New York Institute of Technology
COECS

A Robust Malware Detection Approach for Android System against Adversarial Example Attacks

Student Members:
Neha Bala 1133176
Aemun Ahmar 1047508
Fernanda Tovar 1090913
Arpit Battu 1129990
Prachi Bambarkar 1261251

Senior Design Project Fall 2019

Advisor:
Dr. Wenjia Li, Ph.D.
Department of Computer Science

NYIT School of Engineering & Computing Sciences

Table of Contents

Abstract

In recent years, Android has become the leading smartphone operating system across the world. However, due to their increasing popularity, Android devices have become the primary target to mobile malware.

To address the arising security threats, many malware detection approaches have been studied that aim at providing strong defense mechanisms against malware. However, with more such malware detection systems being distributed and deployed, malware authors tend to generate adversarial examples by manipulating mobile applications to avoid being detected by the malware detection systems.

For this project, we investigate different types of adversarial example attacks while researching a viable approach to fight against them. We do this by first conducting a literature review on existing malware detection approaches and adversarial attacks on these approaches. We then use that information to develop our own attack models to generate adversarial examples in order to study their behavior.

Then, we focus on evasion attack models and data poisoning attack models to generate mutated samples. By working with various app features such as API calls and Permissions, we will generate feature sets.

As a result, we used the manipulated dataset to develop and train our classifier to detect both evasion and data poisoning attacks. The goal of our approach is to further enhance the robustness of malware detection approach in the presence of adversarial example attacks.

Introduction

Amongst various mobile operating systems, Android has become the leading operating system in terms of the percentage of mobile devices that are based on it [2].

"In recent years, mobile devices, such as smartphones and tablet computers, have become increasingly popular around the world with more than 2.5 billion smartphone users [1]. "

However, Android devices are more susceptible to various security threats including mobile malware, because of the large quantity of mobile users as well as diversified mobile applications, as shown in Fig. 1.

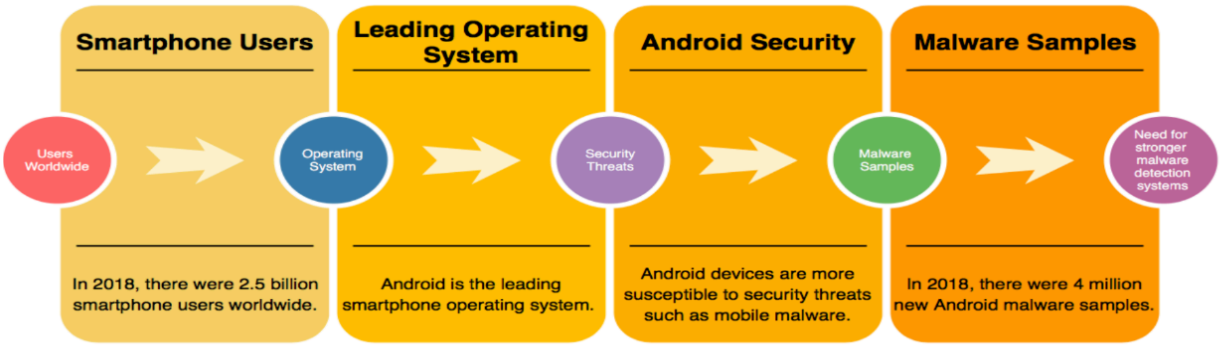


Figure 1 Current Status of Android System regarding its popularity and security threats by malware

"In 2018, there were about 4 million new malware samples discovered on the Android platform [3]. "

In general, mobile malware tricks users into downloading malicious applications with the intention of stealing sensitive and valuable user information such as banking information, location information, and login credentials, etc. Malware samples are normally known to derive out of the existing malware families, and as a result, making it harder for current malware detection systems to track down all malicious activities [4].

In recent years, researchers have explored different means, including various machine learning algorithms, such as Support Vector Machine, Random Forest, Decision Tree, and Logistic Regression, to successfully detect malicious applications in Android devices [5]. However, many of the existing malware detection systems have become vulnerable to adversarial example attacks, which are designed by malware authors to ensure that malware remains undetected by the traditional machine learning based approaches [6]. To address these emerging security threats against the malware detection systems, some research efforts have been made, such as an adversary-aware approach which can address potential adversarial threats generated by the attacker [6].

Literature Review

In this paper, we first introduce and analyze three types of well-known adversarial example attacks, namely data poisoning attack, exploratory attack and evasion attack. More particularly, we look into the evasion attack, which is conducted by feeding mutated malware samples into a detection system in order for them to falsely classify malicious applications as benign ones.

In addition, we also introduce a data poisoning attack model to generate adversarial examples. Data poisoning attacks insert malicious data into their training set. To battle both types of adversarial example attacks, we researched ways to develop a robust malware detection approach for Android system, which contains the functionalities of feature extraction, feature representation, adversarial example generation, and detection of malware with the mutated dataset, which is depicted in Fig. 2.

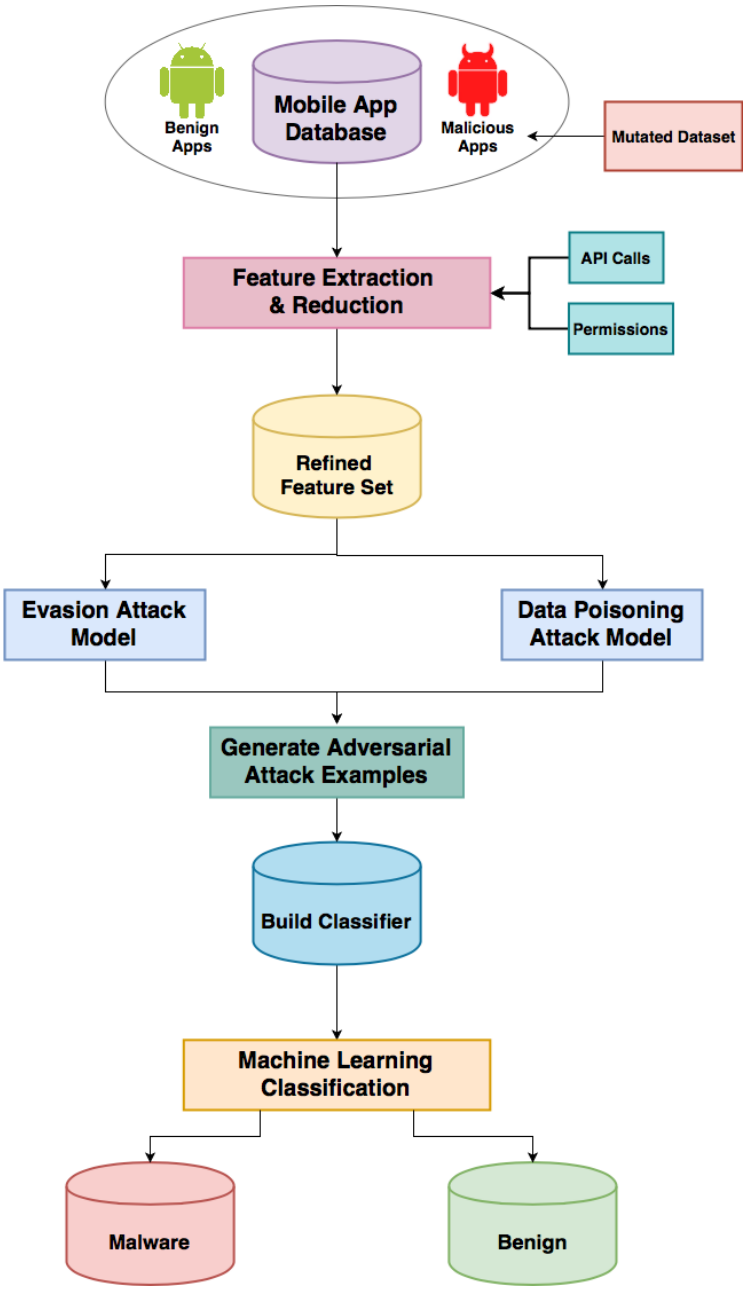


Figure 2 An Overview of System Architecture

Literature Review Cont'd

During the literature phase, the group read about several different malware detection approaches. These approaches are Static Analysis, Machine Learning, and Dynamic Analysis. Static Analysis identifies defects and issues in mobile applications without the necessity of executing them. Unfortunately, this approach doesn't find security vulnerabilities that are introduced during run time, thereby making it important for it to be followed by Dynamic Analysis so that subtle malware can be detected.

Dynamic Approach detects malware while the Android applications are being executed in a sandbox environment or on real devices to gather information about the app behavior. Once again, Dynamic Approach still has some limitations due to having an additional time overhead while running the mobile apps to check for abnormal behaviors.

The most efficient approach currently known is Machine Learning. Machine Learning takes into consideration large amounts of malware samples including those that apply the automated code obfuscation techniques to evade detection. Overall the goal of machine learning is to make the process as autonomous as possible to have a decrease in human intervention.

After reading several articles, we decided to focus on Machine Learning and more specifically we dwell into the technique known as Adversarial Machine Learning. This method tricks classification models through twisted labeled data. These adversarial examples are usually used to attack and disallow the machine learning models from properly functioning.

There are three main adversarial example attacks: Data Poisoning Attack, Exploratory Attack, and Evasion Attack. Figure 3 depicts the relationships between these attacks.

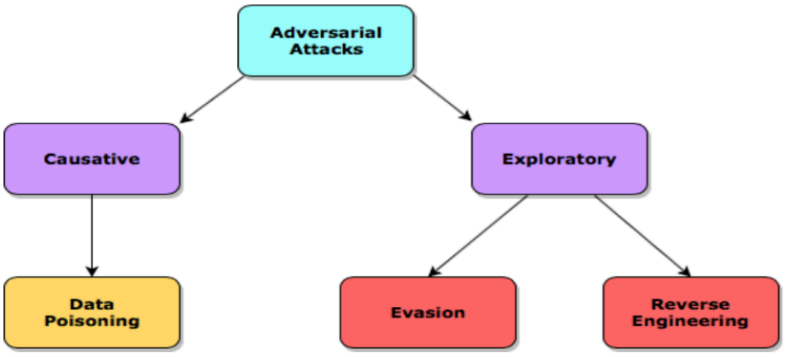


Figure 3 Taxonomy of Adversarial Example Attacks

Literature Review Cont'd

Data Poisoning Attacks

According to recent research, Machine Learning algorithms can be weakened by inserting malicious data into their training set which leads to the formation of a new type of attack called Data Poisoning Attacks [20].

In a Data Poisoning Attack, the adversary gets access to the training dataset of the learning algorithm and degrades the performance of the system. The degraded system creates a substandard attack model and leads to system evasion i.e., malware escaping detection from classifier.

There are four different attack strategies for model alteration including Label Modification, Data Injection, Data Modification and Logic Corruption, which are shown in Fig. 4.

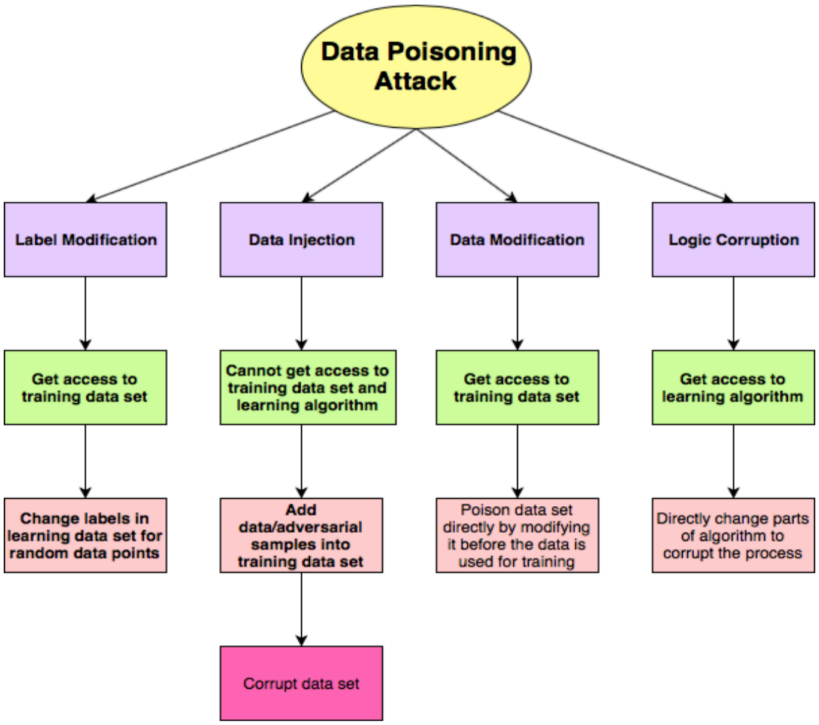


Figure 4 Workflow of Data Poisoning Attack

Literature Review Cont'd

Data Poisoning Attacks cont'd

In Label Modification, the attacker changes the labels in learning datasets for random data points. In Data Injection, the attacker does not have access to the training dataset or the learning algorithm but has the ability to add new data into the training dataset, hence corrupting it. In Data Modification, the attacker only has access to the training dataset which allows them to poison it directly by modifying data before it is used for training. Finally, in Logic Corruption, the attacker directly changes some parts of the learning algorithm.

Data Poisoning Attack is generally carried out on a non-mutated dataset as this type of attack is dependent on changing the dynamic of the learning paradigm of the algorithm. It is also dependent on changing the dataset by adding new data or samples. Defense from such Poisoning Attacks is difficult with current techniques. Methods from robust statistics that are tough against noise do not work well against poisoned data and restrictive adversarial models hinder operations of methods for sanitizing cleaning data [21].

For example, in the article "Detection of Adversarial Training Examples in Poisoning Attacks through Anomaly Detection", it mentions how malware samples are collected from compromised machines with known vulnerabilities or online services, where the labelling errors are frequent. There is also mention of previous work that has demonstrated that popular machine learning algorithms can be poisoned with well-crafted adversarial examples. These attack strategies focus on worst-case analysis and are designed to maximize the damage for the learning algorithm

Literature Review Cont'd

Exploratory Attacks

Exploratory attacks use a dynamic approach to evade the user. Instead of mutating or modifying data, exploratory attacks are executed when the system is operating. Exploratory attacks are especially complex and incognito because they work by probing the classifiers [22].

These types of attacks reveal confidential and crucial information about the datasets. Exploratory attacks are very hard to defend against because they can be extremely unpredictable. They can be implemented in multiple different ways, but they all carry the same formula, which is to be aggressive and attack the classifiers [22], [23].

Reverse Engineering: Exploratory attacks can be developed in multiple ways and one way is by Reverse Engineering, which is demonstrated in Fig. 5.

This type of attack works by trying to learn and model the classifier's boundaries. Learning more about boundaries can also reveal sensitive information about the classifiers and how the defense system works. If implemented properly, this attack can disrupt defense system [22].

For example, in the article "Adding Robustness to Support Vector Machines Against Adversarial Reverse Engineering", exploratory attacks fall under two categories, evasion and reverse engineering. Reverse Engineering presents a two-fold problem. First, the development of new machine learning algorithms not only facilitate the classifiers but also allow the adversaries to reverse engineer said classifiers. Second, once the classifier is up and running it is designed to perform according to some predefined metrics. It is possible to alter the behavior of the classifier during operation and doing so will make it harder to reverse engineer. However, it also results in reduced accuracy which indirectly translates into adversarial gain. [22]

In reverse engineering the adversary attempts to model the classifiers decision boundary. Such attack can be used as a primary step towards a sophisticated exploratory attack or it can be used by itself, for example, the decision boundary could reveal sensitive information. In the case of linear classifiers, for example, one simple reverse engineering tactic is the sign witness test, where an adversary can infer with relative ease whether a feature contributes positively or negatively to the classifier's final decision. In general, some important types of classifiers can be reverse engineered using a few probe messages.

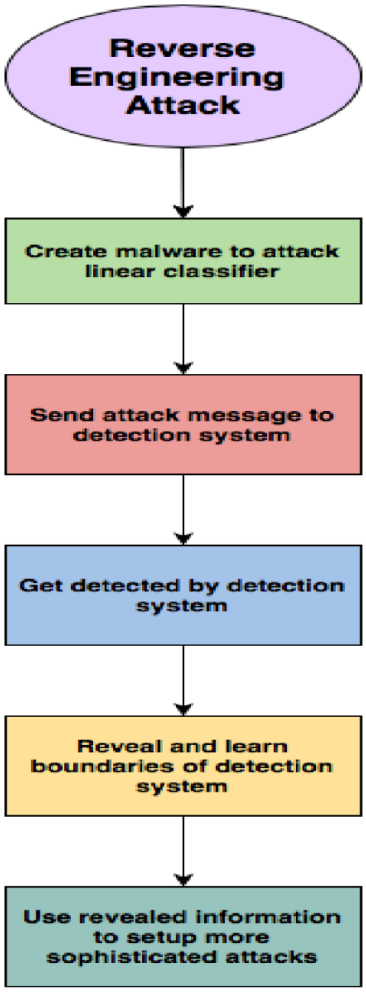


Figure 5 How Reverse Engineering Attack works

Literature Review Cont'd

Evasion Attacks

Evasion attacks are another category of exploratory attacks. The adversary creates an attack message by manipulating the input that will be detected by the classifier in order to produce incorrect outputs. This approach is called the white box. Unlike your ordinary message, this altered message can evade the detection system [22]. For this to be effective, the adversaries have a limit on how much they can disguise their message where it will be detected. If the message is camouflaged too much, the classifier might not detect it [22].

On the other hand, if the message is not disguised properly, the classifier will figure out that it is a much stronger than it shows, for example a spam message. In other words, the intruder uses malware samples at test time to have them incorrectly classified as benign by the classifier. This can bring a huge risk to the safety of the system. Evasion attacks have proven to be successful and effective making it the most popular kind of attack as shown in figure 6.

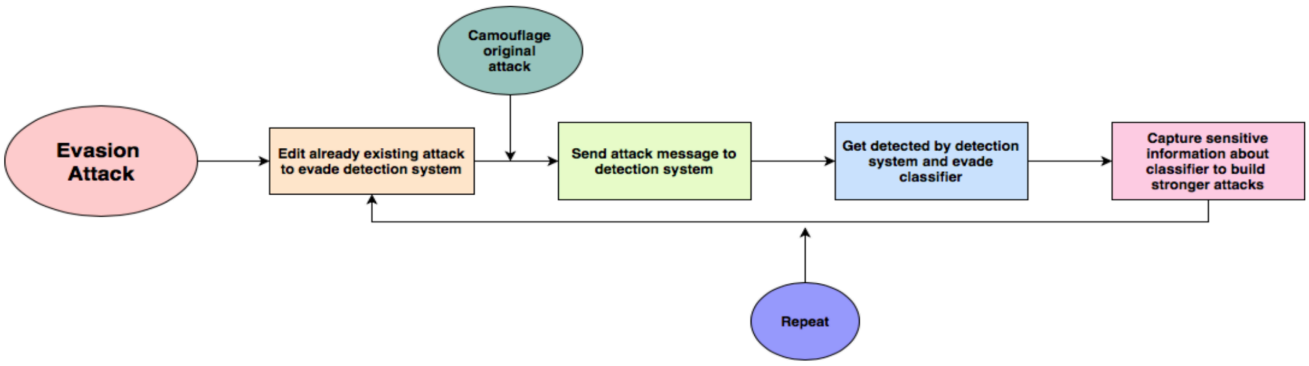


Figure 6 The workflow of Evasion Attack

In the article, "Yes, Machine Learning can be more secure! A case study on Android Malware Detection", it mentions different evasion scenarios. They are distinguished based on how much information the intruder knows about the system. The more they know the easier it is to evade with the chance to have less manipulation for malware samples. The first scenario is with the Zero-Effort Attack and this includes malware data that is neither obfuscated nor modified. The second one is with DexGuard-Based Obfuscation Attack. Similar to the first scenario, the intruder is unaware of any knowledge of the attacked system. The intruder evades in this scenario by modifying the classes.dex file with the tool called DexGuard. [6]

The third scenario is with the Mimicry attack. With this, the intruder has a substitute dataset, which includes malware and benign applications and they also know the feature space. The strategy behind this is to change malware applications to be similar to the benign ones. The fourth scenario is with Limited-knowledge (LK) attack in which the intruder is familiar with the learning algorithm and a substitute classifier based on the data that is there. Last but not least, the last scenario is with Perfect-Knowledge (PK) attacks, which consists of the intruder knowing the classifier. [6]

Methodology

Project Goals

The main goal and objective for this project was to use two different types of adversarial attacks in order to create a counter for the rapidly advancing malware attacks on Android devices. The two different types of attack models we used were Evasion Attacks and Data Poisoning Attacks. These two attacks models were used to generate adversarial examples.

In order to generate the examples, we used a machine learning approach. The approach we used for machine learning is to train the classifiers with an existing dataset. For both of the attacks we used the Drebin dataset. After training the classifiers we learned the patterns on how the attacks behave and how well the classifier performs for both.

Why we chose Evasion and Poisoning Attacks?

We chose Data Poisoning and Evasion attacks because these attacks have been successful and efficient in attacking Android devices. Data Poisoning attacks can be implemented in multiple different ways such as label modifications, data injection, data modification and logic corruption. In this project we implemented Data Poisoning by randomly flipping the labels for permissions. Evasion attacks are very complicated attacks and very detailed. We implemented evasion attacks by using Android HIV's mutated dataset and attack model. With access to their research we were able to generate adversarial samples to achieve results.

Workflow

Original Malware Dataset:
Utilize Drebin malware dataset to generate adversarial examples.

Evasion Attack Model:
Utilized an existing evasion attack model provided by the Android HIV authors.

Data Poisoning Attack Model:
Generate new adversarial examples using label modification.

Machine Learning Classifier:
Use support vector machine to build the classifier and perform analysis on the adversarial examples.

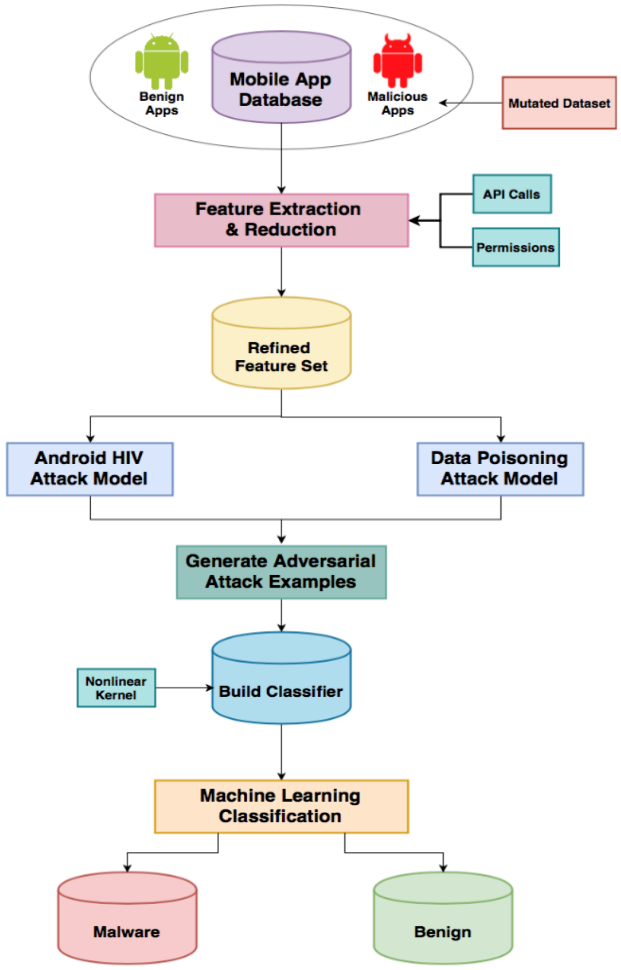


Figure 7 An overview of system architecture

Implementation

Dataset

For our project, we used the Drebin dataset which contains over 5,000 malware application. Drebin is a widely accepted malware dataset for detection research. We utilized the Drebin dataset to generate new adversarial examples using a Data Poisoning Attack algorithm. Android HIV also worked with Drebin as one of their main datasets in order to generate new adversarial examples through their Evasion Attack Model. Therefore, working with the same malware dataset allowed us to obtain consistent results and perform more accurate analysis.

Data Poisoning Attack

Reveal Droid

In order to implement the Data Poisoning Attack, extensive research was done to see how others have implemented it. Through our research we found RevealDroid and IagoDroid. Both tools are used to implement a Data Poisoning Attack. IagoDroid is a tool that is used to modify classifiers. This tool is implemented alongside RevealDroid. RevealDroid generates a model and that model is input into IagoDroid to modify and cheat the classifier, therefore implementing the Data Poisoning Attack.

Our Data Poisoning Attack

Although the model provided by RevealDroid would have given us the ideal algorithm for implementing the Data Poisoning Attack, certain issues with the tools lead us to writing our own algorithm instead. This algorithm was developed by using the methodology of RevealDroid as our guideline. By doing so we were able to implement a model that simulates a Data Poisoning Attack, more specifically a Label Modification Attack.

Our algorithm takes feature vectors in the form of a CSV file as an input to run the attack. The attack is run on a 2D array that contains the feature vectors from the CSV file. To create this 2D array, our algorithm first reads each line in the CSV and adds it into a String array. Each index in that array represents a different APK. The value of that index represents all of the features as a single string. In order to run the model, the features need to be read separately. We did this by creating a loop that would run through the entire String array of APKs created previously. In this loop we created a temporary String array. For each APK, we took the single string of features, split it by comma, and added it into the temporary array. By doing this we ensure that the features have now been separated and can be read as such. We then go through the current APK's feature array and add the value of each feature into the 2D feature space array. This is done for all of the APKs, until our feature space array has been filled and matches the values in the CSV file. This is depicted in fig 8.

```
/*This loop separates the permissions for all of the APKs, parses it as an integer, and adds it to the 2D array*/
for(int i = 0; i < numAPKS; i++)
{
    String[] currAPK = temp[i].split(","); //split the single feature string as separate features
    String label = currAPK[currAPK.length - 1]; //the class of the APK is the last comma separated value
    for(int j = 0; j < currAPK.length - 1; j++)
    {
        if(label.equals("Malware"))
        {
            int vectorVal = Integer.parseInt(currAPK[j]);
            vectors[i][j] = vectorVal;
        }
    }
}
```

Figure 8 Permission Separator

Implementation Cont'd

Once all of that has been done, the algorithm runs the attack on the features. As mentioned before, we developed a Label Flipping Attack. As shown in Figure 9, in this attack, the label of a random number of features is flipped, thus mutating the APK's features. To run this attack, the algorithm goes through the feature space. For each APK, it selects a random feature and if that feature hasn't been mutated yet, the attack flips the label. If the selected feature has already been mutated, it selects another feature until it finds one that hasn't been mutated yet. For our attack model, we chose to mutate 60 features.

```
for(int i = 0; i < numAPKS; i++)
{
    //This loop is used to recursively flip 60 permissions
    for(int j = 0; j <= 60; j++)
    {
        randCol = (int) (Math.random() *(double) numPermissions); //Choose a random permission (column)

        //If the permission hasn't been flipped yet, flip it -> if a 1 change to 0, if a 0 change to 1
        if(!(visited.contains(randCol)))
        {
            int value = vectorSpace[i][randCol];

            if(value == 0)
            {
                vectorSpace[i][randCol] = 1;
            } else
            {
                vectorSpace[i][randCol] = 0;
            }

            visited.add(randCol); //add the flipped permission into the list
        } else
        {
            //Choose another permission if the previous one has already been flipped
            randCol = (int) (Math.random() *(double) numPermissions);
            continue;
        }
    }
}
```

Figure 9: Flipping Algorithm

After the attack is run on all of the APKs, the resulting vectors are output into a new CSV file as shown in Figure 5.3.

```
//Go through the mutated feature vector array and print out each of the vectors into a new CSV file.
PrintWriter writer = new PrintWriter("mutatedSampleVectors.csv", "UTF-8");

for(int i = 0; i < vectorSpace.length; i++)
{
    for(int j = 0; j < vectorSpace[i].length; j++)
    {
        writer.print(vectorSpace[i][j] + ",");
    }
    writer.print("Malware");
    writer.println();
}
```

Figure 10: Resulting vectors from flipping algorithm

Implementation Cont'd

Evasion Attack

Based on the Android HIV research paper, we were able to get access to their mutated dataset. We were able to use the attack model to generate adversarial samples using the Drebin dataset.

Extracting Mutated Vectors

The Android HIV vectors were provided in the form of an NPZ file. We used Python to convert NPZ format to TXT file in order to gain access to the feature vectors. We first do this by importing the load function from the numpy library. The load function is used to return the input array from NPZ file. We load the data into a variable and then recursively print the contents into an open text file as shown in Figure 11. The resulting feature vectors contains about 45,476 features and 10 instances.

```
import sys

import numpy
from numpy import load
numpy.set_printoptions(threshold=sys.maxsize)

data = load('C:/Users/Aemun/Documents/Python Projects/AndroidHIV/
lst = data.files

text_file = open("NPZ Output/Attack Output", "w")

for item in lst:
    print(item)
    print(data[item])

    listToStr = ' '.join([str(elem) for elem in data[item]])

    text_file.write(item + "\n")
    text_file.write(listToStr + "\n")

text_file.close()
```

Figure 11: NPZ Extractor

Implementation Cont'd

Classifier

Support Vector Machine (SVM)

For our classifier we used a Machine Learning based algorithm called the Support Vector Machine. An SVM allows the data to be separated into distinct classes. For our classifier, we had two classes named OriginalMalware and Benign. Depending on the type of dataset we were training or testing the classifier, the classes allow the data to be processed and evaluated.

In general, the support vector machine develops an optimal hyperplane that builds a maximum margin between the classes we are working with. The data points that are closest to the hyperplane of the support vectors which balances the position of the hyperplane.

For this project, we used an open source machine learning software called Weka. In order to build an SVM classifier in Weka, we installed an external tool called LibSVM. Using the LibSVM we were able to build a classifier using different kernel types such as linear and nonlinear kernel functions. Linear kernel type is used for supervised learning of the dataset and nonlinear kernel type is used for unsupervised learning of the dataset.

In order to feed a dataset to a classifier, each application is represented in a form of a vector in order to be read and analyzed correctly. For our dataset, we formatted our feature vectors into ARFF files which is the recommended file for Weka. An ARFF file is formatted by first declaring all the attributes or the number of features that we are working with in the dataset as shown in Figure 12.

Implementation Cont’d


```
@attribute 177 numeric
@attribute 178 numeric
@attribute 179 numeric
@attribute 180 numeric
@attribute 181 numeric
@attribute 182 numeric

@attribute class{OriginalMalware, Benign}

@data

1,1,1,1,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,OriginalMalware
1,0,1,0,0,0,1,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,Benign
```

Figure 12 ARFF File Format

Furthermore, we declare our classes in which the classifier will be working with followed by the feature vectors for each application. As seen in Figure 12, at the end of each vector we declare the name of class in which the application belongs to. The classifier will analyze the vector and predict which class each application belongs to.

The classifier will analyze the vector to predict which class each application belongs to. These predictions are made based on the training set. Once the classifier gives its predictions, they will look at which class each application really belongs to based on the declared class in the ARFF file. This will allow the classifier to give an output of correctly classified instances and incorrectly classified instances.

Training Dataset Results

To train the classifier we used the Drebin dataset with non-mutated malware samples and benign samples. In order for the classifier to process the applications, we formatted the malicious and benign apps into feature vectors. We used the feature permissions as columns and the applications as our rows.

In our feature vector we specify which class each application belongs to. For our project, we worked with two classes named OriginalMalware and Benign. For the training dataset, the OriginalMalware class represents non-mutated malware applications while the Benign class represents non-malicious applications.

```
=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         164               95.9064 %
Incorrectly Classified Instances         7                4.0936 %
Kappa statistic                          0.9181
Mean absolute error                      0.0409
Root mean squared error                  0.2023
Relative absolute error                  8.1847 %
Root relative squared error             40.4516 %
Total Number of Instances              171

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 0.929    0.012    0.988      0.929    0.958      0.920    0.959     0.953     OriginalMalware
                 0.988    0.071    0.934      0.988    0.960      0.920    0.959     0.929     Benign
Weighted Avg.    0.959    0.041    0.961      0.959    0.959      0.920    0.959     0.941

=== Confusion Matrix ===

  a  b   <-- classified as
 79  6 |  a = OriginalMalware
  1 85 |  b = Benign
```

Figure 13: Original Training Dataset Classification Results

Results Cont'd

Testing Dataset Results

Evasion Attack

After training the classifier, we did testing with the Evasion Attack vectors first. As mentioned before, for Evasion Attack we used the Android HIV mutated feature vectors. The testing results are shown in Figure 14. As seen in the figure, the percentage for Correctly Classified Instances is 0% while the percentage for Incorrectly Classified Instances is 100%.

In Weka, the higher the percentage of Correctly Classified Instances, the more accurate the classifier is. If the percentage for Incorrectly Classified Instances is higher than that of the Correctly Classified Instances, then that means that the mutated APKs were able to successfully deceive the classifier. In the case of the Android HIV vectors, the classifier was deceived 100% of the time.

```
=== Summary ===

Correctly Classified Instances           0               0      %
Incorrectly Classified Instances        10             100      %
Kappa statistic                         -1
Mean absolute error                      1
Root mean squared error                  1
Relative absolute error                183.3333 %
Root relative squared error            183.3333 %
Total Number of Instances               10

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 0.000    1.000    0.000      0.000    0.000      -1.000   0.000     0.500     OriginalMalware
                 0.000    1.000    0.000      0.000    0.000      -1.000   0.000     0.500     Benign
Weighted Avg.    0.000    1.000    0.000      0.000    0.000      -1.000   0.000     0.500

=== Confusion Matrix ===

 a b   <-- classified as
 0 5 | a = OriginalMalware
 5 0 | b = Benign
```

Figure 14: Evasion Attack Classification Results

Results Cont'd

Data Poisoning Attack

Once testing on the Evasion Attack vectors was completed, we did testing on the Data Poisoning Attack vectors. As mentioned earlier, the vectors for this attack were generated by running our Label Flipping algorithm on the original Drebin dataset.

As shown in Figure 15, the percentage of Correctly Classified Instances is about 75% while the percentage for Incorrectly Classified is about 25%. For this attack, the classifier was able to correctly identify the mutated APKs more often than the Evasion attack.

```
=== Evaluation on training set ===

Time taken to test model on training data: 0.01 seconds

=== Summary ===

Correctly Classified Instances         128               74.8538 %
Incorrectly Classified Instances        43               25.1462 %
Kappa statistic                          0.4967
Mean absolute error                      0.2515
Root mean squared error                  0.5015
Relative absolute error                 50.2941 %
Root relative squared error            100.2937 %
Total Number of Instances              171

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 0.682    0.186    0.784      0.682    0.730      0.501    0.748     0.693     OriginalMalware
                 0.814    0.318    0.722      0.814    0.765      0.501    0.748     0.681     Benign
Weighted Avg.    0.749    0.252    0.753      0.749    0.747      0.501    0.748     0.687

=== Confusion Matrix ===

  a  b   <-- classified as
 58 27 |  a = OriginalMalware
 16 70 |  b = Benign
```

Figure 15: Data Poisoning Attack Classification Results

Analysis

For the analysis, we looked at the classifier's response to the adversarial vector. We found that the confidence score of classifier was 100%. This means that the purpose of adversarial attack was successfully done. In Figure 16, we can see that 5 instances are actually malware; however, predictions show that each application got classified as Benign or non-malicious because the applications were mutated. The prediction number is the confidence score and for each application the score is 1 or 100%. This means that the classifier is 100% confident that the malicious applications are non-malicious.

analyzeResults

inst#	actual	predicted	error	prediction
1	1:OriginalMalware	2:Benign	+	1
1	1:OriginalMalware	2:Benign	+	1
1	1:OriginalMalware	2:Benign	+	1
1	1:OriginalMalware	2:Benign	+	1
1	1:OriginalMalware	2:Benign	+	1

Figure 16: Confusion Analysis

We also looked at the confidence score for security measure. In order to ensure the confidence of the adversarial attack, we further analyze the results based on the testing. In Figure 17, we can see that Benign represents the mutated application and the predicted class is malware. Therefore, this analysis, gives an insight that the mutated applications that are trying to confuse the classifier by pretending to be benign applications. In this case, we see that almost all applications that are considered to be Benign are predicted to be malicious. The confidence score for most of the applications are quite high. However, application 171 passed the test and still remained to be Benign, but the confidence score was only about 65%.

analyzeResults

inst#	actual	predicted	error	prediction
163	2:Benign	1:OriginalMalware	+	0.914
164	2:Benign	1:OriginalMalware	+	0.888
165	2:Benign	1:OriginalMalware	+	0.539
166	2:Benign	1:OriginalMalware	+	0.974
170	2:Benign	1:OriginalMalware	+	0.914
171	2:Benign	2:Benign		0.651

Figure 17: Security Measure Analysis

Conclusion

In this paper we went over the different types of attacks we used to conduct our research and achieve results. Training the classifiers strengthened the classifier to segregate malicious and benign applications. Android attacks have been becoming more common and now that we understand how the adversaries execute the attacks, we can create a solution to solve this issue.

Future Works

Due to time limitations we couldn't develop counter measures. One possible approach that can work is counter featurization. Counter featurization is an approach that whose objective is to reduce training time on data that contains categorical features. This is done by feeding the learning algorithm with limited subset of the collected data. The general goal and purpose for this approach is to featurize the data with the conditional probability of the class given the frequency. The frequency would be the feature value that is observed with each class, as opposed to directly using the value of a categorical feature. This technique can possibly lead to a solution of fighting against adversarial attacks by making the detection system much more accurate.

Another possible approach can be Secure Linear Classification. In this approach, the goal and objective is to improve the security of the linear classification system. This is done by enforcing learning at a more evenly distributed level of feature weights. By doing this, the attacks would be required to manipulate more features to evade the Android devices. There can always be many more possibilities to stop these attacks. The research to stop these attacks needs to keep moving forward because the attacks are getting more sophisticated. An abstract idea that is possible in the near future would be to use the attack models in Deep Neural Networks and Deep Learning. This could have different results and can possibly much more efficient in stopping these attacks.

Technical Limitations

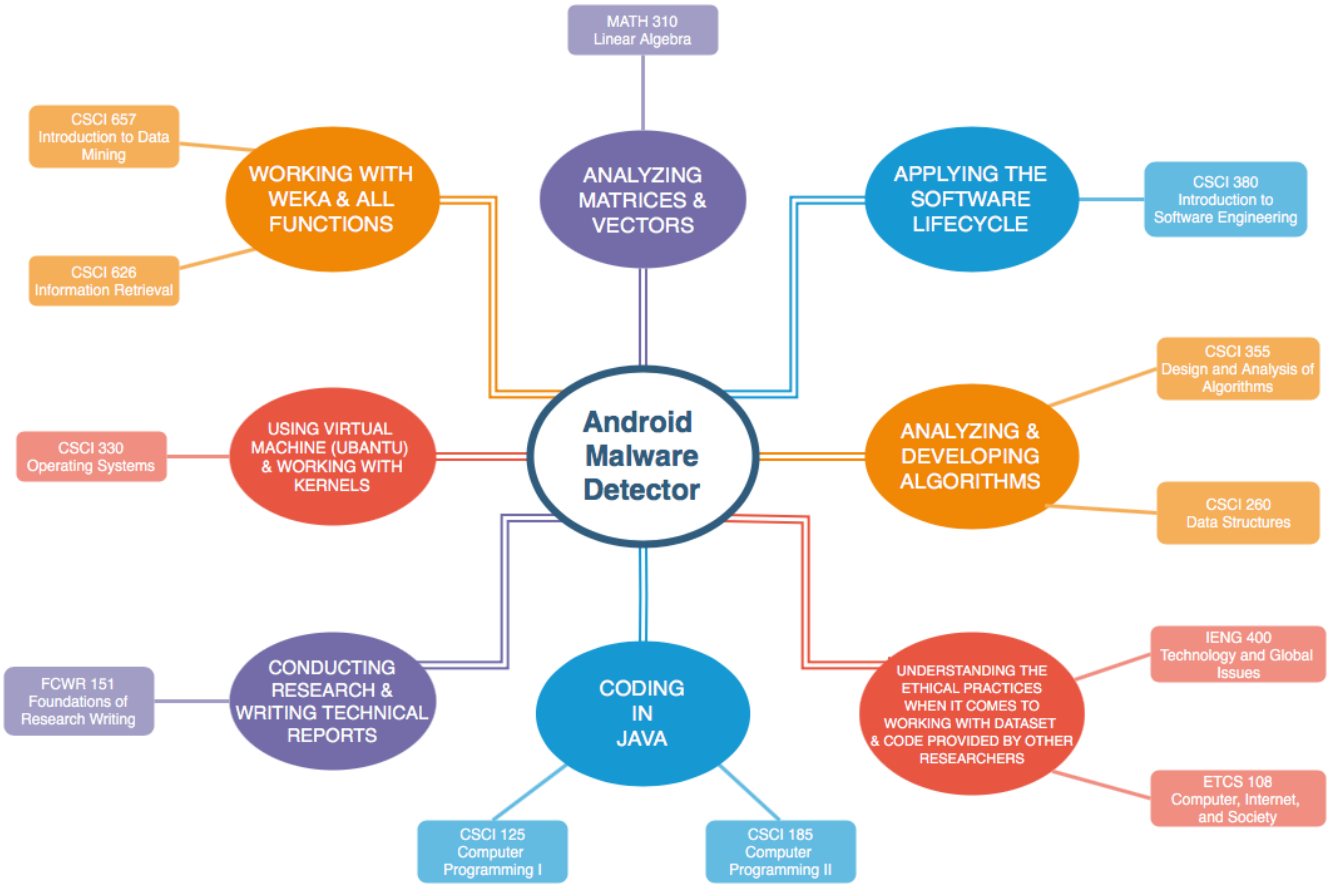
In the project, we decided to work with Evasion Adversarial Attacks. In order to implement an Evasion Attack, we need to mutate all malware applications we decide to work with. To mutate an application, we need to ensure the following: the application is still malicious, has stable functionality after mutation, and the application does not crash after mutation. These aspects are crucial in order to make sure the Evasion Attack is effective. However, in order to meet the mutation criteria for Evasion Attack, the process would have been very time consuming. Therefore, we were not able to implement our own Evasion Attack. However, we were successful to implement our own Data Poisoning Attack because label flipping is not directly mutating the application itself.

Conclusion cont'd

Ethical Practices

Throughout the project, we worked with an open source drebin dataset and generated our own Adversarial Attacks using our Data Poisoning algorithm. However, in this project, one of the datasets we worked with was provided by the authors of the research paper "Android HIV: A Study of Repackaging Malware for Evading Machine-Learning Detection". The authors allowed us to utilize their mutated vectors and their code to run an Adversarial Attack in order to generate new Adversarial Examples. In terms of ethical practices, we understand that the dataset and code provided by the authors of Android HIV are private and should only be used for research purposes. We understand that we should not share their dataset and code with anyone outside of NYIT Department of Computer Science. In terms of using the Evasion Attack mutated vectors, we give complete acknowledgement and credit to the original authors.

Class Outline



Team Contributions

Neha Bala

Attribute generator code, Arff files, train the classifier, test the train dataset run algorithms on classifier, generate results, analysis

Aemun Ahmar

Decode NPZ file vectors, got evasion attack to work, implemented data poisoning algorithm, analysis

Fernanda Tovar

Developed data poisoning label flipping algorithm, website, weekly reports, Final Report

Prachi Bambarkar

Worked to implement data poisoning via RevealDroid tool, literature review algorithms

Arpit Battu

Worked to implement data poisoning via RevealDroid tool, literature review algorithms

Acknowledgements

Authors of Android HIV Research Paper

Thank you to Dr. Xiao Chen, Dr. Chaoran Li, Dr. Jun Zhang, and Dr. Sheng Wen for giving us the dataset that we implemented in our project.

References

[1] A. Holst, “Smartphone users worldwide 2016-2021.” https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/. Accessed: 2019-10-30.

[2] A. Holst, “Global mobile os market share 2009-2018, by quarter.” https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/. Accessed: 2019-10-30.

[3] C. Lueg, “Cyber attacks on android devices on the rise.” https://www.gdatasoftware.com/blog/2018/11/31255-cyber-attacks-on-android-devices-on-the-rise. Accessed: 2019-10-30.

[4] Y. Zhou, “Dissecting android malware: Characterization and evolution,” 2012 IEEE Symposium on Security and Privacy, May 2012.

[5] M. Ali, D. Svetinovic, Z. Aung, and S. Lukman, “Malware detection in android mobile platform using machine learning algorithms,” 2017 International Conference on Infocom Technologies and Unmanned Systems (Trends and Future Directions) (ICTUS), December 2017.

[6] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, and F. Roli, “Yes, machine learning can be more secure! a case study on android malware detection,” IEEE Transactions on Dependable and Secure Computing, vol. 16, pp. 711 – 724, July-August 2019.

[7] J. Sahs and L. Khan, “A machine learning approach to android malware detection,” 2012 European Intelligence and Security Informatics Conference, August 2012.

[8] C. Gibler, J. Crussell, J. Erickson, and H. Chen, “Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale,” in Proceedings of the 5th International Conference on Trust and Trustworthy Computing, TRUST’12, (Berlin, Heidelberg), pp. 291–307, Springer-Verlag, 2012.

[9] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, “Riskranker: Scalable and accurate zero-day android malware detection,” in Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services, MobiSys ’12, (New York, NY, USA), pp. 281–294, ACM, 2012.

[10] L. Chen, S. Hou, and Y. Ye, “Securedroid: Enhancing security of machine learning-based detection against adversarial android malware attacks,” 33rd Annual Computer Security Applications Conference, pp. 362–372, April 2017.

[11] L. Chen, S. Hou, Y. Ye, and S. Xu, “Droideye: Fortifying security of learning-based classifier against adversarial android malware attacks,” 2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining, pp. 782–789, October 2018.

[12] L. K. Yan and H. Yin, “Droidscope: Seamlessly reconstructing the {OS} and dalvik semantic views for dynamic android malware analysis,” in Proceedings of the 21st USENIX Security

References

Symposium
(USENIX Security
12), pp. 569–584, 2012.

[13] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra, “Madam: a multi-level anomaly detector for android malware,” in International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security, pp. 240–253, Springer, 2012.

[14] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, “Drebin: Effective and explainable detection of android malware in your pocket.,” in Ndss, vol. 14, pp. 23–26, 2014.

[15] W. Li, J. Ge, and G. Dai, “Detecting malware for android platform: An svm-based approach,” in 2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing, pp. 464–469, Nov 2015.

[16] K. Zhao, D. Zhang, X. Su, and W. Li, “Fest: A feature extraction and selection tool for android malware detection,” in 2015 IEEE Symposium on Computers and Communication (ISCC), pp. 714–720, July 2015.

[17] Z. Wang, J. Cai, S. Cheng, and W. Li, “Droiddeeplearner: Identifying android malware using deep learning,” in 2016 IEEE 37th Sarnoff Symposium, pp. 160–165, Sep. 2016.

[18] X. Su, D. Zhang, W. Li, and K. Zhao, “A deep learning approach to android malware feature learning and detection,” in 2016 IEEE Trustcom/BigDataSE/ISPA, pp. 244–251, Aug 2016.

[19] W. Li, Z. Wang, J. Cai, and S. Cheng, “An android malware detection approach using weight-adjusted deep learning,” in 2018 International Conference on Computing, Networking and Communications (ICNC), pp. 437–441, March 2018.

[20] M. Kermani, S. Kolay, A. Raghunathan, and N. Jha, “Systematic poisoning attacks on and defenses for machine learning in healthcare,” IEEE Journal of Biomedical and Health Informatics, vol. 19, pp. 1893 – 1905, November 2015.

[21] M. Jagielski, A. Oprea, B. Biggio, C. Liu, C. Rotaru, and B. Li, “Manipulating machine learning: Poisoning attacks and countermeasures for regression learning,” 2018 IEEE Symposium on Security and Privacy (SP), vol. 1804.00308v1, pp. 19–35, April 2018.

[22] I. Alabdulmohsin, X. Gao, and X. Zhang, “Adding robustness to support vector machines against adversarial reverse engineering,” pp. 231–240, November 2014

References

[23] W. Yang, D. Kong, T. Xie, and C. Gunter, “Malware detection in adversarial settings: Exploiting feature evolutions and confusions in android apps,” 33rd Annual Computer Security Applications Conference, pp. 288–302, December 2017.

[24] X. Chen, C. Li, D. Wang, S. Wen, J. Zhang, S. Nepal, Y. Xiang, and K. Ren, “Android hiv: A study of repackaging malware for evading machine-learning detection,” IEEE Transactions on Information Forensics and Security, vol. 15, pp. 987 – 1001, July 2019.

[25] C. Kim, “Ntmaldetect: A machine learning approach to malware detection using native api system calls,” ArXiv, vol. abs/1802.05412, May 2018.

[26] P. Russu, A. Demontis, B. Biggio, G. Fumera, and F. Roli, “Secure kernel machines against evasion attacks,” 2016 ACM Workshop on Artificial Intelligence and Security, pp. 59–69, October 2016