# Mitigating Malicious Adversaries Evasion Attacks in Industrial Internet of Things

Husnain Rafiq, Nauman Aslam, Usman Ahmed, and Jerry Chun-Wei Lin*

***Abstract*—With advanced 5G/6G networks, data-driven interconnected devices will increase exponentially. As a result, the Industrial Internet of Things (IIoT) requires data secure information extraction to apply digital services, medical diagnoses and financial forecasting. This introduction of high-speed network mobile applications will also adapt. As a consequence, the scale and complexity of Android malware are rising. Detection of malware classification vulnerable to attacks. A fabricate feature can force misclassification to produce the desired output. This study proposes a subset feature selection method to evade fabricated attacks in the IIOT environment. The method extracts application-aware features from a single android application to train an independent classification model. Ensemble-based learning is then used to train the distinct classification models. Finally, the collaborative ML classifier makes independent decisions to fight against adversarial evasion attacks. We compare and evaluate the benchmark Android malware dataset. The proposed method achieved 91% accuracy with 14 fabricated input features.**

***Index Terms*—Industrial Internet of Things (IIoT), adversarial attacks, android, malware.**

## I. Introduction

The Industrial Internet of Things (IIoT), with the combination of a 5G/6G network, will be able to connect over trillion devices. As a result, tremendous data will flow from the mobile network [1]. This results in the IIOT based application-oriented digital application, i.e., medical diagnosis and financial forecasting. Smartphones have become an indispensable part of our lives in recent years, being used in virtually every area, including banking, social networking and shopping. Android systems are believed to have captured 87.5 percent of the cell phone market, but malware within legitimate apps is also spreading at an exponential rate [2]. "Malware" is a term that refers to malicious code developed with a dangerous intent and often offered for use in mobile app stores under the guise of a regular and safe program. They are injected or downloaded by users and installed on mobile devices unnoticed. They come in various forms, including viruses, Trojan horses and worms. According to a recent report, there are about one million Android mobile device apps infected with malware [3]. Another frightening fact revealed in a survey is that the financial costs associated with these malware apps reach $400$ billion per year [3]. This data shows how vital malware app detection systems are.

Existing approaches to malware identification lack the analysis and accuracy effect of combining URL, email, IP, and text features with application permissions, intents, and API calls features [1, 4]. The proposed solution is a hybrid technique based on both static and dynamic feature sets. The first part of the technique analyzes the manifest files to extract applications' permissions and intents. These features use by our static classifier to identify potential malware applications automatically. The next phase of our proposed technique analyses the application's behaviour on runtime along with applications' dynamic features. It gives the features as an input vector to the classifier for applications class identification, i.e., malware or not malware. This research is beneficial to the research and industrial community to analyze such dynamic features which have not yet been used to identify malware. These features, along with others, can help identify the ever-changing wide range of malware. The use of classification will help in identifying a wide variety and latest malware that traditional approaches are unable to identify [5].

With the advances in machine learning-based techniques over the past decade, the academic community has shown a strong interest in applying them to Android malware detection [5]–[7]. Through static or dynamic analysis, researchers have identified several characteristics of Android apps in most previous studies [2, 8]. Multiple data such as APIs, permissions, intents, and network addresses can be retrieved from an Android APK file and integrated into a single feature vector space to categorize dangerous and benign apps using machine learning. Surprisingly, these systems can be easily manipulated using malicious examples, i.e., intentionally generated input examples to mislead the detection model during the testing phase. This is challenging because machine learning theory assumes that the training dataset used in the learning phase remains representative of the problem domain and that no intentional dangerous modification of the data occurs [9]. The techniques employed to fool the underlying ML models by providing a tampered input fall under the umbrella of adversarial ML. Adversarial attacks in ML can be classified into two major categories [10] (1) evasion attacks; (2) poisoning attacks. Evasion attacks are performed when an attacker carefully fabricates a malicious input. The underlying model miss-classifies it as a legitimate sample. At the same time, poisoning attacks are performed in the training phase when an attacker manipulates the training data with carefully crafted samples to compromise the whole learning process eventually.

The purpose of this study is to address adversarial evasion

H. Rafiq and N. Aslam are with the Department of Computer and Information Sciences, Northumbria University, United Kingdom. Email: husnain.rafiq@northumbria.ac.uk, nauman.aslam@northumbria.ac.uk

U. Ahmed and J. C. W Lin are with the Department of Computer Science, Electrical Engineering and Mathematical Sciences, Western Norway University of Applied Sciences, 5063, Bergen, Norway. Email: usman.ahmed@hvl.no, jerrylin@ieee.org. Website: http://ikelab.net. (*Corresponding author: Jerry Chun-Wei Lin)

attacks. Thus, the primary contributions of this work are as follows:

1) We provide a unique and scalable countermeasure against adversarial evasion attacks on Android malware classifiers based on machine learning. It uses a collection of classifiers based on machine learning. To prevent evasion attacks, each classifier in the model is trained on a separate subset of distinguishing features.
2) We discover and evaluate the best discriminating subsets of malware detection features collected from Android applications. We create semantic subsets of the original feature vector and rank them according to their detection accuracy. We use the most advanced machine learning-based classifiers with the optimal hyperparameter values. Finally, the model is trained using the discriminative feature subsets found.
3) We evade DREBIN [6], one of the mainstream Android malware classifiers, to present the crucial concern about the fragility of ML-based classifiers. Consequently, we perform an empirical case study to present the effectiveness of the proposed model against such evasion attacks.

## II. Related work

Android security issues, particularly malware detection in legitimate applications, have been a popular area of study due to the exponential increase in smartphone users worldwide. Numerous malware detection methods have been developed, each with advantages and disadvantages. These strategies use static features, dynamic features, or a combination of both.

Static techniques analyze the applications statically without running them and studying their behaviors. However, dynamic techniques are somewhat capable of recognizing new malware as they try to predict them by analyzing their behaviors on run-time. However, they are time and computationally expensive. On the other hand, hybrid approaches can identify a wider range of malware with reasonable accuracy; however, they inherit both static and dynamic techniques limitations. This section gives an overview of the state-of-the-art techniques in this area, distributed under the headings of static, dynamic and hybrid techniques. These and many more such techniques share the same concept of analyzing the application's behaviour on runtime and identifying the application like malware or not malware. Hybrid malware analysis approaches identify malicious apps by combining static and dynamic features. This is a relatively new part of the solution, and several researchers have begun to focus on it. Researchers use static and dynamic feature pools to develop various successful malware identification systems.

The study presents a system to protect linear regression from malicious activities [1]. The proposed method develops a privacy-preserving verified learning technique for linear regression to prevent dishonest cloud server computations and inconsistent user data inputs. They developed a privacy-preserving prediction technique with lightweight verification to prevent malicious clouds from providing inaccurate inference results. HyMalD logically performs static and dynamic analysis simultaneously to identify obfuscated malware [4]. First, it extracts static features of the opcode sequence using a newly created dataset and dynamic features of the API call sequence. HyMalD employs Bi-LSTM and SPP-Net to identify and classify IoT malware. The detection accuracy of HyMalD was 92.5%.

Android Application Sandbox was offered as another hybrid technique that uses (.dex files) for static analysis [11], while low-level information about system interactions is used for dynamic analysis. The static analysis begins by decompiling .dex files into a human-readable format and then examining for suspicious patterns. The dynamic analysis uses low-level facts about the program that arise during its execution in the sandbox environment. As is known, a sandbox environment is used to ensure system analysis security and data security. In dynamic analysis, the approach additionally analyzes the behaviour of an application by generating random events.

Zhao et al. proposed the term AMDetector [12] for a hybrid malware detection approach. The approach uses a modified attack tree model that uses static features to elicit information about an application. The classifier then uses this information to categorize applications as usual or dangerous. In addition, the application behaviour that triggers the various code components of an application is evaluated, which serves as the basis for dynamic analysis. By using structured rules (including attack trees), this approach achieves high code coverage and up to 96.5 per cent accuracy. However, manual rule development and dynamic analysis are time-consuming.

Yuan *et al.* presented another hybrid approach that uses deep learning to classify Android malware using Droid-Sec [13]. The approach extracts over 200 static and dynamic features from an application and feeds them into a deep neural network for classification. Experiments were conducted on 599 applications that contained both malicious and benign samples and had no class imbalance. The approach achieved 96.5 percent accuracy. Another work used different algorithms like naïve Bayes, J48, Random Forest, Multi-class classifier, and multilayer perceptron [14]. The data set included 3258 Samples of Android apps. The multi-class classifier performs better than others regarding the classification accuracy is 99.81%. In terms of computational complexity, the Naïve Bayes classifier proved to be the most efficient in classifying malware datasets.

Alzaylaee *et al.* propose a unique hybrid technique for generating test inputs to improve dynamic analysis on Android devices [15]. The author created a hybrid system by combining a random-based tool (Monkey) with a state-based tool (Droidbox) to detect more dangerous behaviours. The dataset contains 2444 apps, with 1222 benign and 1222 malicious apps. The author evaluates three scenarios, random, State-based and hybrid approaches and checks their performance. The result shows that the hybrid technique improved the number of dynamic feature accuracy over the random base and state base test input methods.

Arora *et al.* discuss the hybrid malware detection technique [16]. The author evaluates both permission and traffic features to detect malware from the sample. The idea is based on supervised and unsupervised learning algorithms (KNN and K-Medoids). The result shows that the hybrid approach gives the 91.98% detection accuracy far better than the dynamic and static accuracies of 81.13% and 71.46%, respectively.

A recent study conducted by Hussain *et al.* uses gradient boosting based supervised machine learning approach for their hybrid malware detection technique [17]. The authors used the consent model associated with the intent of the application in combination with others. The approach works in two phases. The first phase using static analysis, tries to identify malware applications. The candidate applications are marked, and the next phase, using dynamic analysis, tries to confirm whether the suspected applications are malware or not. The authors used two feature selection strategies and conducted a comparative analysis among classifiers to see the best features and classifiers. The authors used 500 benign applications belonging to 28 different categories and 5,774 malware applications belonging to 178 different categories. The results show 96% accuracy in detecting the malware application using a gradient boosting classifier. Though the results are convincing, the dataset malware versus benign applications seems unbalanced and may suffer from a class imbalance problem. Also, the technique is time and computational costly due to confirmation and reconfirmation strategy.

## III. Methodology

This section covers the detailed methodology and workflow of our proposed technique. The basic workflow of our proposed system is composed of three phases, as mentioned in Fig. 1. The first phase is the data acquisition phase, followed by the feature extraction and selection phase and, finally, the classification phase. Detail explanation of each phase is described below.

The designed system will be capable of classifying a wide range of malware, including that found on Android devices. We used the hybrid approach, a mixture of dynamic and static approaches. In static mode, we used android intents and permissions as the essential feature for malware detection, and in the dynamic mode, we used the system call feature for malware classification. In static mode, used four-level detection model consists of Decompiler, Extractor, intelligent learner, and decision-maker. The Decompiler converts an APK file into readable components. Each APK file consists of several components such as Java files, XML files, and a manifest file. Each component has been decoded and made readable.

The extractor module is responsible for extracting various information required for malware detection, such as intentions and permissions. Androguard is used to reverse engineer the Dex file and the gorgeous soup package to determine the permissions and intent of the manifest file. This submodule accepts data from the feature database and learns the data pattern using a Bayesian network technique. The output model is then sent to the decision maker submodule. The decision-maker sub-module is responsible for assessing whether the data is harmful or not. It receives data from the Extractor and Intelligent Learner submodules and the feature database. The decision-maker submodule uses the model to detect the maliciousness of the application. If the output of the static model is an as malicious app, it has been sent directly to the malware classifier database. If the output of the static model is a clean App, it sends to the dynamic mode for further procedure. The dynamic module is used to check the application's behaviour at run time. The benign apps that came from the decision-maker have been again analyzed to find out the application's behaviour at run time. The application is tested in a virtual device called an emulator by using a monkey tool to check all functionalities. I will use the system call feature for dynamic analysis in this research. It has been used to extract system calls. We use the stace tool for recording the system calls. For each system call, we construct a weighted directed graph. Each system call represents by a node. The node size shows system call frequency, and direct edges indicate the sequence of system calls.

Even though people would consider the manufactured samples to be benign, their inclusion in learning models could cause them to behave in different ways that are not intended. Real-world applications of adversarial attacks that succeed in their goal. As a result, researchers in machine learning and cybersecurity are increasingly interested in adversarial attack and defense tactics. The field of adversarial machine learning (ML) encompasses the strategies used to deceive the ML models working in the process by providing a manipulated input. In ML, adversarial attacks fall into two main categories: (1) evasion attacks and (2) poisoning attacks. An attacker performs a circumvention attack when he intentionally fabricates a malicious input so that the underlying model incorrectly identifies it as a valid sample. Poisoning attacks, on the other hand, are performed during the training phase. In this case, the training data is manipulated using carefully constructed samples to eventually subvert the entire learning process. In this study, we used the scenario of a circumvention attack where features are faked to change the input based on the feature analysis. This fabrication leads to mis-classification, which is discussed in Section V-B.

## IV. Feature Extraction

Apk files are used to bundle Android apps. APK is an abbreviation for Android Package Kit. It is a file type used by the Android operating system to provide apps in the android application framework, as shown in Fig. 1. APK files are usually compressed files that can be downloaded directly from the Google Play Store or third-party app stores for Android devices. As seen in Fig. 1 and *(Algorithm 1, Lines 2-3)*, APK files contain several files and directories, including the folder META-INF, the folder res, and the files resource.arc, AndroidManifest.xml, and classes.dex. This information is occasionally maintained in a separate folder called original. The Android manifest.xml file format is a binary XML file. This section contains metadata about the application, such as the application name, version, intents, and permissions. Classes.dex files contain compiled application code index format. We used a Python feature extraction script to extract the features. This feature extraction script splits the APK file into classes.dex and AndroidManifest.xml files extract permissions and intents tags from the AndroidManifest.xml file and save

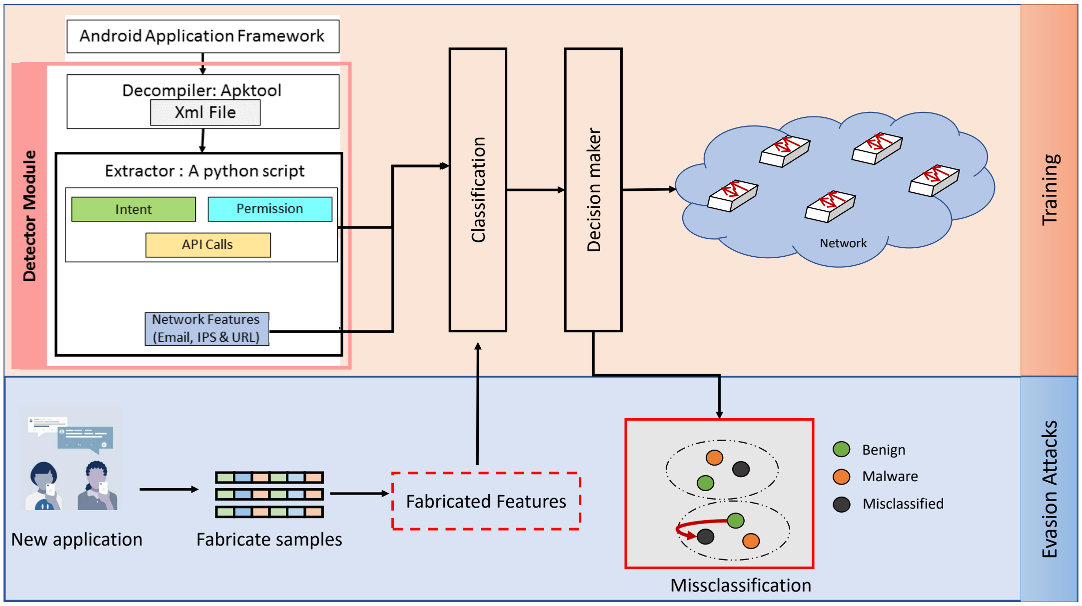


Fig. 1: A workflow of the proposed approach.

them to .txt files. Similarly, API calls and network features (IP addresses, email addresses, and URLs) are collected from the deconstructed dex files and stored in .txt files. These text files are also used to generate feature vectors. The following describes the exact operation of the feature extraction script:

1) Decompile APK into their basic files and directories using APK Tool.
2) In the second step, we obtain the dex files, resource files, and XML files due to APK decompilation.
3) The script takes the AndroidManifest.xml files and reads permissions and intent tags. Extract all permissions and intents and store them into .txt files.
4) For mining API calls, the script takes decompiled dex files. These dex files consist of classes.dex files. Some methods are used in each class. These classes can call these methods.
5) The feature extraction script creates a call graph of classes in which each method is a node. When a method calls another method, it will create an edge to that node. Each node in the call graph constitutes an API call feature.
6) Similarly, the script extracts network features (IP addresses, email addresses, and URL) from dex files by using regular expressions.
7) The extracted API calls and network features are stored in .txt files.

A similar process is repeated for all malware and non-malware APK files in our dataset. The .txt files obtained from the feature extraction process are used for feature vector creation. These are the five types of information we extract from the dataset: permissions, APIs, intentions, hardware components, and network addresses, as mentioned in (Algorithm 1, input). These attributes are derived from the properties of the data collection. Instead of embedding all features into a single non-linear feature vector space, the different types of extracted features are each embedded into their own feature vector space. This is done to improve performance and avoid evasion attacks.

The extracted permission files are then compared to the unique permission list for the apps in the training set. If the extracted permissions match the permission list, the permission feature vector bit is set to 1; if not, it is set to 0 (Algorithm 1, lines 4-10). The same procedure is used to extract intent-based, hardware-based, and API-based characteristics (Algorithm 1, lines 10-24). However, to extract API-based features, this study used the Java source code rather than the Android manifest. In addition, network-based characteristics are retrieved from the Java source code. The IP addresses retrieved from the source code of each app are used as the feature vector. Moreover, malware is labeled as 1 and non-malware is labeled as 0, so it is a binary classification problem. Moreover, the five types of extracted feature subsets (permission, intent, hardware, network, and API) are stored in different repositories for each app in the dataset (Algorithm 1, lines 26-28). Finally, the method outputs five different subsets of features (Algorithm 1, line 29). The returned subset of features is then used by the model selected based on the hyperparameter setting, which is different for each type of feature, as discussed in Section IV-A and mentioned in Table I.

### A. *Model selection*

The most tedious part of ML is to select the correct algorithm and tune the corresponding hyperparameters for a selected algorithm to obtain optimal results. This process can be burdensome and time-intensive brute force search. There are many ML algorithms, and each algorithm has numerous hyperparameters. In this study, we use TPOT [18], an automated machine learning (AutoML) tool to design and optimize machine learning pipelines. TPOT is an AutoML system based on genetic programming that optimizes features

**Algorithm 1** Feature Extraction and Classification Detection

**INPUT:** $APK_{File}$.
**OUTPUT:** Malware or Non-Malware.

**for** $all\ f \in F$ **do** ▷ F is APK folder
  $APK_{File} \leftarrow Open(file)$;
  $manifest_{File}$ , $java_{File} \leftarrow APK\_Tool(APK_{File})$;
  **if** $manifest_{File} == androidmanifest.xml$ **then**
    permission$\leftarrow Get_{(Permission)}(manifest.xml)$;
    **for** $all\ p \in permission$ **do**
      **if** $Permission_{(list)}[i] == p$ **then**
        $Vector_{(Permission)}[\,] \leftarrow 1$;
      **end if**
      $Vector_{(Permission)}[\,] \leftarrow 0$;
    **end for**
    intent$\leftarrow Get_{intent}()$;
    **for** $all\ intent_{(i)} \in intent$ **do**
      **if** $Intent_{(list)}[(i)] == intent_{(i)}$ **then**
        $Vector_{intent}[\,] \leftarrow 1$;
      **end if**
      $Vector_{intent}[\,] \leftarrow 0$;
    **end for**
    network$\leftarrow Get_{networks}(java_{File})$;
    **for** $all\ email, url, ips_{(i)} \in network$ **do**
      $Data_{network}[\,] \leftarrow email, url, ips_{(i)}$;
    **end for**
    $Vector_{network}[\,] \leftarrow TF - IDF(Data_{network})$;
  **end if**
**end for**
$Output(Vector_{Intent}) \leftarrow Classify(Vector_{Intent})$;
$Output(Vector_{(Permission)}) \leftarrow Classify(Vector_{(Permission)})$;
$Output(Vector_{network}) \leftarrow Classify(Vector_{network})$;
**Return** $Vector_{network}, Vector_{(Permission)}, Vector_{Intent}$;

and machine learning models to achieve the best classification results in supervised learning. TPOT integrates all algorithms from the SciKit-Learn package [19], an open-source machine learning toolkit for Python programmers. Thus, each operator in the TPOT library pipeline corresponds to a specific machine learning method for classification, feature preprocessing, or feature selection. Table I, depicts the information about classifiers and corresponding hyperparameters returned by the TPOT library for permissions, intents, API, hardware components and network address-based features.

### *B. Training and Testing*

The model setting and dataset are provided on the link. After selecting the ideal classification model and hyperparameters, we train and test our model on each extracted feature subset repository. We use TPOT to train and evaluate a total of 11,010

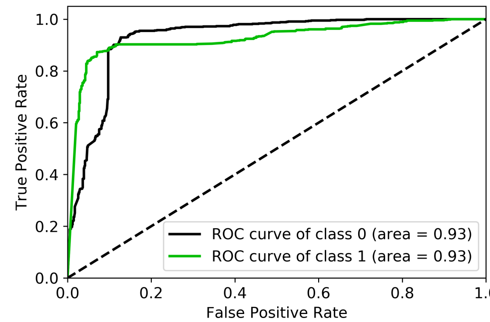


Fig. 2: Permission area under the ROC under different threshold values.

Android applications (5,560 malicious and 5,450 benign) from the Drebin benchmark dataset. We used 70% (i.e., 7,707) of the applications for training purposes and 30% (i.e., 3,303) for testing purposes. However, for the network address class, we could only identify 3,888 out of 5,560 malicious examples with URL-based features. Therefore, we trained and evaluated our model for the class of network addresses on 9,338 samples (3,888 malicious and 5,450 benign). A ROC curve (receiver operating characteristic curve ) is a graph that illustrates the performance of the general classification thresholds of a classification model. A ROC curve compares the TPR to the FPR at different classification levels. As you lower the classification threshold, more objects are classified as positive, increasing both the number of false positives and true positives. We could repeatedly test a model with different classification thresholds to calculate the points on a ROC curve, but that would be inefficient. AUC, an efficient method based on sorting, can give us this information. The area of the ROC curve quantifies the two-dimensional area under the full ROC curve. Area under the curve (AUC) is an aggregate performance metric for the full potential classification thresholds. AUC can be interpreted as the likelihood that a random positive example will be classified higher by the model than a random negative example. AUC is independent of scale. It evaluates the accuracy with which predictions are classified, not their absolute values. AUC is independent of the classification threshold [20]. It evaluates the accuracy of the model's predictions independent of the classification threshold. The AUC is the area under the ROC curve. In general, the higher the AUC value, the better the performance of a classifier for the task at hand.

The output of the tree-based pipeline (TPoT) for the permissions-based features class is shown in Fig. 2. The ROC curve for permissions-based features class is 0.98 for malware (class 1 in the plot) and 0.98 (class 0 in the plot), which signifies the excellent prediction results. According to our classification results, the permission-based class contains the highest discriminative features for malware detection of all static features in the Android App.

In addition, Fig. 3 shows the categorization results for the class of API-based features. The average API ROC curve is 0.96, slightly less accurate than the class of permission-based features. We rank the API-based feature class in the second

TABLE I: TPOT model selection for feature subsets

<table>
<tr><th>Features class</th><th>Classifier</th><th colspan="6">Hyper Tuning</th></tr>
<tr><td rowspan="2">Permissions</td><td rowspan="2">KNeighbors</td><td colspan="2">Number of neighbours</td><td colspan="2">Power parameter</td><td colspan="2">Weights of points</td></tr>
<tr><td colspan="2">59</td><td colspan="2">1</td><td colspan="2">distance</td></tr>
<tr><td rowspan="2">API</td><td rowspan="2">KNeighbors</td><td colspan="2">Number of neighbours</td><td colspan="2">Power parameter</td><td colspan="2">Weights of points</td></tr>
<tr><td colspan="2">47</td><td colspan="2">1</td><td colspan="2">distance</td></tr>
<tr><td rowspan="2">Hardware</td><td rowspan="2">LogisticRegression</td><td colspan="2">Regularization strength</td><td colspan="2">Primal formulation</td><td colspan="2">Penalty</td></tr>
<tr><td colspan="2">5.0</td><td colspan="2">False</td><td colspan="2">12</td></tr>
<tr><td rowspan="2">Intents</td><td rowspan="2">RandomForest</td><td>Bootstrap samples used</td><td>Criterion</td><td>No. of features Considered</td><td>Min. samples for leaf node</td><td>Min. samples required for split</td><td>Number of tress</td></tr>
<tr><td>True</td><td>Entropy</td><td>0.95</td><td>1</td><td>13</td><td>100</td></tr>
<tr><td rowspan="2">Network</td><td rowspan="2">BernoulliNB</td><td colspan="3">Additive smoothing parameter</td><td colspan="3">Class prior probabilities</td></tr>
<tr><td colspan="3">0.001</td><td colspan="3">True</td></tr>
</table>

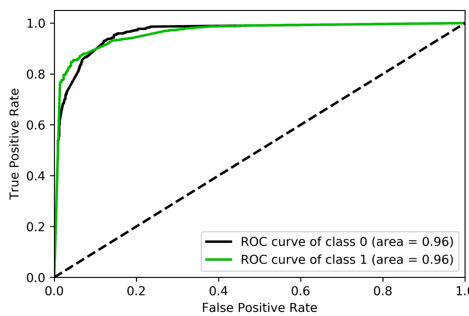


Fig. 3: API area under the ROC under different threshold values.

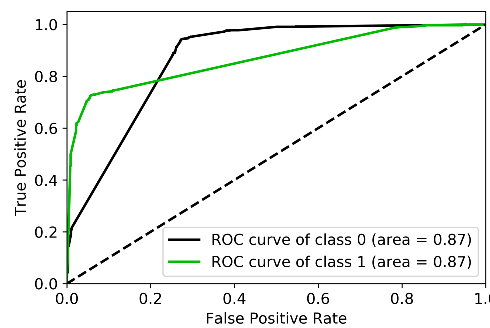


Fig. 5: Intent area under the ROC under different threshold values.

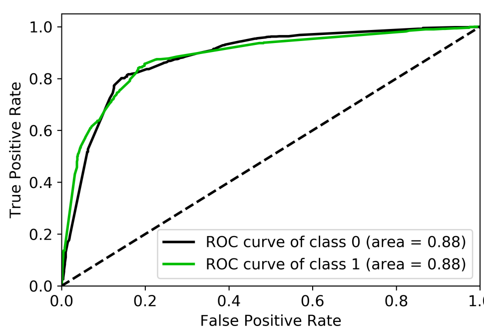


Fig. 4: Hardware area under the ROC under different threshold values.

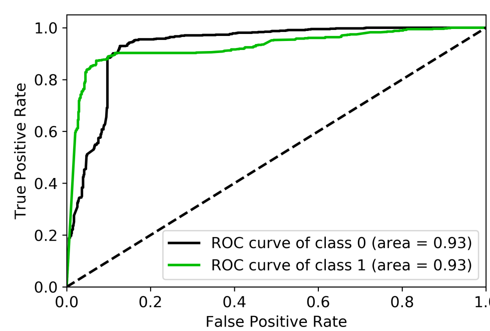


Fig. 6: Network area under the ROC under different threshold values.

position.

Moreover, Fig. 4 shows the classification results for the class of hardware-based features. The average Receiver Operating Characteristic Curve for the class of hardware-based features is 0.89. Our results suggest that the hardware-based features class is ranked third after permissions and API-based features.

Similarly, Fig. 5 shows the classification results for the intent-based feature class. The average receiver operating characteristic curve for the intent-based feature class is 0.88. Our results show that the intent-based feature class performs slightly worse than the hardware-based feature class and ranks fourth.

Finally, Fig. 6 shows the categorization results for features based on network addresses. Even though the average ROC curve is 0.95, we rank the network address-based features fifth. We classified 3,888 malware samples in the class of network address based features out of 5560 malware samples. For 1,672 malicious samples from the Derbin dataset, we could not locate a network address. However, for 3,888 malicious samples containing network addresses, we achieved a high level of accuracy. We placed the network address-based feature class in fifth place, as the feature was missing in 30.1% of the malicious samples.

Table II summarizes the results for all five categories of features (APIs, hardware components, network addresses, permissions, and intents). **Permissions**: Android permissions protect privacy. Before sending SMS or accessing contacts, apps must get the user's consent. **Intents**: Android intents allow app components to interact. Intents pass data between activities. The manifest file lists intents that can be used to

TABLE II: Classification results for feature subsets

| | **Precision** | **Recall** | **F-measure** |
|---|---|---|---|
| Permissions | **0.940** | 0.939 | **0.939** |
| API | 0.852 | 0.928 | 0.888 |
| Hardware | 0.857 | 0.798 | 0.827 |
| Intents | 0.745 | 0.924 | 0.825 |
| Network | 0.854 | **0.954** | 0.901 |

identify malware. **hardware**: AndroidManifest.xml specifies hardware components such as camera, GPS, and touchscreen. Malware may require a specific hardware pattern to perform malicious activities. Therefore, hardware-based features can help in identification. **API calls**: an Android app needs to follow APIs when dealing with other app components, e.g., to send SMS or get the user's location. Android API call patterns can help in malware detection. We use API calls to identify malware. **network addresses**: Malware makes remote connections using IP addresses or domain names. We extract the network address from deconstructed code to create a malware-identifying feature vector.

As can be seen in Table II, classifiers trained on each of these feature sets alone can distinguish dangerous from benign applications. Therefore, we train the proposed model using the four best discriminating feature subsets. Although the fifth feature set, network addresses, has a reasonably high detection rate. However, we still reject it as a component of the proposed system because 30.1% of malicious samples do not have network-based features.

## V. ADVERSARIAL ATTACKS COUNTERMEASURES

This section discusses a method for mitigating malicious evasion attempts in machine learning-based classification models. Moreover, we perform an empirical case study to evade the Drebin classifier by performing adversarial evasion attacks. Finally, we demonstrate the effectiveness of the proposed model in hostile contexts. There are three possible strategies to mitigate machine learning evasion attempts:

1) Using adversarial examples to train the target classifier is called adversarial training.
2) By employee the ensembles of classifiers.
3) Making target classifiers hard to attack.

By using ensemble classifiers and making the target model hard to attack, we focus on options 2 and 3. ML-based classifiers tend to be very fragile in case of evasion attacks. Authors in [21] proposed a prototype tool named Lagodroid to perform evasion attacks on a recent open-source Android malware classifier named RevealDroid [7]. Surprisingly, LagoDroid could perform evasion by modifying just a single feature of the original malicious application. The findings in [21] suggest that a small modification in original malware can result in miss-classification. Therefore, our proposed scalable categorization model can be used to develop a framework that is resistant to adversarial evasion attacks. The learning model includes many classifiers, each trained independently on a subset of data to create an output fork. The proposed model uses four high-level feature subsets (permissions, APIs, intents, and hardware components). Each classifier in the pool is trained individually on each of these subgroups to obtain a label. Finally, the proposed model creates a final label for the observed sample by performing an *OR* operation on the output of each classifier in the pool. If an attacker creates a subset of the application, such as APIs, the classifier trained for that subset will fail. However, the proposed model would detect the malicious App by using the results of other classifiers in the pool trained on different subsets, such as permissions, intents, or hardware-based attributes. Nevertheless, the proposed model would be vulnerable to evasive attacks. However, our method makes it difficult for an attacker to evade. Compared to classical classifiers, such as Drebin [6], an attacker needs to modify the malicious sample more to evade the proposed model. Circumventing the model can be more difficult by including additional classifiers in the pool, each trained on a separate subset of distinguishing features. In the next part, the usefulness of the proposed model in adversarial contexts is demonstrated through an empirical case study.

### A. *Case Study*

Drebin [6], a state-of-the-art classifier for Android malware detection, was evaded as a proof of concept. Drebin is a lightweight on-device malware detector that extracts features from the Android App by performing static analysis. Drebin's collection contains 5,560 malicious and 123,453 benign applications. We also used the same dataset to evaluate the static aspects of an Android application for malware detection. Drebin collects various characteristics of Android apps, such as requested permissions, application components, local API calls, filtered intents, hardware components, used permissions, suspicious API calls, and network addresses. Moreover, all these retrieved features are contained in feature vectors' single multi-dimensional vector space. After feature extraction, Drebin uses linear *Support Vector Machines* (SVM). Drebin achieved an amazing 94% recall on the malware class with only 1% FPR. We replicated Drebin's case study with an identical dataset. We classified malicious and benign applications with linear SVM.

The purpose of this case study is to show how weak a ML-based classifier can be in an adversarial environment and how our proposed model can be incorporated to make the process of evasion more complex for the attacker. Once the attacker knows the underlying classifier and the data on which the classifier was trained (in the best case for an attacker), it is easy to bypass the classifier. An attacker can highlight the top features from the training data based on a particular classifier (linear SVM in Drebin's case) and carefully modify the top features to achieve evasions (Fig. 7, evasion attack block). An attacker can either add a new feature or remove a feature from the existing feature set. Drebin uses a binary feature set where 1 indicates the presence of a feature in the application and 0 indicates the absence of a particular feature. Removing a feature can potentially change the semantics of the malware. Therefore, in this study, we rely only on adding new features in the app, i.e., mutating 0 to 1. As mentioned in the Eq. (1), the method is evaluated based on the evasion rate (the ratio of mis-classified instances after the fabricated input to the total number of instances in the testing set) [20] compared.

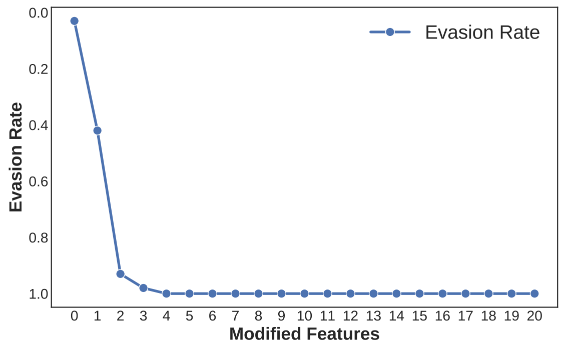


Fig. 7: Evasion attack on Drebin.

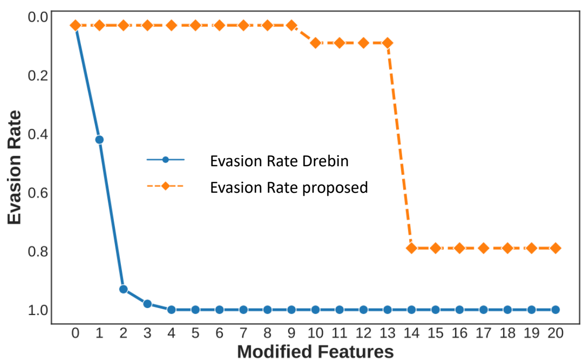


Fig. 8: Performance of proposed in adversarial environment.

$$E_{Rate} = \frac{Malware\ samples\ missclassified}{Total\ Malware\ samples\ in\ Testing\ set} \tag{1}$$

### B. Support vector-based fabricated feature selection

In this study, data points are nonlinearly separable due to the characteristics of these features, i.e., API, network, hardware, intent, and permission. Therefore, malware data that is not linearly separable can be mapped into a higher-dimensional space using the radial-based kernel method, resulting in linear separation of our data. After we completed the fitting of our linear SVM, the proposed model used the trained model to obtain the classification coefficients of the model. The orthogonal vector coordinates are obtained using feature weights orthogonal to the hyperplane. On the other hand, their orientation reflects the class that was predicted. Consequently, the magnitude of these coefficients can be compared to determine the relevance of the features. Thus, by looking at the SVM coefficients, it is possible to determine the characteristic features used in the classification and remove the irrelevant features (which have less variance).

As shown in Fig. 7, by creating three fabricated examples without changing the intended meaning of the malicious entity, an attacker can completely bypass all malicious examples in the Drebin dataset. However, the results of our study suggest that the proposed method has the potential to complicate the attacker's evasion process. It uses a group of classifiers, each of which is trained on its own set of features. We independently classified the vast majority of samples as malicious or benign by identifying five distinct subsets of the most important and distinctive features. As part of our investigation, we modified Drebin. We train the SVM independently on each of the four feature sets, rather than training them on a single integrated feature vector as originally intended (permissions, APIs, intents, and hardware components). Even if the attacker now has access to the data and the target classifier to extract the most relevant features, it will be very difficult for them to find a way around the classifier. This is because all members of a given class within a subgroup are essential features. Consequently, changing a single property can affect the validity of a single class (e.g., permissions). However, in our tests with different subgroups, we were still successful in identifying the virus (e.g., APIs, intents, and hardware components). The evasion attack Drebin is vulnerable to is also included in the proposed model, as you can see in Fig. 8. On the other hand, the proposed classifier can accurately classify malware with 91% accuracy up to 14 different modifications of the unsafe feature vector. With only three modifications to the malicious samples, Drebin was avoided.

### C. Comparison

As can be seen in Table III, [22]–[24, 26], evasion attacks are discussed. Although these strategies achieve considerable evasion rates, the authors have not been able to develop a countermeasure to thwart these attacks. In contrast to these methods, our proposed evasion algorithm was able to bypass the target classifier (Drebin) in 100% of the cases by using three features. As mentioned in the methodology, we also present a countermeasure that can be used to defend against such evasion attacks. As a result, the authors not only avoided target classifiers but also offered strategies to counter such attacks [21, 25]. Grosse *et al.* [25] used deep neural network classifiers to undertake evasion attacks and achieved evasion rates of up to 63% with feature vector perturbations. Grosse *et al.* presented two responses to adversarial circumvention attacks, including distillation and classifier retraining. However, neither of the recommended defenses produced promising results against evasive threats, with a peak detection rate of 33% when the classifier was retrained. In addition, LagoDroid [21] evaded a newer classifier called RevealDroid with a evasion rate of 97%. To prevent evasion attempts against RevealDroid, a countermeasure called RevealDroid [7] is proposed. RevealDroid* works well with few changes, but its performance degrades with more changes. Moreover, RevealDroid* requires multiple ensemble classifiers to detect possible evasion. In their experiments, the authors used 16 decision tree-based classifiers. Using an ensemble of four SVM-based classifiers, we achieved a high detection rate of up to 14 changes in the actual feature vector.

## VI. Conclusion and Future Work

In a 5G/6G powered network, large scale data will be generated from several interconnected mobile devices; as a result, the Industrial Internet of Things (IIoT) provides several opportunities for secure machine learning for industrial applications. The timely information extraction from IIOT

TABLE III: A comparison among different evasion techniques related to proposed model

| Technique | Year | Target | Dataset | Evasion Rate | Countermeasure |
|---|---|---|---|---|---|
| Android HIV [22] | 2019 | Drebin (SVM) | Drebin | 99% | No |
| TLAMD [23] | 2019 | Random Forest | Drebin | 93% | No |
| Harel [24] | 2020 | Drebin (SVM) | Drebin | 99% | No |
| Grosse [25] | 2016 | Deep Learning | Drebin | 63% | Yes |
| **Proposed model** | **2022** | **Drebin (SVM)** | Drebin | **100%** | **Yes** |

data openness of security-critical IIOT issues becomes more challenging with the adoption of machine learning. This study used several discriminating features from the Android App for malware detection. We proposed the adversarial based evasion method to defend against the evasion attacks. The proposed model employs an ensemble-based classification model to train a separate set of features. The tree base pipeline optimization method improves the classification generalization. We then compared our proposed model again the state of the art Drebin method to evaluate the countermeasure to evade by just modifying three features in the feature vector. In contrast, our proposed model achieves 91% accuracies with the change in 14 features. We plan to increase the subset of features in future to defend against adversarial attacks and employ the dynamic analysis on Android ransomware shortly.

## VII. Acknowledgment

This work is partially supported by the National Centre for Research and Development under the project Automated Guided Vehicles integrated with Collaborative Robots for Smart Industry Perspective and the Project Contract no. is: NOR/POLNOR/CoBotAGV/0027/2019 -00.

## References

[1] Z. Ma, J. Ma, Y. Miao, X. Liu, K.-K. R. Choo, Y. Gao, and R. H. Deng, “Verifiable data mining against malicious adversaries in industrial internet of things,” *IEEE Transactions on Industrial Informatics*, vol. 18, no. 2, pp. 953–964, 2021.

[2] K. Liu, S. Xu, G. Xu, M. Zhang, D. Sun, and H. Liu, “A review of android malware detection approaches based on machine learning,” *IEEE Access*, vol. 8, pp. 124 579–124 607, 2020.

[3] N. Martins, J. M. Cruz, T. Cruz, and P. H. Abreu, “Adversarial machine learning applied to intrusion and malware scenarios: a systematic review,” *IEEE Access*, vol. 8, pp. 35 403–35 419, 2020.

[4] J. Jeon, B. Jeong, S. Baek, and Y.-S. Jeong, “Hybrid malware detection based on bi-lstm and spp-net for smart iot,” *IEEE Transactions on Industrial Informatics*, 2021.

[5] T. Kim, B. Kang, M. Rho, S. Sezer, and E. G. Im, “A multimodal deep learning method for android malware detection using various features,” *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 3, pp. 773–788, 2018.

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

[7] J. Garcia, M. Hammad, and S. Malek, “Lightweight, obfuscation-resilient detection and family identification of android malware,” *ACM Transactions on Software Engineering and Methodology*, vol. 26, no. 3, pp. 1–29, 2018.

[8] Y. Pan, X. Ge, C. Fang, and Y. Fan, “A systematic literature review of android malware detection using static analysis,” *IEEE Access*, vol. 8, pp. 116 363–116 379, 2020.

[9] P. Laskov and R. Lippmann, “Machine learning in adversarial environments,” *Mach. Learn.*, vol. 81, no. 2, pp. 115–119, 2010.

[10] Z. Katzir and Y. Elovici, “Quantifying the resilience of machine learning classifiers used for cyber security,” *Expert Systems with Applications*, vol. 92, pp. 419–429, 2018.

[11] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak, “An android application sandbox system for suspicious software detection,” in *International Conference on Malicious and Unwanted Software*, 2010, pp. 55–62.

[12] C. Shang, M. Li, S. Feng, Q. Jiang, and J. Fan, “Feature selection via maximizing global information gain for text classification,” *Knowledge-Based Systems*, vol. 54, pp. 298–309, 2013.

[13] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue, “Droid-sec: deep learning in android malware detection,” in *Proceedings of the ACM conference on SIGCOMM*, 2014, pp. 371–372.

[14] P. R. K. Varma, K. P. Raj, and K. S. Raju, “Android mobile security by detecting and classification of malware based on permissions using machine learning algorithms,” in *International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud*, 2017, pp. 294–299.

[15] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, “Dynalog: An automated dynamic analysis framework for characterizing android applications,” in *International Conference On Cyber Security And Protection Of Digital Services*, 2016, pp. 1–8.

[16] A. Arora, S. K. Peddoju, V. Chouhan, and A. Chaudhary, “Hybrid android malware detection by combining supervised and unsupervised learning,” in *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, 2018, pp. 798–800.

[17] S. J. Hussain, U. Ahmed, H. Liaquat, S. Mir, N. Jhanjhi, and M. Humayun, “Imiad: intelligent malware identification for android platform,” in *International Conference on Computer and Information Sciences*, 2019, pp. 1–6.

[18] R. S. Olson and J. H. Moore, “Tpot: A tree-based pipeline optimization tool for automating machine learning,” in *Workshop on Automatic Machine Learning*, 2016, pp. 66–74.

[19] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, “Scikit-learn: Machine learning in python,” *The Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[20] U. Ahmed, J. C.-W. Lin, and G. Srivastava, “Mitigating adversarial evasion attacks of ransomware using ensemble learning,” *Computers and Electrical Engineering*, vol. 100, p. 107903, 2022.

[21] A. Calleja, A. Martín, H. D. Menéndez, J. Tapiador, and D. Clark, “Picking on the family: Disrupting android malware triage by forcing misclassification,” *Expert Systems with Applications*, vol. 95, pp. 113–126, 2018.

[22] 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, 2019.

[23] X. Liu, X. Du, X. Zhang, Q. Zhu, H. Wang, and M. Guizani, “Adversarial samples on android malware detection systems for iot systems,” *Sensors*, vol. 19, no. 4, p. 974, 2019.

[24] H. Berger, C. Hajaj, and A. Dvir, “When the guard failed the droid: A case study of android malware,” *CoRR*, vol. abs/2003.14123, pp. 14–13, 2020.

[25] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. D. McDaniel, “Adversarial perturbations against deep neural networks for malware classification,” *CoRR*, vol. abs/1606.04435, pp. 44–35, 2016.

[26] Y. Xue, G. Meng, Y. Liu, T. H. Tan, H. Chen, J. Sun, and J. Zhang, “Auditing anti-malware tools by evolving android malware and dynamic loading technique,” *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 7, pp. 1529–1544, 2017.