adversarial attacks and defense mechanisms. With this research we have obtained we were able to take the necessary steps to implement and test machine learning and deep learning models against datasets that contain both benign and malicious data. For our testing and implementation, we used many tools and technologies. Visual Studio Code and Google Colaboratory were utilized as our Integrated Development Environment (IDE) and we used python as our programming language to build our code. For environmental control and package management, we utilized anaconda. Some of the packages and libraries that were utilized in our implementation were Sci-kit Learn, NumPy, Pandas, Tensorflow, Seaborn, and Matplotlib. For a visualization report that we created, we used Power BI. The two datasets that are utilized in our implementation and testing are the CICMalDroid 2020 dataset and the DADA Dataset.

Phase II - Interim

Throughout our interim phase, we performed various implementations in order to obtain different findings. Firstly, we identified our datasets, specifically the CICMaldroid2020. We also wanted another dataset to have a comparison, so we had requested access to AndroZoo. We obtained the AndroZoo API access and we wrote 3 scripts in order to begin obtaining the APKs, decompiling the APKs, and extracting the permissions from the manifest.xml files. We were able to successfully download the APKs and decompile some of them using apktool; however, some of the APK files ended up being corrupt and unusable. We were also unable to extract the permissions from the manifest.xml files as the permissions were not being converted to the csv file properly.

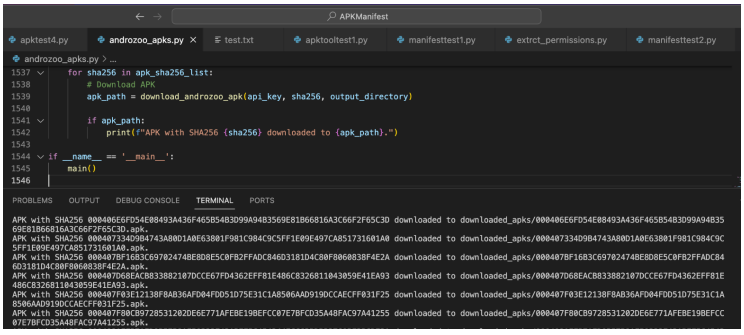
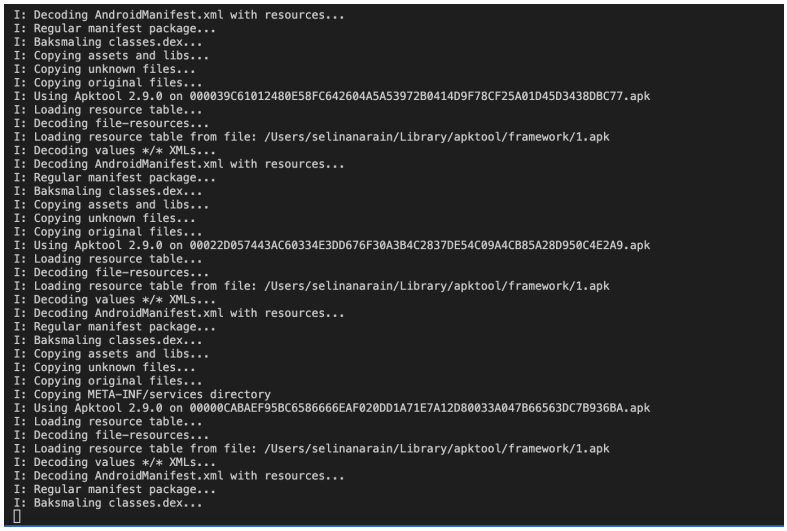


Figure 1: Downloading AndroZoo APKs



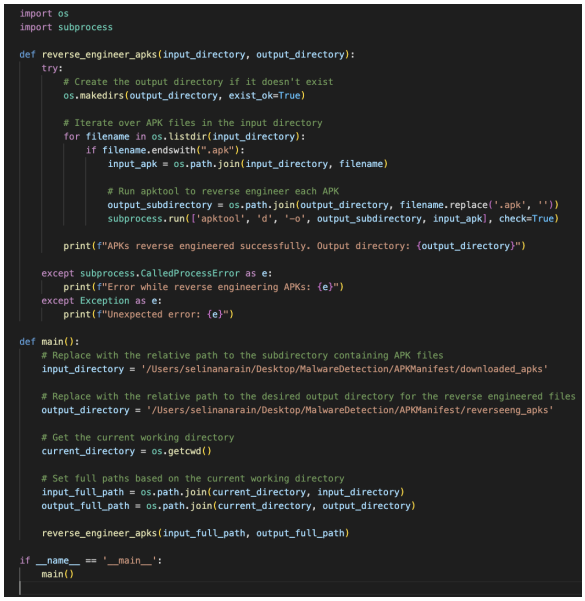


Figure 2: Reverse Engineering/Decompilation of APK Files

Due to the time constraints, we decided to use a different dataset, DADA: Debiased Android Datasets. This dataset provides debiased data in terms of malware and benign samples from Drebin, AMD, AndroZoo, and more. Once we obtained this dataset, we decided to use it along with CICMaldroid to conduct our machine learning and deep learning models. We then implemented the FGSM adversarial attack and the adversarial training defense mechanism on the models and further evaluated the metrics of all of our generated models.

Phase III - Final

In the final phase of our research project, we were able to successfully detect malware in our datasets based on the performance of the machine learning and deep learning models. The six machine learning models that we built and tested against our dataset were Naive Bayes, Random Forest, Logistic Regression, KNN, SVM (kernel - ‘rbf’), and SVM (kernel - ‘linear’). The deep learning model that we used in our implementation was a Dense Neural Network (DNN) model. After building these models, we were then able to capture their evaluation metrics and use them for analysis in our research. After seeing which model had the highest performance, which was the Random Forest model in both datasets, we then applied an adversarial attack and defense mechanism on it to see how it would affect the performance of the model.

Android Malware Detection Research Design

Our design is broken down into five main stages.

(1) Preprocessing Data Phase 1

This stage includes preparing and cleaning our data. We had to collect the data from both New Brunswick and DADA. Additionally, we were also processing data from AndroZoo's API but there were complications with the extraction process.

(2) Preprocessing Data Phase 2

This stage entailed identifying duplicates, null values, splitting the data, and selecting features. We further standardized and transformed the data.

(3) Implementation and Testing

This stage consisted of applying our machine learning models on our preprocessed data. This included both multiclass classification and binary classification.

(4) Adversarial Attack and Defense Mechanism

This stage implemented our adversarial attack on our highest performing model. We then created a defense mechanism for the performed adversarial attack.

(5) Evaluation Metrics

Our evaluation metrics consisted of understanding our metrics on how our models performed as well as how robust the adversarial attack and defense mechanisms performed.

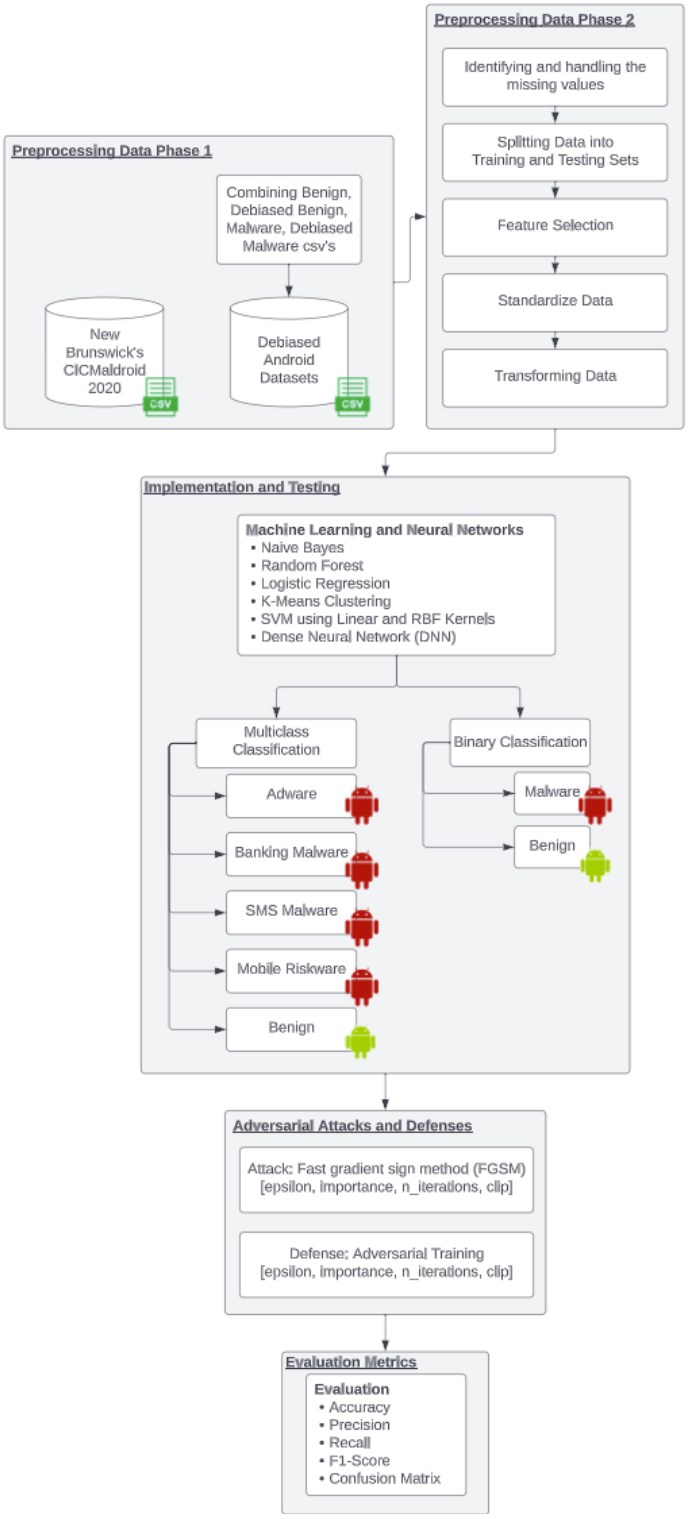


Figure 3: Flowchart for Research Design

Implementation

Datasets

University of New Brunswick CICMaldroid 2020

CICMalDroid 2020 is an android malware dataset that is based on over 17,341 Android samples capturing static and dynamic features. This data is sourced from various sources such as the VirusTotal service, Contagio security blog, AMD, MalDozer, and more. The data is broken down into five categories: Adware, Banking malware, SMS malware, Riskware, and Benign. Adware is based on advertising material such as advertisements that usually hides inside the legitimate apps that have been infected. In terms of Banking Malware, it is based on specialized malware that is designed to obtain access to users' online banking accounts. It does so by mimicking the original banking applications or banking web interface. SMS Malware entails malware that exploit SMS services. It does so by using these services as a medium of operation to intercept SMS payload to execute attacks. Mobile Riskware refers to executables that have the potential to cause damage if malicious attackers exploit them. Benign refers to applications that do not contain malware which serves as a control in our experimentation. Overall in the dataset, there samples are as follows: Adware: 1,253, Banking: 2,100, SMS malware: 3,904, Riskware: 2,546, Benign: 1,795 which totals 11,598 samples.

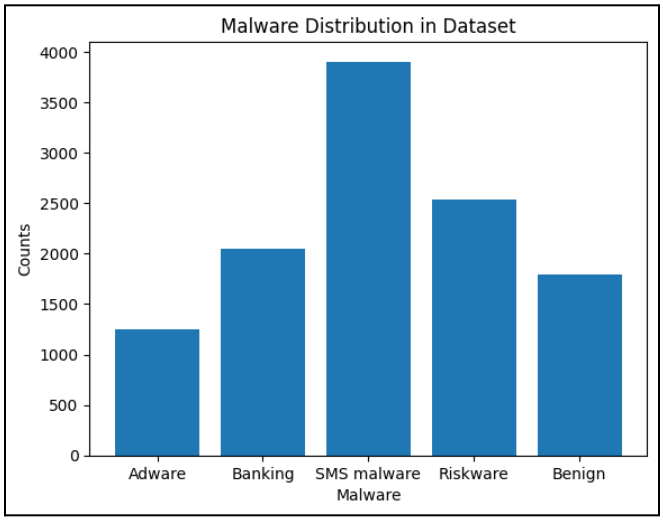
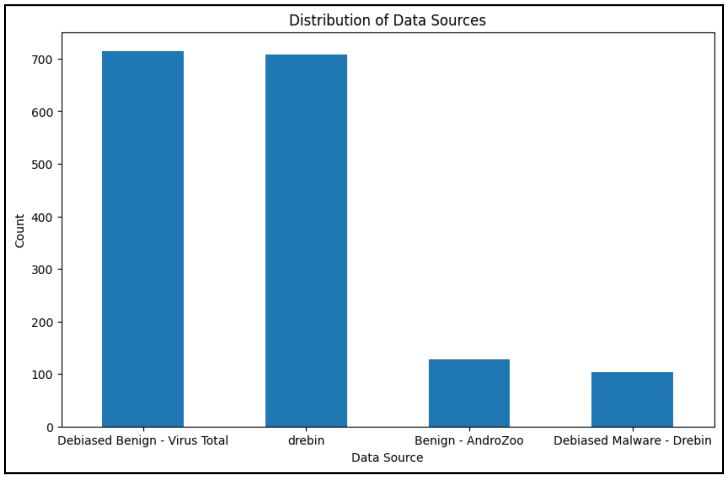


Figure 4: Malware Distribution of CICMaldroid 2020 Dataset

DADA: Debiased Android Datasets - "Debiasing Android Malware Datasets: How can I trust your results if your dataset is biased?"

This dataset is based on previous research conducted on existing datasets that have been used in Android malware scanning experiments. Based on their research, they have found that many existing data from AndroZoo, Drebin, AMD, and other sources are not completely representative of the entire current population of Android applications. Therefore, they performed debiasing through choosing characteristics of highest significance. They have produced debiased datasets containing samples that are malware and benign which can be used to train machine learning models on a debiased model. For our implementation, we used the produced mixed dataset, DR-AG$_{Deb}$, which represents debiased processed data from Drebin and AndroZoo. In this dataset, 0 is considered benign and 1 is considered malware.



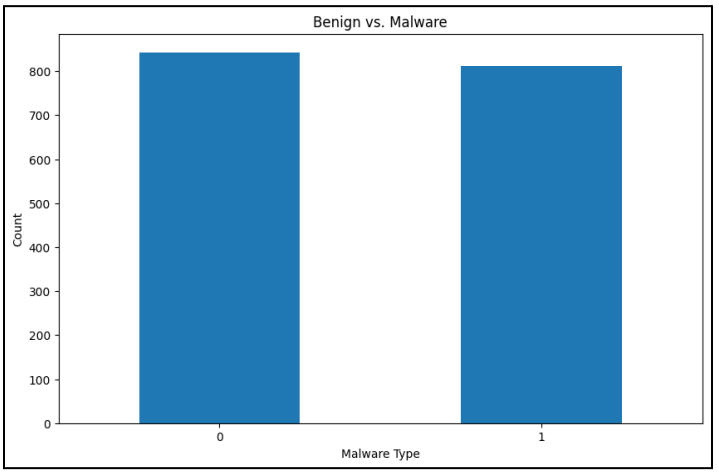


Figure 5: Data Distribution of DADA Dataset

For both of these datasets, we split the data into 80-20 training and testing sets. We then implemented feature selection, standardization, and transformation of the datasets to be processed for our machine learning and deep learning models. We dropped features from both datasets and used Sklearn's standard scaler to center and scale each feature by computing the relevant statistics on the samples in the training set.

```
#Features
X = df.drop(columns=['Class'])
#Target
y = df['Class']

#Splitting data into training and test
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify = y, test_size=0.2, random_state=42)

#ANOVA-based feature selection
selected_features = 120
k_best_features = SelectKBest(score_func = f_classif, k = selected_features)
X_train_selected_features = k_best_features.fit_transform(X_train, y_train)
X_test_selected_features = k_best_features.transform(X_test)

#Indicies of selected features
selected_features_idx = k_best_features.get_support(indices=True)

#Converting selected features to a DataFrame
X_train = X_train.iloc[:, selected_features_idx]
X_test = X_test.iloc[:, selected_features_idx]
X_train.head()
```

✓ 0.2s

	CREATE_FOLDER_____	CREATE_PROCESS`_____	CREATE_THREAD_____	EXECUTE_____	FS_ACCESS____
9652	2	0	10	0	14
11116	12	2	26	5	44
8165	0	0	9	0	6
6913	6	0	40	0	42
6682	7	0	45	0	52

5 rows × 120 columns

```
# Scale the data using Standard Scaler function from SKLearn
scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Figure 6: CICMalDroid 2020 Dataset - Preprocessing Data

```
#Features
X = df.drop(columns = ['APK size', 'Year', 'year', 'dex_date', 'dataset', 'sha256', 'malware'])
#Target
y = df['malware']

#Split the data into training and test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 42)

X_train.head()
```

	Internet Permission	External storage	Uses Play Services	Generates UUIDs	Vibrate phone	NFC	Bluetooth	Uses HTTP	Uses JSON	Specify User-Agent	...	zerg
1301	1	1	0	1	0	0	0	1	0	0	...	0
306	1	1	1	0	0	0	0	0	0	0	...	0
192	1	1	1	0	0	0	0	0	0	0	...	0
309	1	1	1	0	0	0	0	0	0	0	...	0
1168	1	0	0	0	0	0	0	0	1	1	...	0

5 rows × 215 columns

```
# Scale the data using Standard Scaler function from SKLearn
scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Figure 7: DADA Dataset - Preprocessing Data

As the CICMalDroid 2020 has multiple classes (Adware, Banking malware, SMS malware, Riskware, and Benign), multiclass classification would be performed as opposed to the DADA Dataset which only has two classes (Malware, Benign) and would have binary classification performed.

Machine Learning Models

After preprocessing the datasets, six machine learning models were implemented using supervised machine learning algorithms for our testing. The machine learning models that were implemented on the CICMaldroid 2020 Dataset and DADA Dataset include Naive Bayes, Random Forest, Logistic Regression, K-Nearest Neighbors (KNN), SVM with kernels, ‘rbf’ and ‘linear’.

```python
#Create Random Forest Model
random_forest_model = RandomForestClassifier(n_estimators=300, random_state=42)

#Fit and train classifier on the training data
random_forest_model.fit(X_train, y_train)

#Predict the labels using X_test
y_pred_rf = random_forest_model.predict(X_test)

#Calculate the metrics for the Random Forest Classifier
random_forest_accuracy = accuracy_score(y_test, y_pred_rf)
random_forest_precision = precision_score(y_test, y_pred_rf,average='weighted')
random_forest_recall = recall_score(y_test, y_pred_rf,average='weighted')
random_forest_f1 = f1_score(y_test, y_pred_rf,average='weighted')

#Display the metrics for the Random Forest Classifier
print("Random Forest Classifier Accuracy:  %.4f" % (random_forest_accuracy))
print("Random Forest Classifier Precision: %.4f" % (random_forest_precision))
print("Random Forest Classifier Recall: %.4f" % (random_forest_recall))
print("Random Forest Classifier F1-Score: %.4f" % (random_forest_f1))
print("Classification Report: " "\n", classification_report(y_test, y_pred_rf))
print("Confusion Matrix: ", "\n", confusion_matrix(y_test, y_pred_rf))

#Create and display heatmap
conf_matrix = confusion_matrix(y_pred_rf, y_test)
sns.heatmap(conf_matrix , cmap='Blues', fmt='',
            xticklabels=['Adware' ,'Banking' ,'SMS malware', 'Riskware','Benign'],
            yticklabels=['Adware' ,'Banking' ,'SMS malware', 'Riskware','Benign'], annot=True)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix - Random Forest Classifier')
plt.show()
```

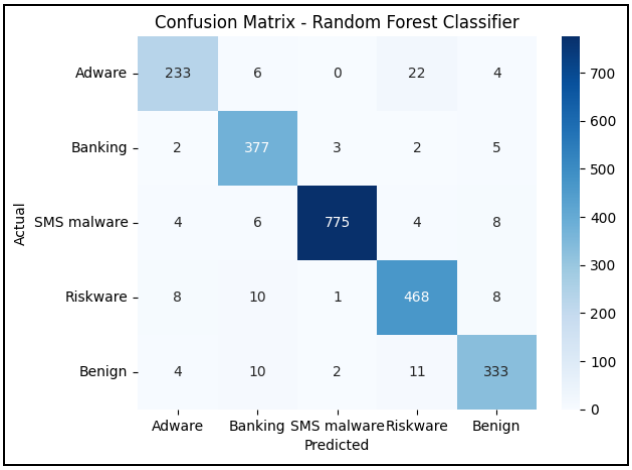


Figure 8: Random Forest Model - CICMaldroid 2020 Dataset

```python
#Create Random Forest Model
random_forest_model = RandomForestClassifier(n_estimators = 300, random_state = 42)

#Fit and train classifier on the training data
random_forest_model.fit(X_train, y_train)

#Predict the labels using X_test
y_pred_rf = random_forest_model.predict(X_test)

#Calculate the metrics for the Random Forest Classifier
randomForestAccuracy = accuracy_score(y_test, y_pred_rf)
randomForestPrecision = precision_score(y_test, y_pred_rf,average = 'weighted')
randomForestRecall = recall_score(y_test, y_pred_rf,average = 'weighted')
randomForestF1 = f1_score(y_test, y_pred_rf,average = 'weighted')

#Display the metrics for the Random Forest Classifier
print("Random Forest Classifier Accuracy: %.4f" % randomForestAccuracy)
print("Random Forest Classifier Precision: %.4f" % randomForestPrecision)
print("Random Forest Classifier Recall: %.4f" % randomForestRecall)
print("Random Forest Classifier F1-Score: %.4f" % randomForestF1)
print("Classification Report: " "\n", classification_report(y_test, y_pred_rf))
print("Confusion Matrix: ", "\n", confusion_matrix(y_test, y_pred_rf))

#Create and display heatmap
conf_matrix = confusion_matrix(y_test, y_pred_rf, labels = classes)
sns.heatmap(conf_matrix , cmap = 'Blues', fmt = 'd', annot = True)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix - Random Forest Classifier')
plt.show()
```

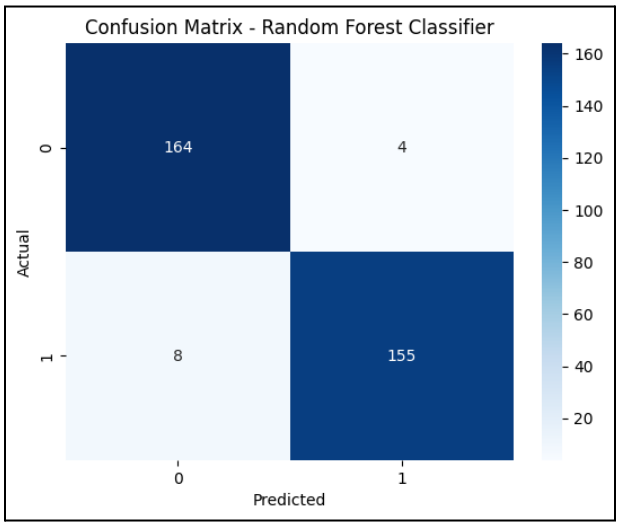


Figure 9: Random Forest - DADA Dataset

The Random Forest model consists of a set of decision trees classifiers where each one is trained from a data sample and is averaged. In our implementation we built the model using parameters including n_estimators = 300 and random_state = 42. The model was then fit, trained and predicted on X_test. We were then able to capture the evaluation metrics, classification report, confusion matrix and heatmap. According to Figure 8, we can see that 233 Adware features, 377 Banking features, 775 SMS malware features, 468 Riskware features and 333 Benign features were classified correctly in the CICMaldroid 2020 Dataset followed by a minority of features that were misclassified. For the DADA dataset in Figure 9, we can see that 164 benign features and 155 malware features were classified correctly and only 12 features were misclassified.

```python
#Create Naive Bayes Model
naive_bayes_model = GaussianNB()

#Fit X_train and y_train into the model
naive_bayes_model.fit(X_train, y_train)

#Predict the labels using X_test
y_pred_nb = naive_bayes_model.predict(X_test)

#Calculate the metrics for the Naive Bayes Classifier
naive_bayes_accuracy = accuracy_score(y_test, y_pred_nb)
naive_bayes_precision = precision_score(y_test, y_pred_nb,average='weighted')
naive_bayes_recall = recall_score(y_test, y_pred_nb,average='weighted')
naive_bayes_f1 = f1_score(y_test, y_pred_nb,average='weighted')

#Display the metrics for the Naive Bayes Classifier
print("Naive Bayes Classifier Accuracy: %.4f" % (naive_bayes_accuracy))
print("Naive Bayes Classifier Precision: %.4f" % (naive_bayes_precision))
print("Naive Bayes Classifier Recall: %.4f" % (naive_bayes_recall))
print("Naive Bayes Classifier F1-Score: %.4f" % (naive_bayes_f1))
print("Classification Report: " "\n", classification_report(y_test, y_pred_nb))
print("Confusion Matrix: ", "\n", confusion_matrix(y_test, y_pred_nb))

#Create and display heat map
conf_matrix = confusion_matrix(y_pred_nb, y_test)
sns.heatmap(conf_matrix , cmap='Blues', fmt='',
            xticklabels=['Adware' ,'Banking' ,'SMS malware', 'Riskware','Benign'],
            yticklabels=['Adware' ,'Banking' ,'SMS malware', 'Riskware','Benign'], annot=True)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix - Naive Bayes Classifier')
plt.show()
```

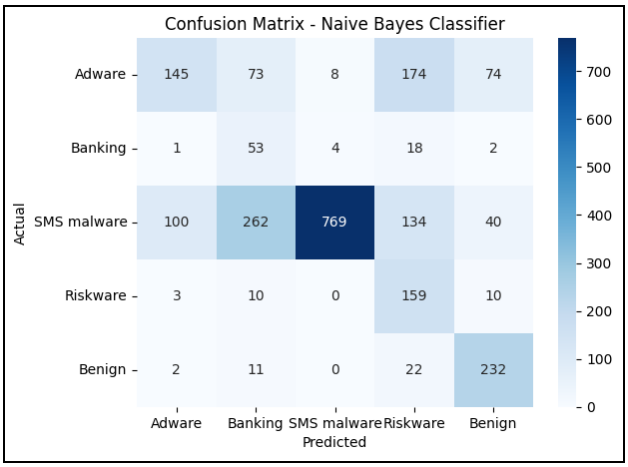


Figure 10: Naive Bayes Model - CICMaldroid 2020 Dataset

```python
#Create Naive Bayes Model
naiveBayesModel = GaussianNB()

#Fit X_train and y_train into the model
naiveBayesModel.fit(X_train, y_train)

#Predict the labels using X_test
naivebayes_y_pred = naiveBayesModel.predict(X_test)

#Calculate the metrics for the Naive Bayes Classifier
naiveBayesAccuracy = accuracy_score(y_test, naivebayes_y_pred)
naiveBayesPrecision = precision_score(y_test, naivebayes_y_pred,average = 'weighted')
naiveBayesRecall = recall_score(y_test, naivebayes_y_pred,average = 'weighted')
naiveBayesf1 = f1_score(y_test, naivebayes_y_pred,average = 'weighted')

#Display the metrics for the Naive Bayes Classifier
print("Naive Bayes Classifier Accuracy: %.4f" % (naiveBayesAccuracy))
print("Naive Bayes Classifier Precision: %.4f" % (naiveBayesPrecision))
print("Naive Bayes Classifier Recall: %.4f" % (naiveBayesRecall))
print("Naive Bayes Classifier F1-Score: %.4f" % (naiveBayesf1))
print("Classification Report: " "\n", classification_report(y_test, naivebayes_y_pred))
print("Confusion Matrix: ", "\n", confusion_matrix(y_test, naivebayes_y_pred))

#Create and display heatmap
conf_matrix = confusion_matrix(y_test, naivebayes_y_pred, labels = classes)
sns.heatmap(conf_matrix , cmap = 'Blues', fmt = 'd', annot = True)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix - Naive Bayes Classifier')
plt.show()
```

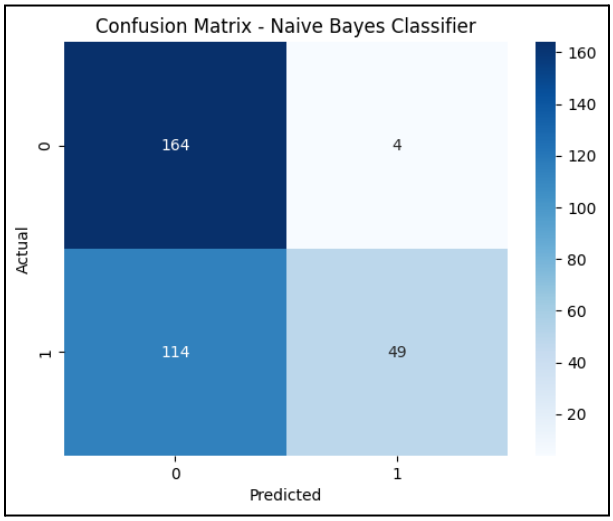


Figure 11: Naive Bayes Model - DADA Dataset

The Naive Bayes Model is based on the Bayes' theorem that assumes the conditional independence between each pair of features and the class variable value. We fit and trained the models using X_train, y_train and then predicted the labels using X_test. Once doing so, the evaluation metrics were obtained along with the classification report, confusion matrix and heatmap. Based on our evaluation metrics and confusion matrix heatmap, we realized that the model did not do a great job in classifying the features correctly. For example in Figure 11, the heatmap shows that 164 benign features and 49 features were classified correctly but there were still a total of 118 features that were misclassified.

```python
#Create Logistic Regression Model
log_reg_model = LogisticRegression(max_iter=1000, random_state=42)

#Fit and train classifier on the training data
log_reg_model.fit(X_train, y_train)

#Predict the labels using X_test
y_pred_log_reg = log_reg_model.predict(X_test)

#Calculate the metrics for the Logistic Regression Classifier
log_reg_accuracy = accuracy_score(y_test, y_pred_log_reg)
log_reg_precision = precision_score(y_test, y_pred_log_reg,average='weighted')
log_reg_recall = recall_score(y_test, y_pred_log_reg,average='weighted')
log_reg_f1 = f1_score(y_test, y_pred_log_reg,average='weighted')

#Display the metrics for the Logistic Regression Classifier
print("Logistic Regression Accuracy: %.4f" % (log_reg_accuracy))
print("Logistic Regression Precision: %.4f" % (log_reg_precision))
print("Logistic Regression Recall: %.4f" % (log_reg_recall))
print("Logistic Regression F1-Score: %.4f" % (log_reg_f1))
print("Classification Report: " "\n", classification_report(y_test, y_pred_log_reg))
print("Confusion Matrix: ", "\n", confusion_matrix(y_test, y_pred_log_reg))

#Create and display heatmap
conf_matrix = confusion_matrix(y_pred_log_reg, y_test)
sns.heatmap(conf_matrix , cmap='Blues', fmt='',
            xticklabels=['Adware' ,'Banking' ,'SMS malware', 'Riskware','Benign'],
            yticklabels=['Adware' ,'Banking' ,'SMS malware', 'Riskware','Benign'], annot=True)
plt.title('Confusion Matrix - Logistic Regression Model')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()
```

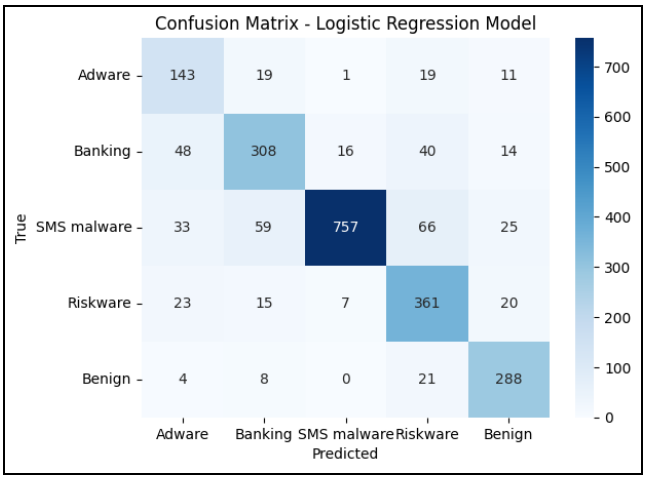


Figure 12: Logistic Regression - CICMaldoid 2020 Dataset

```python
#Create Logistic Regression Model
logisticRegressionModel = LogisticRegression(max_iter = 1000, random_state = 42)

#Fit and train classifier on the training data
logisticRegressionModel.fit(X_train, y_train)

#Predict the labels using X_test
logisticregression_y_pred = logisticRegressionModel.predict(X_test)

#Calculate the metrics for the Logistic Regression Classifier
logisticRegressionAccuracy = accuracy_score(y_test, logisticregression_y_pred)
logisticRegressionPrecision = precision_score(y_test, logisticregression_y_pred,average = 'weighted')
logisticRegressionRecall = recall_score(y_test, logisticregression_y_pred,average = 'weighted')
logisticRegressionF1 = f1_score(y_test, logisticregression_y_pred,average = 'weighted')

#Display the metrics for the Logistic Regression Classifier
print("Logistic Regression Accuracy: %.4f" % (logisticRegressionAccuracy))
print("Logistic Regression Precision: %.4f" % (logisticRegressionAccuracy))
print("Logistic Regression Recall: %.4f" % (logisticRegressionAccuracy))
print("Logistic Regression F1-Score: %.4f" % (logisticRegressionAccuracy))
print("Classification Report: " "\n", classification_report(y_test, logisticregression_y_pred))
print("Confusion Matrix: ", "\n", confusion_matrix(y_test, logisticregression_y_pred))

#Create and display heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(confusion_matrix(y_test, logisticregression_y_pred), annot=True, fmt='d', cmap='Blues',
            xticklabels=classes, yticklabels=classes)
plt.title('Confusion Matrix - Logistic Regression Model')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()
```

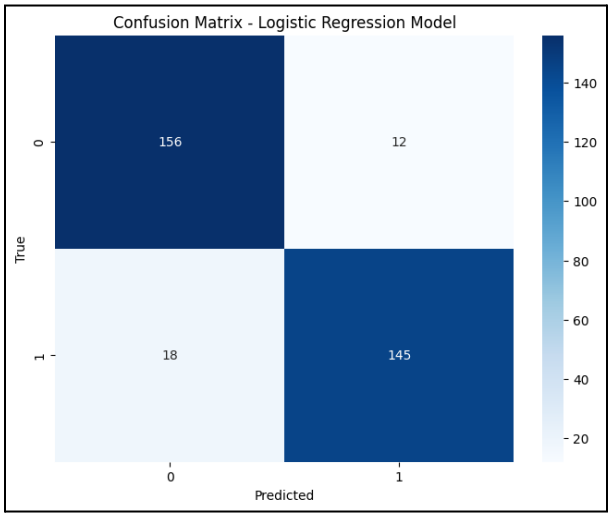


Figure 13: Logistic Regression - DADA Dataset

The Logistic Regression Model helps to predict and differentiate between classes based on specific feature variables. For this model, the parameters we used for our implementation included max_iter = 1000 and random_state = 42. We then fit, trained and predicted the model to obtain its evaluation metrics, classification report, confusion matrix and heatmap. In Figure 12 of the CICMaldroid 2020 Dataset, we can see that there were 143 Adware features, 308 Banking features, 757 SMS malware features, 361 Riskware features and 288 Benign features that were classified correctly followed by a minority of features that were misclassified. According to the DADA Dataset in Figure 13, 156 benign features and 145 malware features were classified correctly and about 30 features were still misclassified.

```python
#Range of K values
K_values = [2, 3, 4, 5, 7, 9, 10, 11]

K_best = None
accuracy_best = 0

#Create KNN Model using K_values for the value of n_neighbors
for k in K_values:
    KNN_model = KNeighborsClassifier(n_neighbors=k)
    #perform cross-validation to evaluate the KNN model
    scores = cross_val_score(KNN_model, X_train, y_train, cv=5, scoring='accuracy')
    mean_accuracy = scores.mean()
    if mean_accuracy > accuracy_best:
        accuracy_best = mean_accuracy
        K_best = k

#Training the KNN model with the best k value
model_KNN_best = KNeighborsClassifier(n_neighbors = K_best)
model_KNN_best.fit(X_train, y_train)

#Predict the labels using X_test
y_pred_KNN = model_KNN_best.predict(X_test)

#Calculate the metrics for the KNN Classifier
KNN_accuracy = accuracy_score(y_test, y_pred_KNN)
KNN_precision = precision_score(y_test, y_pred_KNN, average='weighted')
KNN_recall = recall_score(y_test, y_pred_KNN, average='weighted')
KNN_f1 = f1_score(y_test, y_pred_KNN, average='weighted')

#Display the metrics for the KNN Classifier
print("Best K value:", K_best)
print("K-Nearest Neighbors Classifier Accuracy: %.4f" % (KNN_accuracy))
print("K-Nearest Neighbors Classifier Precision: %.4f" % (KNN_precision))
print("K-Nearest Neighbors Classifier Recall: %.4f" % (KNN_recall))
print("K-Nearest Neighbors Classifier F1-Score: %.4f" % (KNN_f1))
print("Classification Report: " "\n", classification_report(y_test,y_pred_KNN))
print("Confusion Matrix: ", "\n", confusion_matrix(y_test, y_pred_KNN))

#Create and display heatmap
conf_matrix = confusion_matrix(y_pred_KNN, y_test)
sns.heatmap(conf_matrix, cmap='Blues', fmt='',
            xticklabels=['Adware' ,'Banking' ,'SMS malware', 'Riskware','Benign'],
            yticklabels=['Adware' ,'Banking' ,'SMS malware', 'Riskware','Benign'], annot=True)
plt.title('Confusion Matrix - KNN Classifier')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()
```

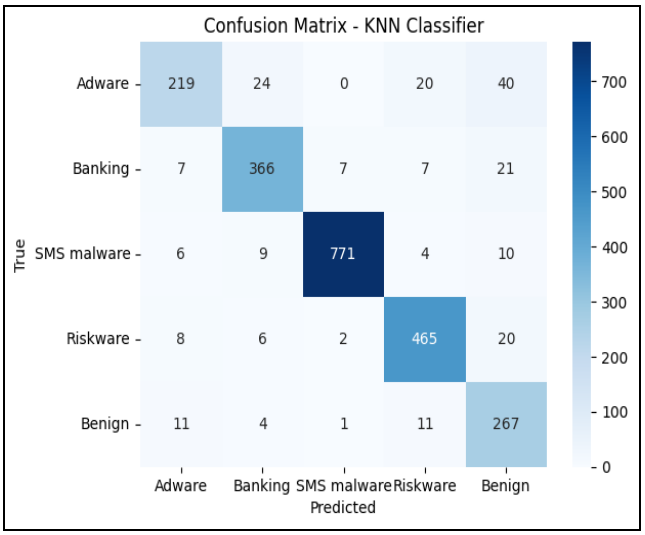


Figure 14: KNN Model - CICMaldroid Dataset

```python
#range of K values
K_values = [2, 3, 4, 5, 7, 9, 10, 11]

K_best = None
accuracy_best = 0

#Create KNN Model using K_values for the value of n_neighbors
for k in K_values:
    KNN_model = KNeighborsClassifier(n_neighbors = k)
    #perform cross-validation to evaluate the KNN model
    scores = cross_val_score(KNN_model, X_train, y_train, cv = 5, scoring = 'accuracy')
    mean_accuracy = scores.mean()
    if mean_accuracy > accuracy_best:
        accuracy_best = mean_accuracy
        K_best = k

#Training the KNN model with the best k value
model_KNN_best = KNeighborsClassifier(n_neighbors = K_best)
model_KNN_best.fit(X_train, y_train)

#Predict the labels using X_test
KNN_y_pred = model_KNN_best.predict(X_test)

#Calculate the metrics for the KNN Classifier
KNN_Accuracy = accuracy_score(y_test, KNN_y_pred)
KNN_Precision = precision_score(y_test, KNN_y_pred, average='weighted')
KNN_Recall = recall_score(y_test, KNN_y_pred, average='weighted')
KNN_F1 = f1_score(y_test, KNN_y_pred, average='weighted')

#Display the metrics for the KNN Classifier
print("Best K value:", K_best)
print("K-Nearest Neighbors Classifier Accuracy: %.4f" % (KNN_Accuracy))
print("K-Nearest Neighbors Classifier Precision: %.4f" % (KNN_Precision))
print("K-Nearest Neighbors Classifier Recall: %.4f" % (KNN_Recall))
print("K-Nearest Neighbors Classifier F1-Score: %.4f" % (KNN_F1))
print("Classification Report: " "\n", classification_report(y_test, KNN_y_pred))
print("Confusion Matrix: ", "\n", confusion_matrix(y_test, KNN_y_pred))

#Create and display heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(confusion_matrix(y_test, KNN_y_pred), annot=True, fmt='d', cmap='Blues',
            xticklabels=classes, yticklabels=classes)
plt.title('Confusion Matrix - KNN Classifier')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()
```

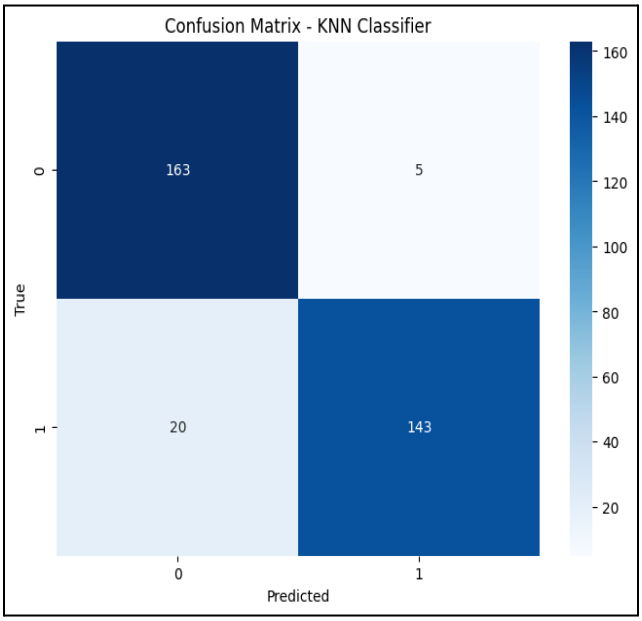


Figure 15: KNN Model - DADA Dataset

K-Nearest Neighbors (KNN) Model calculates the Euclidean distance to categorize data points to the corresponding class. Our parameters included performing a five fold cross validation, the ranges of k values [2,3,5,7,9,11] and n_neighbors = best_k which was 3 for the CICMaldroid

2020 Dataset and 5 for the DADA Dataset. As we looked to see how many features were classified correctly in the CICMaldroid 2020 Dataset, we saw in Figure 14 that there were 219 Adware features, 366 Banking features, 777 SMS malware features, 465 Riskware features and 267 Benign features that were correctly classified followed again by a small amount of features that were misclassified. In Figure 15, the KNN model classified 163 benign features and 143 malware features correctly in the DADA dataset and 25 features were misclassified.

```
#Create SVM Model
svm_classifier_rbf = SVC(kernel='rbf', C=1.0, random_state=42, verbose=5, max_iter=1000, gamma = 'auto')

#Fit and train classifier on the training data
svm_classifier_rbf.fit(X_train, y_train)

#Predict the labels using X_test
y_pred_svm_rbf = svm_classifier_rbf.predict(X_test)

#Calculate the metrics for the SVM Classifier
svm_accuracy_rbf = accuracy_score(y_test, y_pred_svm_rbf)
svm_precision_rbf = precision_score(y_test, y_pred_svm_rbf,average='weighted')
svm_recall_rbf = recall_score(y_test, y_pred_svm_rbf,average='weighted')
svm_f1_rbf = f1_score(y_test, y_pred_svm_rbf,average='weighted')

#Display the metrics for the SVM Classifier
print("SVM RBF Classifier Accuracy: %.4f" % (svm_accuracy_rbf))
print("SVM RBF Classifier Precision: %.4f" % (svm_precision_rbf))
print("SVM RBF Classifier Recall: %.4f" % (svm_recall_rbf))
print("SVM RBF Classifier F1-Score: %.4f" % (svm_f1_rbf))
print("Classification Report: " "\n", classification_report(y_test, y_pred_svm_rbf))
print("Confusion Matrix: ", "\n", confusion_matrix(y_test, y_pred_svm_rbf))

#Create and display heat map
conf_matrix = confusion_matrix (y_pred_svm_rbf, y_test)
sns.heatmap(conf_matrix , cmap='Blues', fmt='',
            xticklabels=['Adware' ,'Banking' ,'SMS malware', 'Riskware','Benign'],
            yticklabels=['Adware' ,'Banking' ,'SMS malware', 'Riskware','Benign'], annot=True)
plt.title('Confusion Matrix - SVM RBF')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()
```

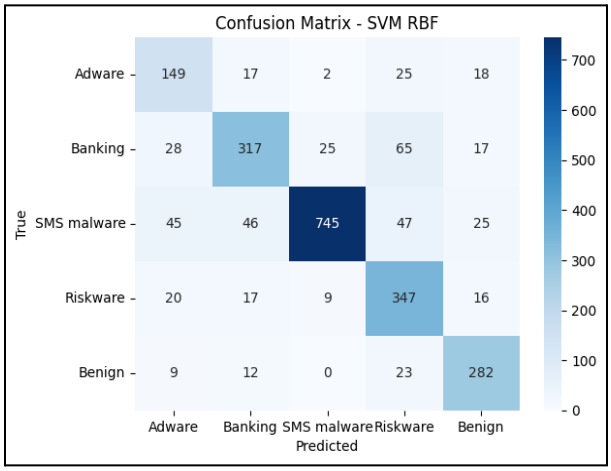


Figure 16: SVM 'rbf' Model - CICMaldroid Dataset

```
#Create SVM Model
svmClassifierRBF = SVC(kernel = 'rbf', C = 1.0, random_state = 42, verbose = 5, max_iter = 1000, gamma = 'auto')

#Fit and train classifier on the training data
svmClassifierRBF.fit(X_train, y_train)

#Predict the labels using X_test
svmRBF_y_pred = svmClassifierRBF.predict(X_test)

#Calculate the metrics for the SVM Classifier
svmRBF_Accuracy = accuracy_score(y_test, svmRBF_y_pred )
svmRBF_Precision = precision_score(y_test, svmRBF_y_pred ,average = 'weighted')
svmRBF_Recall = recall_score(y_test, svmRBF_y_pred ,average = 'weighted')
svmRBF_F1 = f1_score(y_test, svmRBF_y_pred ,average = 'weighted')

#Display the metrics for the SVM Classifier
print("SVM RBF Classifier Accuracy: %.4f" % (svmRBF_Accuracy))
print("SVM RBF Classifier Precision: %.4f" % (svmRBF_Precision))
print("SVM RBF Classifier Recall: %.4f" % (svmRBF_Recall))
print("SVM RBF Classifier F1-Score: %.4f" % (svmRBF_F1))
print("Classification Report: " "\n", classification_report(y_test, svmRBF_y_pred))
print("Confusion Matrix: ", "\n", confusion_matrix(y_test, svmRBF_y_pred))

#Create and display heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(confusion_matrix(y_test, svmRBF_y_pred), annot=True, fmt='d', cmap='Blues',
            xticklabels=classes, yticklabels=classes)
plt.title('Confusion Matrix - SVM RBF')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()
```

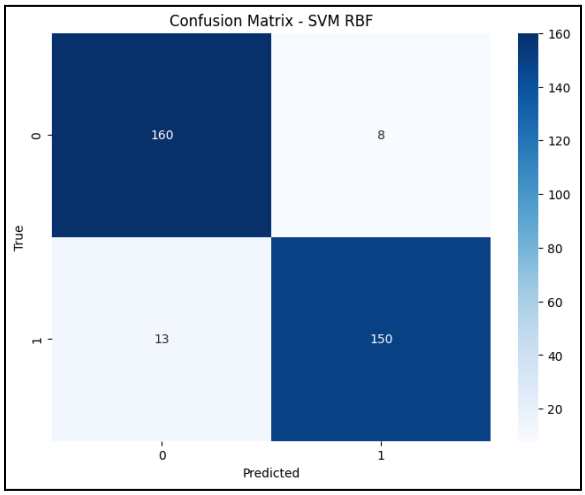


Figure 17: SVM 'rbf' Model - DADA Dataset

The SVM Model solves classification and regression problems by performing different kernel functions to transform the data features. The SVM parameters included in our implementation for SVM Model using the 'rbf' kernel were C=1.0 and random_state = 42. According to Figure 16, the SVM model with the kernel set to 'rbf' classified 149 Adware features, 317 Banking

features, 745 SMS malware features, 347 Riskware features and 282 Benign features correctly. However, we saw that there were still a small number of features that were misclassified. For the DADA dataset in Figure 17, the model displays that 160 benign features and 150 malware features were correctly classified and 21 features were misclassified.

```python
#Create an SVM classifier
svm_classifier_linear = SVC(kernel='linear', C=1.0, random_state=42, verbose = 5, max_iter=1000000, gamma = 'scale')

#Fit and train classifier on the training data
svm_classifier_linear.fit(X_train, y_train)

#Predict the labels using X_test
y_pred_svm_linear = svm_classifier_linear.predict(X_test)

#Calculate the metrics for the SVM Classifier
svm_accuracy_linear = accuracy_score(y_test, y_pred_svm_linear)
svm_precision_linear = precision_score(y_test, y_pred_svm_linear,average='weighted')
svm_recall_linear = recall_score(y_test, y_pred_svm_linear,average='weighted')
svm_f1_linear = f1_score(y_test, y_pred_svm_linear,average='weighted')

#Display the metrics for the SVM Classifier
print("SVM Linear Classifier Accuracy: %.4f" % (svm_accuracy_linear))
print("SVM Linear Classifier Precision: %.4f" % (svm_precision_linear))
print("SVM Linear Classifier Recall: %.4f" % (svm_recall_linear))
print("SVM Linear Classifier F1-Score: %.4f" % (svm_f1_linear))
print("Classification Report: " "\n", classification_report(y_test, y_pred_svm_linear))
print("Confusion Matrix: ", "\n", confusion_matrix(y_test, y_pred_svm_linear))

#Create and display heat map
conf_matrix = confusion_matrix(y_pred_svm_linear, y_test)
sns.heatmap(conf_matrix , cmap='Blues', fmt='',
            xticklabels=['Adware' ,'Banking' ,'SMS malware', 'Riskware','Benign'],
            yticklabels=['Adware' ,'Banking' ,'SMS malware', 'Riskware','Benign'], annot=True)
plt.title('Confusion Matrix - SVM Linear')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()
```

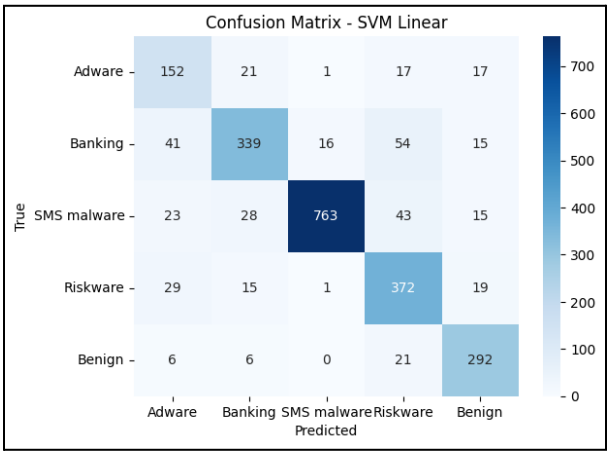


Figure 18: SVM 'linear' Model - CICMaldroid Dataset

```python
#Create SVM Model
svm_LinearClassifier = SVC(kernel = 'linear', C = 1.0, random_state = 42, verbose = 5, max_iter = 1000000, gamma = 'scale')

#Fit and train classifier on the training data
svm_LinearClassifier.fit(X_train, y_train)

#Predict the labels using X_test
svm_Linear_y_pred = svm_LinearClassifier.predict(X_test)

#Calculate the metrics for the SVM Classifier
svm_Linear_Accuracy = accuracy_score(y_test, svm_Linear_y_pred)
svm_Linear_Precision = precision_score(y_test, svm_Linear_y_pred,average = 'weighted')
svm_Linear_Recall = recall_score(y_test, svm_Linear_y_pred,average = 'weighted')
svm_Linear_F1 = f1_score(y_test, svm_Linear_y_pred,average = 'weighted')

#Display the metrics for the SVM Classifier
print("SVM Linear Classifier Accuracy: %.4f" % (svm_Linear_Accuracy))
print("SVM Linear Classifier Precision: %.4f" % (svm_Linear_Precision))
print("SVM Linear Classifier Recall: %.4f" % (svm_Linear_Recall))
print("SVM Linear Classifier F1-Score: %.4f" % (svm_Linear_F1))
print("Classification Report: " "\n", classification_report(y_test, svm_Linear_y_pred))
print("Confusion Matrix: ", "\n", confusion_matrix(y_test, svm_Linear_y_pred))

#Create and display heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(confusion_matrix(y_test, svm_Linear_y_pred), annot=True, fmt='d', cmap='Blues',
            xticklabels=classes, yticklabels=classes)
plt.title('Confusion Matrix - SVM Linear')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()
```

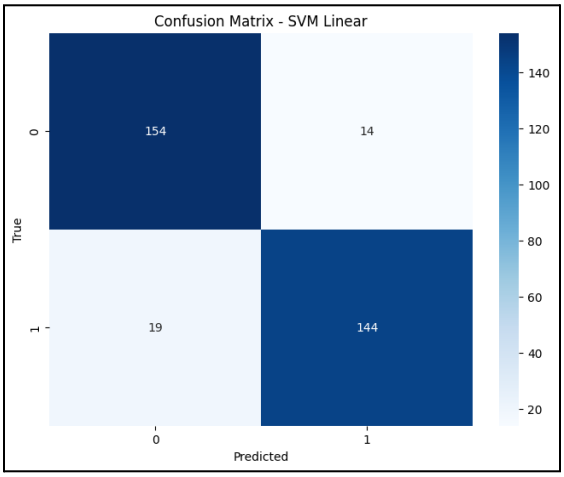


Figure 19: SVM 'linear' Model - DADA Dataset

The SVM parameters included for SVM Model using the 'linear' kernel were C=1.0 and random_state = 42. In Figure 18, the confusion matrix heatmap shows us that the model classified 152 Adware features, 339 Banking features, 763 SMS malware features, 372 Riskware features and 292 Benign features correctly followed by a minority of features that were still misclassified. For the DADA dataset in Figure 19, the model classified 154 benign features and 144 malware features correctly and 33 features were classified incorrectly.

Deep Learning Model

The deep learning model that we decided to use for our implementation was a Dense Neural Network model. The DNN model connects the neurons of the layer to each neuron in its previous layer. Both DNN models are built slightly different from one another to be able to adapt to the two different datasets we are utilizing. During preprocessing the data, we defined our features and targets. We also perform feature selection by dropping the features we are not utilizing and then scaling the data using the Standard Scaler function from Sklearn.

```
#Building the Dense Neural Network model architecture
model = keras.Sequential([
    #1st Dense layer takes the size of the amount of features in the dataframe and utilizes 'relu' activation function
    layers.Dense(470, activation='relu', input_shape=(X_train.shape[1],)),
    #Regularize to prevent overfitting
    layers.Dropout(0.5),
    #2nd Dense layer takes the size of half of the amount of features inthe data frame and utilizes 'relu' function
    layers.Dense(235, activation='relu'),
    #3rd Dense layer takes in the number of class values and utilizs the 'softmax' activation function
    layers.Dense(5, activation='softmax')
])
```

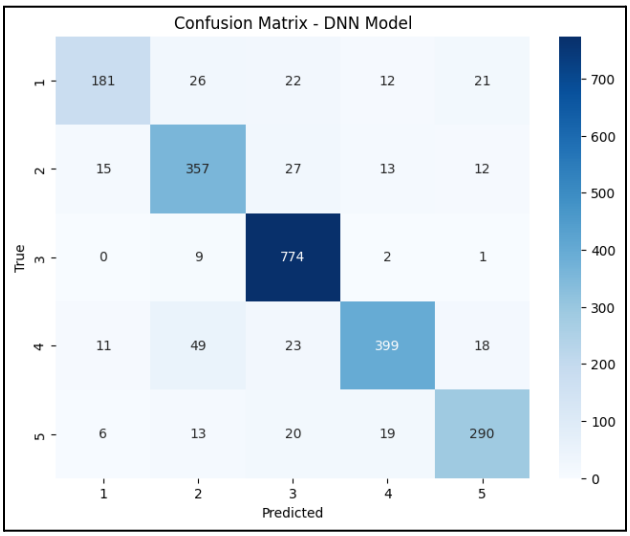


Figure 20: DNN Model - CICMaldroid 2020 Dataset

For the CICMaldroid 2020 Dataset, the DNN model consists of 3 Dense layers and 1 Dropout layer and the Sequential function is used to build the model. The first 2 Dense layers utilize the number of features in the dataset, ‘relu’ activation function and X_train for the input shape. The last dense layer utilizes the ‘softmax’ activation function and to compile this model we used the adam optimizer, sparse categorical cross-entropy and metrics taking in the accuracy. The loss function, sparse categorical cross-entropy is used because in this dataset, it is to be considered a multiclass classification task since there are a total of 5 classes.

```
#Building the Dense Neural Network model architecture
model = keras.Sequential([
    #1st Dense layer takes the size of the amount of features in the dataframe and utilizes 'relu' activation function
    layers.Dense(215, activation='relu', input_shape=(X_train.shape[1],)),
    #Regularize to prevent overfitting
    layers.Dropout(0.5),
    #2nd Dense layer takes the size of half of the amount of features inthe data frame and utilizes 'relu' function
    layers.Dense(108, activation='relu'),
    #3rd Dense layer takes in the number of class values and utilizs the 'softmax' activation function
    layers.Dense(1, activation='sigmoid')
])
```

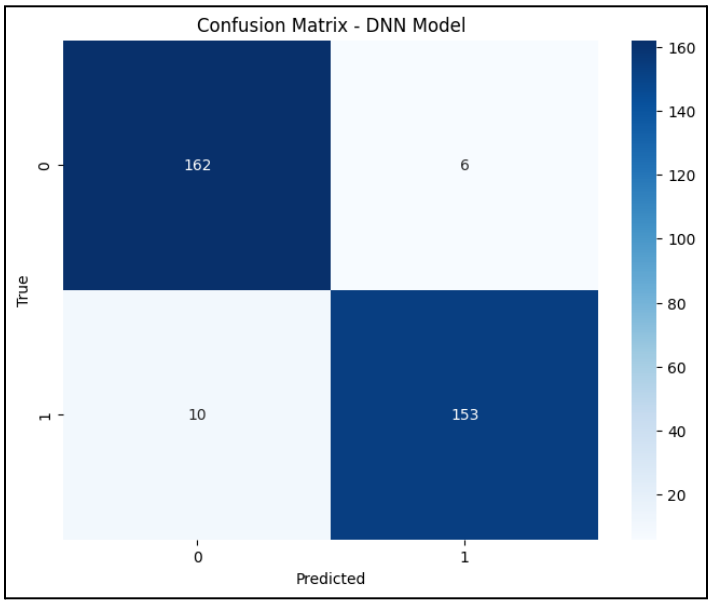


Figure 21: DNN Model - DADA Dataset

For the DADA Dataset, the Sequential function is also utilized to build the DNN model and also consists of 3 Dense layers and 1 dropout layer. Similarly to the CICMaldroid 2020 dataset we X_train for input shape and 'relu' activation function in the first two dense layers. However, because this dataset is binary, consisting of 0's and 1's, we use the 'sigmoid' activation function in the last dense layer and to compile the model, we also use binary cross entropy as the loss function.

The Dense Neural Network model in the CICMaldroid 2020 Dataset classified correctly, 181 Adware features, 357 Banking features, 774 SMS malware features, 399 Riskware features and 290 Benign features according to Figure 20. Majority were correctly classified even though there was still an amount that was misclassified. In comparison to the DADA dataset, we see in Figure 21, the DNN model classified 162 benign features and 153 malware features correctly and 16 features were misclassified.

Adversarial Attack

```python
# Function to generate adversarial examples using feature importances
def fgsm_attack(X, importance, epsilon=0.1):
    perturbation = epsilon * importance
    X_adv = X + perturbation
    X_adv = np.clip(X_adv, 0, 1)
    return X_adv

# Calculate feature importances
importance = random_forest_model.feature_importances_

# Generate adversarial examples using FGSM attack
X_test_adv = fgsm_attack(X_test, importance, epsilon=0.1)

# Evaluate the model on adversarial examples
y_pred_rf_adv = random_forest_model.predict(X_test_adv)
```

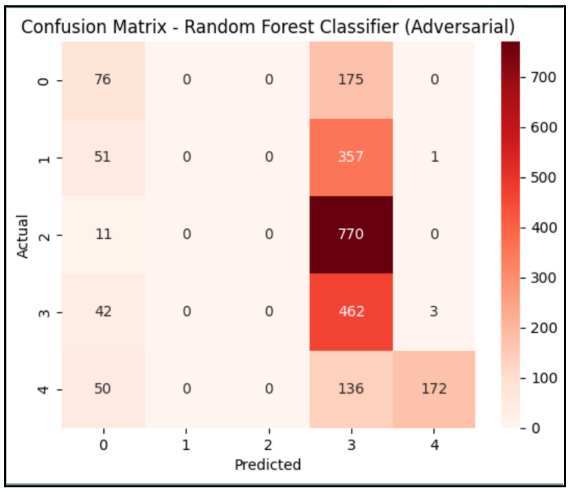


Figure 22: Random Forest Adversarial Attack - CICMaldroid 2020 Dataset

In Figure 22, we can see that the FGSM adversarial attack uses the epsilon value as the size of the attack and the importance of the features in order to obtain the perturbations to attack the model. Based on the CICMaldroid 2020 dataset's confusion matrix for the adversarial attack, we can see that many of the classes have been misclassified, specifically to class 3. This shows that the adversarial attack was effective in attacking the random forest machine learning model as it is misclassifying what has been classified from the original model.

```python
# Function to generate adversarial examples using feature importances
def iterative_fgsm_attack_clip_range(X, importance, epsilon, clip_min, clip_max, num_iterations):
    perturbation = epsilon * importance
    X_adv = X.copy()

    for _ in range(num_iterations):
        X_adv += perturbation
        X_adv = np.clip(X_adv, clip_min, clip_max)

    return X_adv

# Calculate feature importances
importance = random_forest_model.feature_importances_
X_test_flt = X_test.astype('float64')

# Generate adversarial examples using iterative FGSM attack with clipping, increased epsilon, and more iterations
X_test_adv = iterative_fgsm_attack_clip_range(X_test_flt, importance, epsilon=1.0, clip_min=-1,
                                              clip_max=2, num_iterations=100)

# Evaluate the model on adversarial examples
y_pred_rf_adv = random_forest_model.predict(X_test_adv)
```

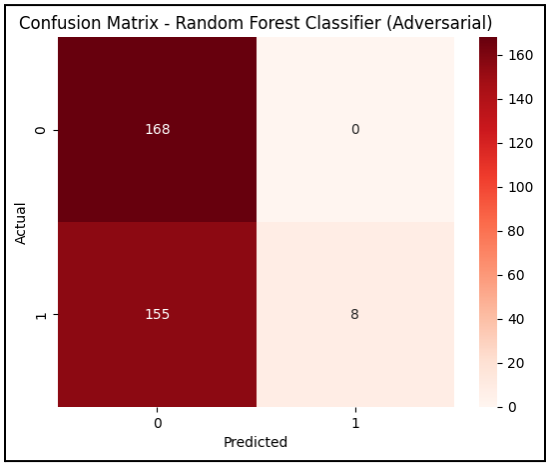


Figure 23: Random Forest Adversarial Attack - DADA Dataset

Similarly, based on the DADA dataset, we can also see that the adversarial attack has effectively attacked the random forest model. In Figure 23, we can see the confusion matrix shows the misclassification of a majority of the samples to be benign when they should be malware.

Defense Mechanism

```python
# Function to perform adversarial training
def adversarial_training(model, X_train, y_train, epsilon=0.1, n_iterations=10):
    for _ in range(n_iterations):
        # Train on clean data
        model.fit(X_train, y_train)

        # Calculate feature importances
        importance = model.feature_importances_

        # Generate adversarial examples using FGSM attack
        X_train_adv = fgsm_attack(X_train, importance, epsilon)

        # Combine clean and adversarial examples
        X_train_combined = np.vstack([X_train, X_train_adv])
        y_train_combined = np.hstack([y_train, y_train])

        # Shuffle the data
        indexes = np.arange(len(X_train_combined))
        np.random.shuffle(indexes)
        X_train_combined_shuffled = X_train_combined[indexes]
        y_train_combined_shuffled = y_train_combined[indexes]

        # Retrain the model on the combined dataset
        model.fit(X_train_combined_shuffled, y_train_combined_shuffled)

    return model

# Adversarial training on the Random Forest model
adversarial_random_forest_model = adversarial_training(random_forest_model, X_train, y_train, epsilon=0.1)

# Evaluate the defended model on adversarial examples
y_pred_rf_adv = adversarial_random_forest_model.predict(X_test_adv)
```

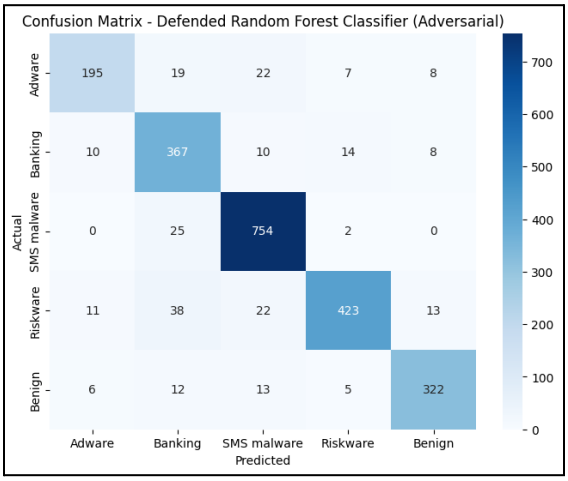


Figure 24: Random Forest Defense Mechanism - CICMaldroid 2020 Dataset

When implementing the defense mechanism through adversarial training on the attack for the CICMaldroid 2020 dataset, from Figure 24, we can see that many of the samples have been reclassified. This has been done by generating clean samples and combining it with the adversarial examples in order to retrain the model for it to learn whether a sample is truly clean or not. When implementing the defense mechanism, it has shown to be effective as many of the misclassified samples are not reclassified to their true labels.

```python
# Function to perform adversarial training
def adversarial_training(model, X_train, y_train, importance, epsilon, clip_min, clip_max, num_iterations):
    for _ in range(num_iterations):
        # Generate adversarial examples
        X_train_adv = iterative_fgsm_attack_clip_range(X_train, importance, epsilon, clip_min, clip_max, num_iterations)

        # Ensure consistent data types
        X_train_combined = np.vstack([X_train.astype('float64'), X_train_adv])
        y_train_combined = np.hstack([y_train, y_train])

        # Shuffle the combined data
        idx = np.random.permutation(len(X_train_combined))
        X_train_combined_shuffled = X_train_combined[idx]
        y_train_combined_shuffled = y_train_combined[idx]

        # Retrain the model on the combined dataset
        model.fit(X_train_combined_shuffled, y_train_combined_shuffled)

    return model

# Convert data types for consistency
X_train_flt = X_train.astype('float64')
X_test_adv = X_test_adv.astype('float64')

# Adversarial training on the Random Forest model
random_forest_model_defense = adversarial_training(random_forest_model, X_train, y_train,
                                                   importance, epsilon=1.0, clip_min=-1, clip_max=2, num_iterations=100)

# Evaluate the defended model on adversarial examples
y_pred_rf_adv_defense = random_forest_model_defense.predict(X_test_adv)
```

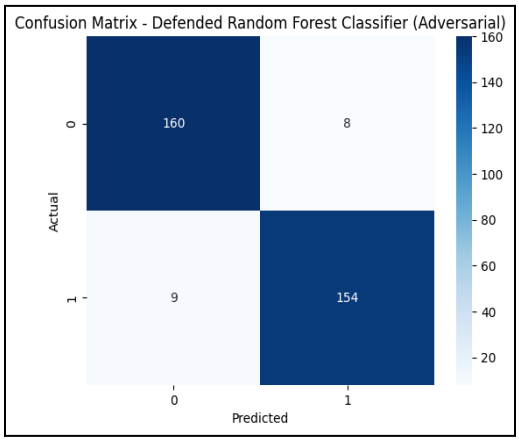


Figure 25: Random Forest Defense Mechanism - DADA Dataset

Similarly to the previous dataset, we can see that the DADA dataset has samples that have been reclassified as their accurate true labels. Based on the confusion matrix in Figure 25, we can see that a majority of the samples have been classified correctly whereas a minimal amount of samples have been misclassified.

Experimentation/Testing

	Naive Bayes	Random Forest	Logistic Regression	KNN	SVM (‘rbf’)	SVM (‘linear’)	Dense Neural Network
Accuracy	0.5889	0.9480	0.8053	0.9055	0.7979	0.8317	0.8625
Precision	0.6804	0.9484	0.8072	0.9092	0.8000	0.8323	0.8629
Recall	0.5889	0.9480	0.8053	0.9055	0.7979	0.8317	0.8625
F1-Score	0.5486	0.9479	0.8007	0.9053	0.7942	0.8285	0.8602

Figure 26: CICMaldroid 2020 Metrics

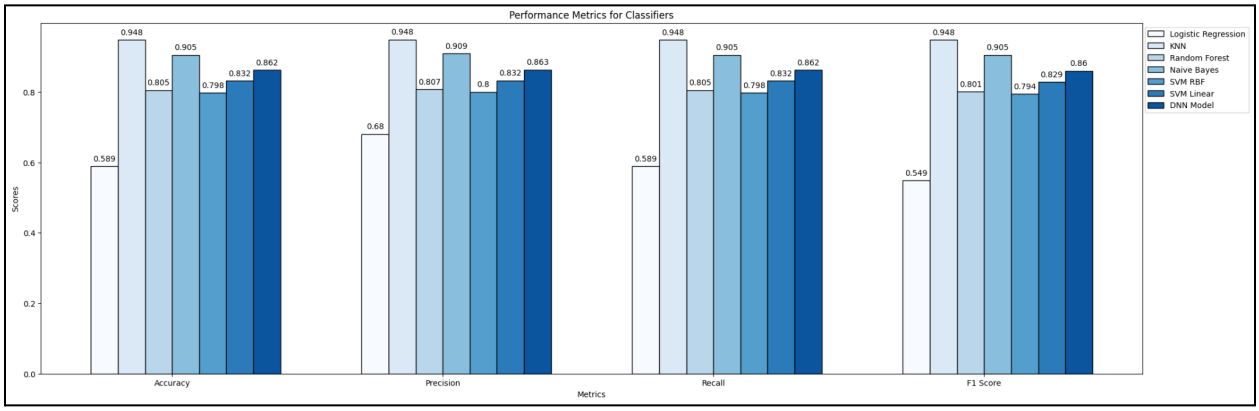


Figure 27: CICMaldroid 2020 Dataset Models’ Metrics Comparison

Based on the performance of the CICMaldroid 2020 Dataset in Figure 27, we can see that all of the machine learning and deep learning models performed differently. The machine learning model that had the highest accuracy score was the Random Forest Model at 94.8%. The machine learning model that had the lowest accuracy score was the Naive Bayes model at 58.89%. Our deep learning model performed fairly well and had an accuracy score of 86.25%. The order in which the models performed for the CICMaldroid 2020 Dataset from highest accuracy score to lowest are as follows: Random Forest, KNN, DNN, SVM (‘linear’), Logistic Regression, SVM (‘rbf’) and Naive Bayes.

	Naive Bayes	Random Forest	Logistic Regression	KNN	SVM (‘rbf’)	SVM (‘linear’)	Dense Neural Network
Accuracy	0.6435	0.9637	0.9094	0.9245	0.9366	0.9003	0.9517
Precision	0.7547	0.9640	0.9094	0.9279	0.9369	0.9006	0.9519
Recall	0.6435	0.9637	0.9094	0.9245	0.9366	0.9003	0.9517
F1-Score	0.5967	0.9637	0.9094	0.9243	0.9365	0.9003	0.9516

Figure 28: DADA Dataset Metrics

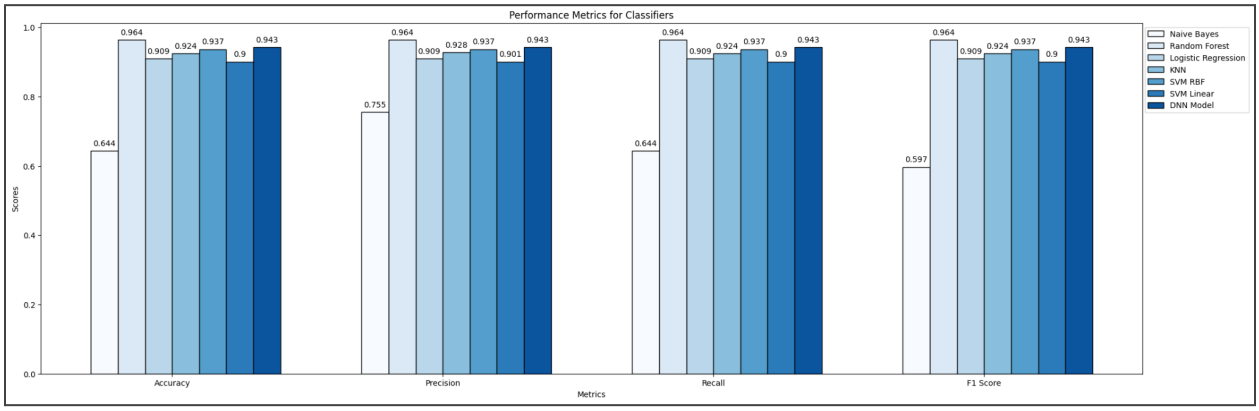


Figure 29: DADA Dataset Models’ Metrics Comparison

Looking at the DADA Dataset and how it performed in Figure 29, we can see that all of the machine learning and deep learning models again performed differently but are similar to the CICMaldroid 2020 Dataset. The machine learning model that had the highest accuracy score was the Random Forest Model at 90.94%. The machine learning model that had the lowest accuracy score was the Naive Bayes model at 64.35%. Our deep learning model performed very well and had the second highest accuracy score which was 95.17%. The order in which the models performed for the DADA Dataset from highest accuracy score to lowest are as follows: Random Forest, DNN, SVM (‘rbf’), KNN, Logistic Regression, SVM (‘linear’) and Naive Bayes.

		Accuracy Score	Precision	Recall	F1-Score
CICMaldroid 2020 Dataset	RF Adversarial Attack	0.3079	0.2411	0.3079	0.2188
	RF Defense Mechanism	0.8938	0.8965	0.8938	0.8934
DADA Dataset	RF Adversarial Attack	0.5317	0.7564	0.5317	0.3934
	RF Defense Mechanism	0.9486	0.9487	0.9486	0.9486

Figure 30: Random Forest Adversarial Attack and Defense Mechanisms Metrics

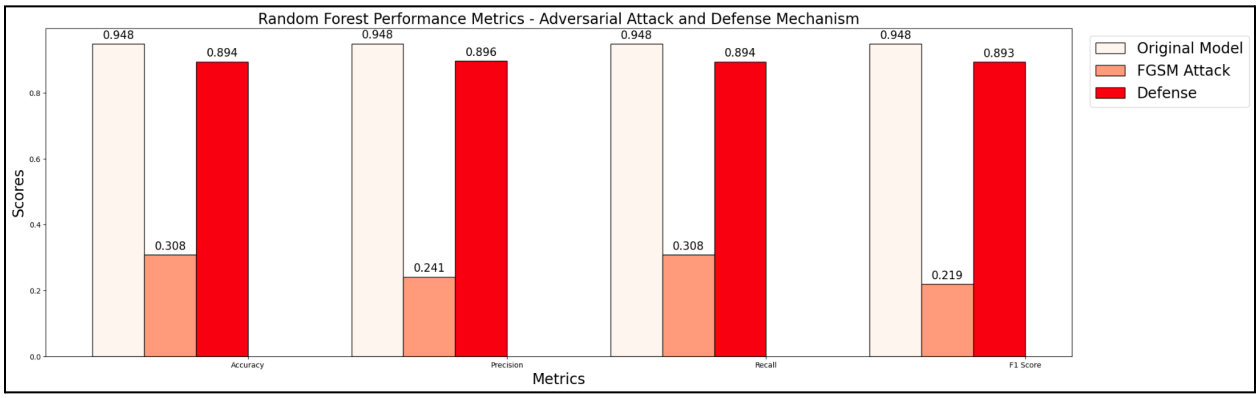


Figure 31: Random Forest Adversarial Attack and Defense Mechanisms Metrics

CICMaldroid 2020 Dataset

On the CICMaldroid 2020 Dataset, in Figure 31 we can see that the accuracy, precision, recall, and f1-score are around 94%. Once the adversarial attack was launched on the model, we can see that all the metrics significantly depleted. Specifically, the accuracy of the original model dropped from 94.8% to 30.8% showing how much the FGSM attack impacted the model. Additionally, looking at the defense mechanism, we can see that the accuracy was brought back up to around 89.4% which shows that the defense did bring the accuracy back up.

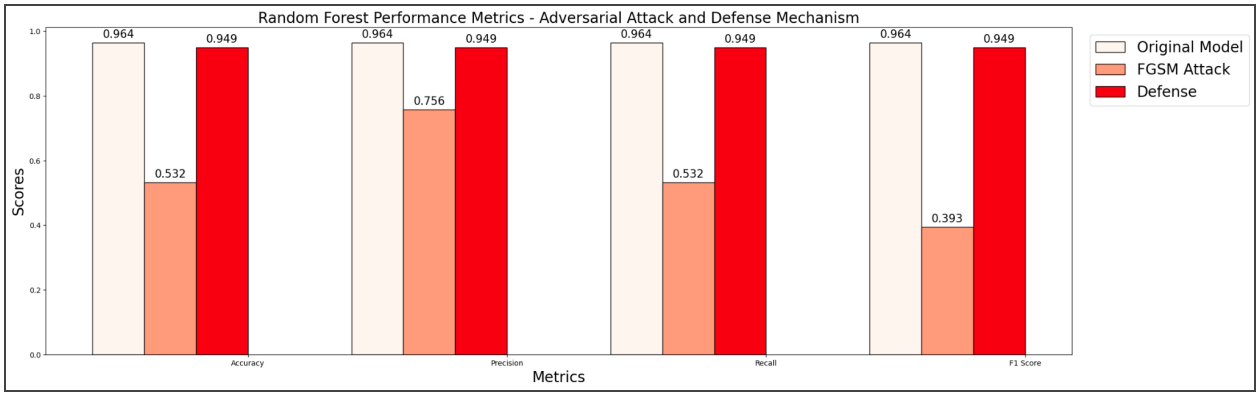


Figure 32: Random Forest Adversarial Attack and Defense Mechanisms Metrics DADA Dataset

On the DADA Dataset, in Figure 32 we can see that the accuracy, precision, recall, and f1-score are all around 96%. When launching the adversarial attack on the model, we can see that all of the metrics depleted. Although not as significant as the CICMaldroid 2020 Dataset, there was still a substantial drop. Specifically, the accuracy of the original model dropped from 96.4% to 53.2% which shows how the FGSM attack impacted the model. When looking at the defense mechanism, we can see that the accuracy was brought back up to around 94% which is almost to the original accuracy. This shows that the adversarial training defense mechanism impacted this model significantly.

PowerBI Analytics Dashboard

In order to better visualize the metrics obtained from the machine learning and deep learning models, we used Power BI. First we took the metrics and imported them into an excel file. Then in Power BI, we transformed the data into columns that can be manipulated for the visualization.

Datasets	Accuracy Score	Precision	Recall	F1-Score	ML/DL Models
CICMaldroid 2020 Dataset	0.948	0.9484	0.948	0.9479	Random Forest
CICMaldroid 2020 Dataset	0.5889	0.6804	0.5889	0.5486	Naïve Bayes
CICMaldroid 2020 Dataset	0.8053	0.8072	0.8053	0.8007	Logistic Regression
CICMaldroid 2020 Dataset	0.9055	0.9092	0.9055	0.9053	KNN
CICMaldroid 2020 Dataset	0.7979	0.8	0.7979	0.7942	SVM 'rbf'
CICMaldroid 2020 Dataset	0.8317	0.8323	0.8317	0.8285	SVM 'linear'
CICMaldroid 2020 Dataset	0.8625	0.8629	0.8625	0.8602	Dense Neural Network
DADA Dataset	0.9637	0.964	0.9637	0.9637	Random Forest
DADA Dataset	0.6435	0.7547	0.6435	0.5967	Naïve Bayes
DADA Dataset	0.9094	0.9094	0.9094	0.9094	Logistic Regression
DADA Dataset	0.9245	0.9279	0.9245	0.9243	KNN
DADA Dataset	0.9366	0.9369	0.9366	0.9365	SVM 'rbf'
DADA Dataset	0.9003	0.9006	0.9003	0.9003	SVM 'linear'
DADA Dataset	0.9517	0.9519	0.9517	0.9516	Dense Neural Network
CICMaldroid 2020 Dataset	0.3079	0.2411	0.3079	0.2188	RF Adversarial Attack
CICMaldroid 2020 Dataset	0.8938	0.8965	0.8938	0.8934	RF Defense Mechanism
DADA Dataset	0.5317	0.7564	0.5317	0.3934	RF Adversarial Attack
DADA Dataset	0.9486	0.9487	0.9486	0.9486	RF Defense Mechanism

Figure 33: PowerBI Table View of Evaluation Metrics

In the Table View seen in Figure 33, we are able to see two columns that consist of the datasets and corresponding models that were used in our research implementation. The next columns are the evaluation metrics and their data corresponding to their models and datasets.

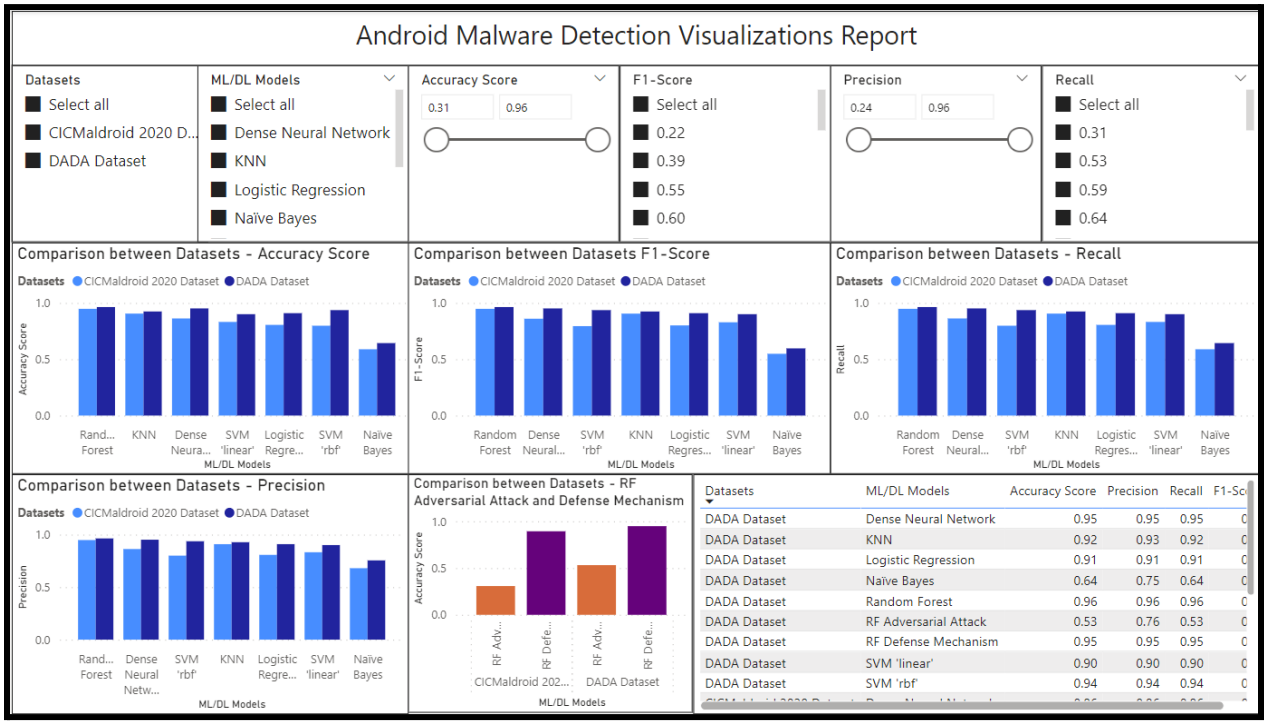


Figure 34: PowerBI Visualization Report for Model Evaluation Metrics

From Figure 34, this report implemented six filters that are utilized for the visualizations. The six filters, Datasets, ML/DL Models, Accuracy Score, F1-Score, Precision and Recall are based on the columns that were created when we transformed the data. The five visualizations that we created are Comparison between Datasets, representing accuracy score, f1-score, recall, precision, and Random Forest Adversarial Attack and Defense Mechanism. All of the filters are set to its default setting, select all which means all the visualizations are currently displaying all models from both datasets. We can use the filters to adjust the visualizations further by taking a closer look to compare two models from both datasets rather than all six. The table at the bottom right displays all the data that is being shown in the visualization that was created from the filters. If the filters are modified, all visualizations including the table will reflect those filters to provide more specific visualizations and analysis.

To further our analysis and understanding, according to this report we see that the Adversarial Attack in the CICMaldroid Dataset had more of an impact on the Random Forest model than the Adversarial Attack in the DADA dataset. We also see that the Defense Mechanism performed well on the Random Forest model after it was attacked in the CICMaldroid dataset; however, in the DADA dataset, the Defense Mechanism did not help to improve the accuracy score by much.

Conclusion

In conclusion, this project has been a comprehensive exploration of Android system security against malicious attacks as well as defense mechanisms. Through the application of machine learning models on our two datasets, the team gained valuable insights into the performance and behavior of machine learning and deep learning models within Android malware detection. The use of the adversarial attack along with a defense mechanism represents contribution to the field of cybersecurity and how attacks can affect machine learning models. Although there were some challenges and limitations which range from corrupt APK files to the lack of computational resources, we were still able to obtain additional insights into how our FGSM attack and defense mechanism performed. Upon reflection, we have recognized the importance of the continuous research and innovation against malware and its significance in the modern era.

Future Works

Based on our challenges faced, some of our future work includes implementing additional machine and deep learning models to test the adversarial attacks and defense mechanisms. We also look to implement various different adversarial attacks on newly generated models in order to work towards robustness of the models and work of fine tuning the current defense mechanism.

Team Contributions

Selina Narain conducted research into the Random Forest Model, the KNN Models, the adversarial attacks, and their defense mechanisms. She worked on the visualizations from the flowchart to the random forest model, KNN and attack and defense mechanism model visualizations. She also worked on the development of the models and training them. Neelam Boywah conducted research into the Dense Neural Network, the Support Vector Machine, the adversarial attack and defense mechanisms. She worked on the visualizations for the DNN, SVM, attack and defense mechanisms. She also worked on the development of the models and training them. She also worked on building the PowerBI Dashboard. Zoya Haq conducted research into the Naive Bayes, Logistic Regressions, adversarial attacks and defense mechanisms. She worked on the visualizations for the Naive Bayes, Logistic Regression, attack and defense mechanisms. She also worked on the development of the models and training them.

References

[1] A. Daniel, G. Hugo, R. Konrad, H. Malte and S. Michael, “Drebin: Effective and Explainable Detection of Android Malware in Your Pocket,” https://www.researchgate.net/profile/Hugo-Gascon/publication/264785935_DREBIN_Effective_and_Explainable_Detection_of_Android_Malware_in_Your_Pocket/links/53efd0020cf26b9b7dcdf395/DREBIN-Effective-and-Explainable-Detection-of-Android-Malware-in-Your-Pocket.pdf

[2] A. Kevin, B. Tegawende, K. Jacques and T. Yves, “AndroZoo:collecting millions of Android apps for the research community”, Proceedings of the 13th International Conference on Mining Software Repositories, 2016, Pages 468–471, doi: 10.1145/2901739.2903508

[3] B. Gaudenz, "What Is Adversarial Machine Learning? Attack Methods in 2024." Viso AI, https://viso.ai/deep-learning/adversarial-machine-learning/.

[4] C. Pestana, N. Akhtar, W. Liu, D. Glance and A. Mian, "Adversarial Attacks and Defense on Deep Learning Classification Models using YCbCr Color Images," 2021 International Joint Conference on Neural Networks (IJCNN), Shenzhen, China, 2021, pp. 1-9, doi: 10.1109/IJCNN52387.2021.9533495.

[5] G. D'Ambrosio and W. Li, "AdversarialDroid: A Deep Learning based Malware Detection Approach for Android System Against Adversarial Example Attacks," 2021 IEEE MIT Undergraduate Research Technology Conference (URTC), Cambridge, MA, USA, 2021, pp. 1-5, doi: 10.1109/URTC54388.2021.9701615.

[6] G. Ian, S. Jonathon and S. Christian, “Explaining and Harnessing Adversarial Examples”, Published as a conference paper at ICLR 2015

[7] H. Li, S. Zhou, W. Yuan, J. Li and H. Leung, "Adversarial-Example Attacks Toward Android Malware Detection System," in IEEE Systems Journal, vol. 14, no. 1, pp. 653-656, March 2020, doi: 10.1109/JSYST.2019.2906120.

[8] Ian J. Goodfellow, Jonathon Shlens & Christian Szegedy
K. Ren, T. Zheng, Z. Qin, and X. Liu, “Adversarial Attacks and Defenses in Deep Learning”, https://www.sciencedirect.com/science/article/pii/S209580991930503X, 2020

[9] N. Bala, A. Ahmar, W. Li, F. Tovar, A. Battu, P. Bambarkar, “DroidEnemy: Battling adversarial example attacks for Android malware detection”, https://www.sciencedirect.com/science/article/pii/S2352864821000900, 2021

[10] P. Vinod, “X–ANOVA Ranked Features for Android Malware Analysis”, https://www.researchgate.net/publication/275955119_X-ANOVA_Ranked_Features_for_Android_Malware_Analysis, 2014

[11] W. Li, J. Ge and G. Dai, "Detecting Malware for Android Platform: An SVM-Based Approach," 2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing, New York, NY, USA, 2015, pp. 464-469, doi: 10.1109/CSCloud.2015.50.

[12] W. Li, N. Bala, A. Ahmar, F. Tovar, A. Battu and P. Bambarkar, "A Robust Malware Detection Approach for Android System Against Adversarial Example Attacks," 2019 IEEE 5th International Conference on Collaboration and Internet Computing (CIC), Los Angeles, CA, USA, 2019, pp. 360-365, doi: 10.1109/CIC48465.2019.00050.

[13] W. Yang, D. Kong, T. Xie, C. A. Gunter, “Malware Detection in Adversarial Settings: Exploiting Feature Evolutions and Confusions in Android Apps”, https://dl.acm.org/doi/pdf/10.1145/3134600.3134642

[14] X. Chen et al., "Android HIV: A Study of Repackaging Malware for Evading Machine-Learning Detection," in IEEE Transactions on Information Forensics and Security, vol. 15, pp. 987-1001, 2020, doi: 10.1109/TIFS.2019.2932228.

[15] Z. Shu and G. Yan, "EAGLE: Evasion Attacks Guided by Local Explanations against Android Malware Classification," in IEEE Transactions on Dependable and Secure Computing, doi: 10.1109/TDSC.2023.3324265.

[16] Z. Wang, J. Cai, S. Cheng and W. Li, "DroidDeepLearner: Identifying Android malware using deep learning," 2016 IEEE 37th Sarnoff Symposium, Newark, NJ, USA, 2016, pp. 160-165, doi: 10.1109/SARNOF.2016.7846747.