Feature Importance and Deep Learning for Android Malware Detection

A. Talbi[1,2], A. Viens[3], L.-C. Leroux[3], M. François[3], M. Caillol[3] and N. Nguyen[4,2]
[1]*Pôle Judiciaire de la Gendarmerie Nationale, Cergy, France*
[2]*ETIS Laboratory, CY Cergy Paris University, Cergy, France*
[3]*CY Tech, CY Cergy Paris University, Cergy, France*
[4]*Léonard de Vinci Pôle Universitaire, Research Center, Paris La Défense, France*

Keywords: Android Malware Detection, Static Analysis, Feature Importance, URL Embedding, Deep Neural Network.

Abstract: Effective and efficient malware detection is key in today's world to prevent systems from being compromised, to protect personal user data, and to tackle other security issues. In this paper, we worked on Android malware detection by using static analysis features and deep learning methods to separate benign applications from malicious ones. Custom feature vectors are extracted from the Drebin and the AndroZoo dataset and different data science methods of feature importance are used to improve the results of Deep Neural Network classification. Experimental results on the Drebin dataset were significant with 99.31% accuracy in malware detection. We extended our work on more recent applications with a complete pipeline for the AndroZoo dataset, with about 40,000 APKs used from 2014 to 2021 pre-tagged as reported malicious or not. The pipeline includes static features extracted from the manifest file and bytecode such as suspicious behaviors, restricted and suspicious API calls, etc. The accuracy result for AndroZoo is 97.7%, confirming the power of deep learning on Android malware detection.

1 INTRODUCTION

Since the start of the 2010 decade, the smartphone has become essential for everyone. As its place in our daily life is growing, we trust it enough to entrust it with important and sensitive data such as our bank details or even medical data. Recently, Android is the number one Operating System (OS) in the world (all platforms), with about 40.39% of the global operating system market share in 2021 (StatCounter, 2021).

At the same time, the prevalence of the Android operating system, combined with its open nature, has caused the number of Android malware to skyrocket. To solve this problem, the research communities and security vendors have designed many techniques to identify and prevent Android malicious samples, and two main classes of software approaches to Android malware program analysis exists and have been studied: static and dynamic. Static approaches leverage static code analysis to check whether an application contains abnormal information flows or calling structures, matches malicious code patterns, requests for excessive permissions, and/or invokes APIs that are frequently used by malware. Static analysis of an Android application can rely on features extracted from the manifest file or the Java bytecode, while dynamic analysis of Android applications can deal with features involving dynamic code loading and system calls that are collected while the application is running. However the main limitation to the use of dynamic analysis is that it requires the study of program behavior, the execution of each instruction, and the ability to modify instructions or registers during this phase.

For the detection and classification purposes, in addition to the use of traditional and manual techniques such as static and dynamic analysis, works using artificial intelligence and more particularly deep learning, have shown encouraging results. Our approach combines static analysis on bytecode and data retrieval from Android Package (APK) files for multiple feature extraction, with the use of 5-layers neural networks that are trained on a large amount of data. For the experimental results, we obtained two application databases: one from Drebin's team (Arp et al., 2014) but whose applications are a bit dated and the other coming from AndroZoo (Allix et al., 2016), which is an application database made available by the University of Luxembourg to the scientific community.

Talbi, A., Viens, A., Leroux, L., François, M., Caillol, M. and Nguyen, N.
Feature Importance and Deep Learning for Android Malware Detection.
DOI: 10.5220/0010875500003120
In *Proceedings of the 8th International Conference on Information Systems Security and Privacy (ICISSP 2022)*, pages 453-462
ISBN: 978-989-758-553-1; ISSN: 2184-4356

On the Drebin data, we used already extracted features such as permissions, activities, API calls, etc. and we obtained an accuracy rate in detecting malicious applications of 99.31%. With the AndroZoo data, we selected only the applications after 2014, and we extracted the features coming from the AndroidManifest.xml file but also those coming from the bytecode, such as suspicious behaviors, network addresses, suspicious API calls, and restricted API calls. We labelled an application as malicious if it was flagged by at least 4 antivirus on VirusTotal (Sood, 2017). Then we followed a process of cleaning and formatting the data, selecting the best features, and performing a dataset slicing. Finally, 5-layers deep learning models are trained and optimized by adjusting different parameters and we came up with results offering very high accuracy.

Our main contributions in combining deep learning and static analysis for Android malware detection are as follows. We proposed a detailed feature engineering process by: i) including more feature types on feature extraction, ii) grouping similar features, iii) selecting important features from a large number of feature sets, and iv) embedding Uniform Resource Locator (URL) using Natural Language Processing (NLP). For feature extraction, we obtained a new set of features targeting suspicious actions from Smali bytecode for AndroZoo dataset, which was not used by Drebin.

The paper is organized as follows. Section 2 references existing work related to malware detection by using artificial intelligence techniques. Then, Sections 3 represents our methodology with 5 steps. The feature extraction step is described in Subsection 3.1. Subsections 3.2 and 3.3 focus, respectively, on the grouping and selection of the most important features required for the classification of applications. In Subsection 3.4, an overview is provided on URL embedding to improve the recognition of certain URLs used more frequently in malware. Subsection 3.5 describes the feeding of features into a deep neural network for binary classification, once these features have been extracted and pre-processed. In Section 4, results obtained with two datasets are presented and compared, in the first instance, the Drebin dataset, using their pre-extracted features from the APK files, and then on the AndroZoo dataset in which feature extraction was performed by ourselves. Finally, conclusions and perspectives are described in Section 5.

2 RELATED WORK

Over the past few years, many solutions have been proposed to detect Android malware using machine learning algorithms. Recent reviews such as (Naway and Li, 2018), (Wang et al., 2019), and (Liu et al., 2020) provided clear and comprehensive surveys of the state of the art in the domain. (Naway and Li, 2018) gave an overview of the different papers using deep learning algorithms for malware detection with their different performances as well as datasets used. (Wang et al., 2019) presented a comparative analysis of 236 published papers on feature extraction techniques for Android applications. (Liu et al., 2020) complemented the previous reviews by surveying a wider range of aspects concerning machine learning development pipeline. For the rest of this section, we will present several approaches which are most relevant and closer to our work.

The authors in (Wu et al., 2012) first extracted the information from each application's manifest file, and then, applied the K-Means algorithm that enhances the malware modeling capability. The number of clusters is decided by Singular Value Decomposition (SVD) method on the low rank approximation. They used the K-Nearest Neighbors (KNN) algorithm to classify the application as benign or malicious. The comparison with Androguard tool showed a better performance of this method. In addition, Droid Mat is efficient since it takes only half of time than Androguard to predict 1,738 apps as benign apps or Android malware.

(Arp et al., 2014) proposed DREBIN, a lightweight method for the detection of Android malware that enables identifying malicious applications directly on the smartphone. As the limited resources impede monitoring applications at runtime, DREBIN performed a broad static analysis, gathering as many features of an application as possible. These features were embedded in a joint vector space, such that typical patterns indicative for malware can be automatically identified and used for explaining the decisions of their method. In an evaluation with 123,453 applications and 5,560 malware samples, DREBIN outperformed several related approaches and detected 94% of the malware with few false alarms, where the explanations provided for each detection reveal relevant properties of the detected malware. On five popular smartphones, the method required 10 seconds for an analysis on average, rendering it suitable for checking downloaded applications directly on the device.

Recently, particular research has been focused on tuning the network parameters. In (Hou et al., 2016), different network architectures are tested while tuning the parameters to reach a higher level in terms of detection accuracy. In (Backes and Nauman, 2017), the hyperparameters of Convolution Neural Network (CNN) are tuned while using a dropout of 0.2 at each convolution layer to reduce overfitting, which leads to the conclusion that the largest network gives the highest statistical metric values.

In the work of (Nix and Zhang, 2017), a CNN is built and evaluated for API call-based Android app classification. Long Short-Term Memory (LSTM) technique is integrated to extract knowledge from system API-call sequences. CNN results are compared with respect to n-gram Support Vector Machine (SVM) and the Naive Bayes algorithm and the performance of CNN is much better compared to others.

In another study (Kapratwar et al., 2017), the authors proposed a malware detection method inspired by deep learning about the exhaustive combination of static and dynamic analysis, which traces all possible execution paths of a given file, then compares the flow graphs in real time for malware identification. Their premise is that deep learning with a deep architecture can evolve high-level representations by associating features from static analysis with those from dynamic analysis, which can then better characterize Android malware.

In another register of Android permission based malware detection technique, (Dong, 2017) gathered a huge set of both malware and benign applications through web crawler and developed a tool to decompile applications to source code and manifest files automatically. Then permissions with other information are extracted for each app, to finally take advantage of machine learning algorithms, including Logistic Regression Model, Tree Model with ensemble techniques, Neural Network and an ensemble model to find patterns and more valuable information. This method generated a good accuracy, F-score and overall performance of malicious application prediction. The default model with one hidden layer and total perceptron's returns an accuracy of 93% and F-score 90%.

(Ganesh et al., 2017) proposed a deep learning-based malware detection to identify and categorize malicious applications. The study partially used Drebin dataset. The method investigated permission patterns based on a CNN and they identified malware with 93% accuracy on a dataset of 2,500 Android applications, of which 2,000 were malicious and 500 were benign.

Also using the Drebin dataset, (Li et al., 2018) proposed an Android malware characterization and identification approach that uses deep learning algorithms to address the urgent need for malware detection. Extensive experimental results showed that their approach achieved over 90% accuracy with only 237 features.

(Kim et al., 2019) presented a different approach that uses various kinds of features to reflect the properties of Android applications from several aspects. The features are refined using existence-based or similarity-based feature extraction method for effective feature representation on malware detection. Besides, a multimodal deep learning method is proposed to be used, for the first time, as a malware detection model and recorded an accuracy of 85%.

(Pektaş et al., 2020) used the API call graph as a graph representation of all possible execution paths that a malware can track during its runtime. The embedding of API call graphs transformed into a low dimension numeric vector feature set is introduced to the DNN. Then, similarity detection for each binary function is trained and tested effectively. This study also focused on maximizing the performance of the network by evaluating different embedding algorithms and tuning various network configuration parameters to ensure the combination of the hyper-parameters and to reach the highest statistical metric value.

It is within this framework that our work fits, with the objective of optimizing the detection precision by using DNN as well as the feature importance and the extraction of additional features.

3 OUR METHODOLOGY

Our objective is to determine if an application is a malware from its APK, with a high precision. To this end, we train a deep learning model from a database of APKs. Our methodology is divided in five steps. First, we extract various features from the APKs, such as permissions, app components, suspicious API calls, network address, etc. Then, we preprocess all these features in order to optimize the vectors that will be fed as the inputs of our deep learning models by using feature grouping and feature filtering techniques. In parallel, a URL embedding framework is carried out. And after training our models with different architectures, we analyze the results to understand the predictions and the improvements to carry on. The flowchart in Figure 1 goes over this whole process. The numbers from the Drebin dataset (detailed later in Section 4) are used to illustrate the vector size reduction.

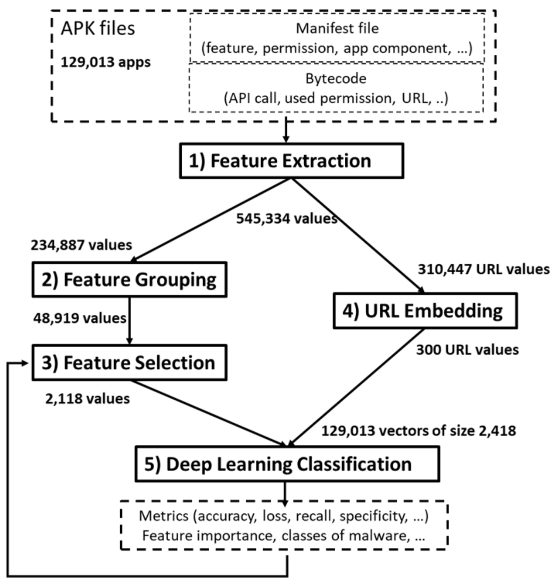
APK files
129,013 apps
Manifest file
(feature, permission, app component, ...)
Bytecode
(API call, used permission, URL, ..)
1) Feature Extraction
545,334 values
234,887 values
310,447 URL values
2) Feature Grouping
48,919 values
4) URL Embedding
3) Feature Selection
300 URL values
2,118 values
129,013 vectors of size 2,418
5) Deep Learning Classification
Metrics (accuracy, loss, recall, specificity, ...)
Feature importance, classes of malware, ...

Figure 1: Methodology Flowchart (Drebin Dataset).

3.1 Feature Extraction

In order to perform malware detection from static analysis, the first step is to extract the needed features from the applications that are in APK format. There exists different types of features suggested by Drebin and proven to be effective, that are divided into 8 groups, ranging from S1 to S8 as shown in Table 1.

The S1 to S4 sets are extracted from AndroidManifest.xml which provides data supporting the installation and later execution of the application. It includes the requested hardware components such as camera, GPS, etc., the permissions that the application needs in order to access protected parts of the system or other applications, the components of the application which include all activities, services, broadcast receivers and content providers, and also filtered intents.

The features S5 to S8 are from the Dalvik bytecode. The restricted API calls (S5) are retrieved by static analysis of the bytecode by taking all the API calls of the application and looking at what permissions are needed to use these APIs. So if one of these permissions is not mentioned in the AndroidManifest.xml file, then the API combined with the needed permission is picked up and set as a feature, as this can sometimes imply an exploit used by malware to perform an action without permission. S6 (Used Permissions) contains permissions that are asked by the application at some point during its execution. The selection of suspicious APIs (S7) is done based on the list of most used APIs by malware. Finally, S8 contains the list of IP addresses and URLs present in the bytecode.

Table 1: Drebin and AndroZoo Static Feature Sets Suggested by (Arp et al., 2014) and Additional Set S9.

S1	Hardware Components	S5	Restricted API Calls
S2	Requested Permissions	S6	Used Permissions
S3	App Components	S7	Suspicious API Calls
S4	Filtered Intents	S8	Network Addresses
S9	*Suspicious Actions (AndroZoo only)*		

The notable differences concerning our exploitation of the datasets are that Drebin was already constituted of the extracted features (from S1 to S8). But for AndroZoo, we only had the list of APKs, the features are extracted with our own scripts by using the general idea of Drebin for the S1 to S8 features. In addition, a new set of features named ”Suspicious Actions” (S9) has been extracted. The interest of this set of features is that it gathers combined actions that represent proven malicious behaviors. For example, when a malware tries to intercept the incoming SMS, it has to run a daemon in back-end listening incoming SMS and to exfilter them through the HTTP protocol. So the features from S9 target specific malicious techniques which are exfiltration of phone data and configuration, geolocation data leakage, exfiltration of interface connection information, abuse of telephony service, interception of audio and telephony streams, establishment of remote connections, Personal Information Manager (PIM) data leakage, operations on external memory, modification of PIM data, arbitrary code execution, and service denial. The extraction of these features was performed with the Python library Androwarn, by using Smali which is an assembler/disassembler for the dex format used by Dalvik bytecode.

These features are then one-hot encoded, which creates a sparse vector where each dimension is 1 if the feature is present, and 0 otherwise. Some of the sets contain very little possible distinct values. The number of possible permissions in S6 (Used Permissions) is limited by Android OS whereas the number of unique network addresses can be substantially large. For some of the sets, the one-hot encoded representation of every possible features may be suboptimal. In fact, most of the dimensions of S3 (App Components) and S8 (Network Addresses) do not bring much information to the neural network, but rather add noise. It inspired us to perform feature selection and feature engineering on S1 to S8 sets, in order to reduce memory and computing time of our feature generation and network training. By selecting only the best features, we expect the same or just a small decrease in results but with limited computing time. However, feature engineering is expected to add information to the network, increasing the overall accuracy.

For S8, due to the nature of URLs, only a small subset is used in different apps, and most of them are unique. Consequently, one-hot encoding URLs may not bring any information to the neural network, and does not capture the similarity between two pages of the same website, or similar URLs. We developed an URL embedding framework to capture the representation of the URLs used in the app in the same-sized numerical vector, which is supposed to capture more information than just the re-use of a specific URL. This framework is discussed in Section 3.4.

3.2 Feature Grouping

A lot of features extracted from the application are texts, separated by dots. For example, a permission feature looks like ‘com.android.launcher.permission.READ_SETTINGS’, and it’s similar for the other subsets. But not all the parts of the text have the same information. For the permissions, we select only the last part of the text, after the last dot: ‘READ_SETTINGS’. Taking only this part regroups a lot of different features finishing by the same text, like: ’com.motorola.launcher.permission.READ_SETTINGS’, or ’org.adw.launcher.permission.READ_SETTINGS’. To illustrate the new features, Figure 2 represents the percentage of malware applications in the Drebin dataset which have these specific features.

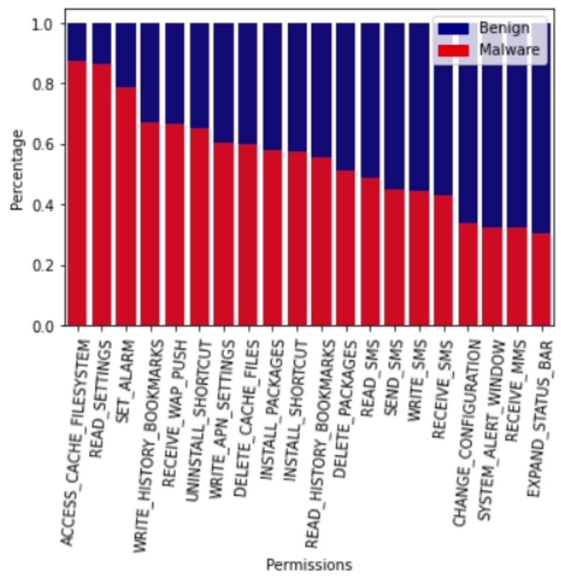

Figure 2: Percentage of Drebin Malware Applications with Permission Grouped Features.

Thereby, we have a new permission feature ’READ_SETTINGS’ created by this process, which we do for all the subsets. We choose to keep the original features for the subsets of small dimensions, like the permissions (S2 and S6) or intents (S4). But for large subsets like activities in S3, we keep only the regrouped features. By this process, we reduced the number of unique values from 234,887 (URLs excluded) to 48,914 for Drebin (Figure 1), and we added new information.

3.3 Feature Selection

Our objective now is to reduce more the number of unique features in order to increase the efficiency of the model’s training. For that, we rank all the features by their importance to the malware application prediction in order to keep only the best ones. To have a precise estimation of the importance of each feature, we combine the importance of the features from 5 different algorithms: the Pearson Correlation, the Chi-Squared metric, and 3 machine learning models which are Logistic Regression, Random Forest and LightGBM. For each algorithm, we rank the features by their importance to the prediction. But to have a more precise ranking, we define a score for each feature, according to its ranking for each algorithm, using the rank product.

Table 2 illustrates the ranking for the permission subset of the Drebin dataset. The column **Top Algo** represents the number of algorithms that the feature is in the top 80% of the ranking, the column **Score** stores the rank product score and is used for filtering the features. With this ranking, we can filter the n top features of each set. The more features are selected, the more the deep learning model will have one-encoded input and will take time to optimize the weights of each feature. In our study, we selected all the features with a score less than 300, which corresponds to 2,118 features with the Drebin dataset (Figure 1).

3.4 URL Embedding

In order to use the URLs found in the APKs, they have to be represented in a fixed-length vector. To this end, we used the work of (Yuan et al., 2018) enabling a URL to be represented in a fixed-size vector space. Their work has shown that it is beneficial to split the URL into different parts in order to improve the representation and subsequently the performance of the machine learning models. For example, a URL can be divided into 5 parts: protocol, sub-domain, domain, domain suffix, and URL path. In our framework, we divided URLs into 3 parts: P1 (Protocol),

Table 2: Permission Features Ranking for Drebin Dataset.

Feature	**Pearson**	**Chi-Square**	**Logistic Regression**	**Random Forest**	**LightGBM**	**Top Algo**	**Score**
android.permission.SEND_SMS	1	1	29	2	7	5	**3.324**
READ_SMS	3	3	156	4	13	5	**9.39**
SEND_SMS	2	2	30	1	722	4	**9.717**
ACCESS_COARASE_LOCATION	845	791	1127	1682	2865	2	**1294.141**
...							

P2 (sub-domain + domain + domain suffix), and P3 (URL path). The usefulness of the URL parts decomposition is that two URLs with the same protocol, and the same domain, but with a different path, will have 2/3 of their vector representation identical.

Then, we train a character model that gives representation of each character in a vector space via language modeling. We model our URLs into a sequence of characters: $U = c_1c_2...c_n$. The characters $c_i \in V$ are part of a vocabulary V, that contains all possible characters in the URL corpus. Subsequently, we use this character model to build a vector representation of each part of a URL by averaging all the character vectors of that part and then we concatenate these vector representations to obtain the representation of the whole URL. The workflow is described in Figure 3.

Almost always, an application accesses not one but multiple URLs or IP addresses. Since the representation of all the URLs of an application has to be of fixed size, the last step is the aggregation of these representations. We used two types of autoencoders to this end. One Long Short-Term Memory (LSTM) autoencoder that accepts variable-length URL representation that outputs one compressed representation, and another autoencoder that accepts padded lists of URL representations. As shown in Figure 4, the autoencoder takes k URLs in its input layer, and the middle hidden layer is a vector of features that carries most of the important information. This URL encoding allows us to reduce the number of URL unique values from 310,447 to 300 (Figure 1). The implementation of these autoencoders is successful but as described later in Section 4, the contribution is negligible, as they could not learn a meaningful representation of all the URLs used in an application.

3.5 Deep Learning Classification

Once each feature has been extracted and preprocessed, we can feed it into a DNN for binary classification or multi-class classification. As the number of input dimensions is still pretty high, DNNs are the most suitable algorithms to perform these classifications. In fact, they are able to construct high-level, intermediate feature representations (or concepts) in the hidden layers that can model complex relationships between input features. Many different machine learning algorithms have been used to tackle the detection of Android malware, but DNNs have been shown to have very good results.

The number of neurons in the input layer corresponds to the size of our feature vectors, while the number of neurons in the output layer corresponds to the number of classes we want to predict (2 for a binary classification and 3 or more for a multi-class classification). For our training, we used Tensorflow 2.0 with CUDA enabled to parallelize calculations on the GPU. However, they are costly to train. We tried different types of network architectures, all with fully connected layers. Since the use of DNN with 2 layers composed of 256 neurons each is recommended in several works (Li et al., 2018), we used this information to test our different models and compared the performance of the 2, 3 and 4 layer configurations for the binary and multi-class classification cases. The hidden layers contain between 256 and 512 neurons each that are connected to a softmax layer, outputting probabilities for the classes 'Malware' or 'Benign'. To optimize the DNN model, we can act on the following hyper parameters: number of hidden layers, number of neurons per layer, backpropagation algorithm, cost function, and activation function.

As backpropagation algorithm, we have tested the stochastic gradient descent algorithm and the Adam algorithm. Adam is an adaptive learning rate optimization algorithm that was designed specifically for training of DNNs. In this algorithm, the learning rate is calculated for each variable and depends on 3 parameters with recommended values: $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\varepsilon = 1e^{-8}$.

To avoid overfitting, we have split randomly our dataset into three subsets. The training set takes 70% of the dataset and is used to train the model. The validation set to compute at each epoch the metrics on a set that has not been used for training. The test set allows us to calculate at the end the capacity of the network to generalize on a new data set. The con-

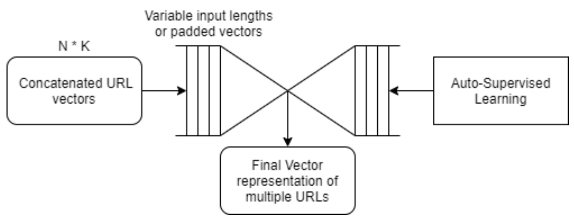
Variable input lengths or padded vectors
N * K
Concatenated URL vectors
Auto-Supervised Learning
Final Vector representation of multiple URLs

Figure 4: Multiple URL Compressed Representation.

fusion matrix and the different metrics are calculated from this set.

4 EXPERIMENTAL RESULTS

4.1 Datasets

We first worked on the Drebin dataset using their already extracted features sets. The Drebin dataset contains 123,453 benign applications and 5,560 malware, divided in 179 malware classes. This dataset has been assembled from 2010 to 2012 from various Android application platforms. The advantages of this dataset are that it is large and the features have already been extracted from the AndroidManifest.xml file and from the bytecode.

Next, we used the AndroZoo dataset to set up an end-to-end pipeline, from the extraction of the features from the manifest file and the bytecode of the APKs to neural network learning and testing. This database is huge, totaling more than 15,500,000 applications, with various information. We extracted 39,156 APKs from AndroZoo, with a proportion of 12,882 malware that have been identified by at least 4 antivirus programs as containing malicious code and 26,274 benign applications from Google markets. The release dates of these APKs vary from 2014 to 2021, hence some of them are really recent. In AndroZoo, information about the size, the origin mobile markets and the number of VirusTotal antiviruses that recognized the app as a malware, or a benign are given. The number of antiviruses recognizing an application as malware spans from 0 to 57, and so we chose to classify as malicious an application with this number superior or equal to 4 whereas the benign apps are required to have 0 positive scans.

4.2 Results

4.2.1 Drebin

With the Drebin dataset, we tried different architectures of our model with different input. An accuracy around 99% with only feature grouping and feature selection is achieved. By adding URL embedding, we found that the accuracy did not increase. On the contrary, the feature importance techniques work well without URL embedding and we have with the best architecture up to 99.31% of accuracy, as shown in Table 3. This architecture is composed of 3 layers (1024, 512 and 256 cells) and trained with 60 epochs. The input used are the features with a score less than 300 for the feature selection.

The column F1 represents the F1-score that subtly combines precision and recall. And in the last column we have indicated the False Negative Rate (FNR), because we think it is important to point out the malware that passes through the analysis without being detected as such.

Table 3: Results on Drebin Dataset.

Model	Accuracy	F1	FNR
Best architecture	0.99317	0.99644	0.00532

Our predictions for the best architecture are quite interesting because we have only 22 false positives and 66 false negatives. The false positives are not very concerning as it is better to predict that a benign application may be a malware than to miss identifying a malware application. We tried to understand where these 66 false negatives come from, and how we can improve our results. For that, we focus on the different malware classes, as the Drebin dataset indicates the type of malware. Table 4 shows the top classes the most present in the dataset, with the number of applications in each family, the accuracy of prediction and the average of the false and true positives rates. As we can see, the Gappusin class is one of the top classes that our model has difficulties to predict.

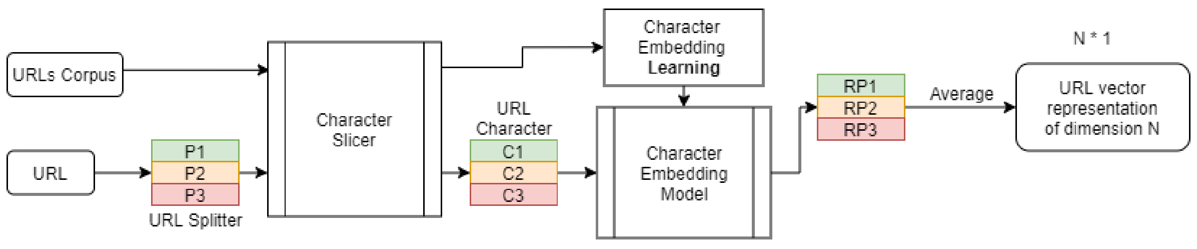
URLs Corpus
URL
P1
P2
P3
URL Splitter
Character Slicer
URL Character
C1
C2
C3
Character Embedding Learning
Character Embedding Model
RP1
RP2
RP3
Average
N * 1
URL vector representation of dimension N

Figure 3: Single URL Embedding Workflow.

We analyzed with SHAP (Lundberg and Lee, 2017) how the predictions of this class are done. We noticed that it's mostly the absence of typical features, which are usually in the malware, that confused our model to predict it malware. This class of malware is very similar to the benign class.

Table 4: Malware Families of Drebin Dataset.

Family	**Apps**	**Accuracy**	**FP**	**TP**
FakeInstaller	104	0.98	0.03	0.97
Plankton	74	0.99	0.04	0.96
Opfake	64	0.95	0.07	0.93
Gappusin	12	**0.25**	0.68	0.32
...				

Then, we analyzed the feature importance of our deep learning model, to understand what are the main characteristics of the decision of the model. Figure 5 gives the feature importance represented by SHAP. In this diagram the highest elements are the ones with the most influence on the model. The horizontal location (x-axis) of points shows whether the effect of that location is associated with a higher or lower prediction, the red dots represent a positive influence and the blue dots a negative influence. This means the more the red dot is on the right and features is on the higher part, the more the model will predict that the application is a malware. We can see that the presence of features like 'com', or 'getSubscriberid' implies that the application is more likely to be a malware. Rather the presence of the features 'android' or 'android.permission.ACCESS_FINE_LOCATION' implies that the application is more likely to be benign.

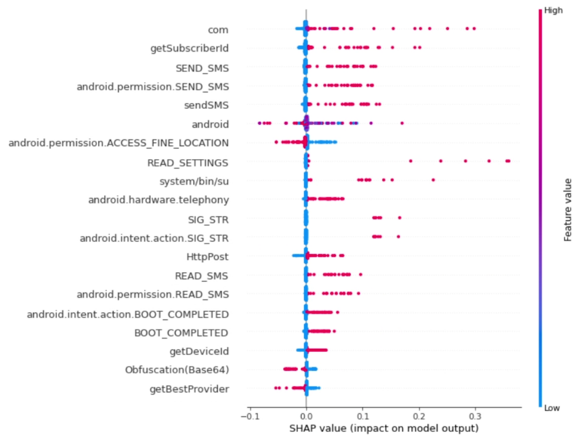

Figure 5: Feature Importance for Drebin Dataset.

Another interesting criteria is the importance of each set in the prediction. For that, we sum up the features of importance for each subset as shown in Figure 6. We can conclude that the permission subset is by far the most important subset to predict if an application is a malware in Drebin dataset.

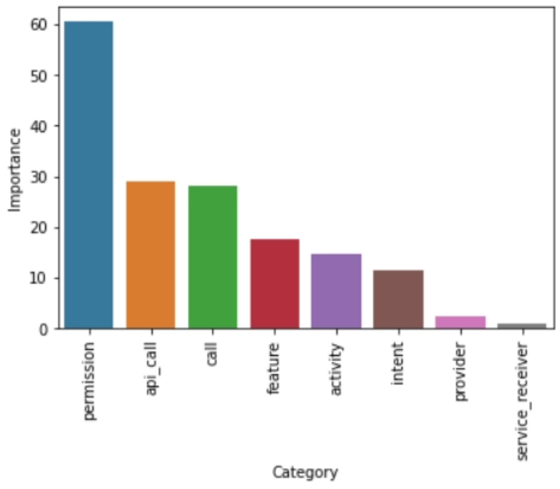

Figure 6: Subset Importance for Drebin Dataset.

Finally, Table 5 compares the different measures from other related work that worked exclusively with the Drebin dataset. And we can see that our method and that of (Li et al., 2018), both use neural networks, achieve the best results. This table compares different metrics, features, the algorithm used and a brief description of their contribution.

4.2.2 AndroZoo

We also worked with the AndroZoo dataset, and we used 27,410 (70%) apps for the training set, 5,873 (15%) apps for the test set and 5,873 (15%) for the validation set. The training and test set was composed of 32.9% of malware. We trained a neural network with an input layer, followed by two 256 cells dense layers, and a softmax layer. Unlike Drebin's dataset, we extracted ourselves the features of the APKs, which can induce a slight difference in the results. Moreover, we extracted the additional feature "Suspicious action" which is not present in Drebin.

With AndroZoo dataset, the best accuracy reached is 97.7%, which is indeed very promising with feature importance. We see that feature selection is really effective on this new dataset, because with substantially fewer features, the network manages to gain in each test metric as shown in Table 6. Moreover, the computing time of training by selecting only the most important features is significant.

With Androzoo the predictions are interesting because we have only 65 false positives (which represents 3.3% of FPR against 4.2% with Drebin dataset)

Table 5: Comparison Table of Published Works with Drebin Dataset.

Reference	**Measures**	**Features**	**Algorithm**	**Contribution**
(Arp et al., 2014)	Acc: 94%	S1, S2, S3, S4, S5, S6, S7, S8	SVM	Online and explainable malware detection
(Li et al., 2018)	Prec: 97.15% Recall: 94.18% F1: 95.64%	S2, S6, S5, S7	DNN	Automatic detection engine to detect malware families
(Shiqi et al., 2018)	Acc: 95.7%	S5, S7	DBN	Combination with image texture analysis for malware detection
Our method	Acc: 99.31% Recall: 99.46% F1: 99.64%	S1, S2, S3 , S4, S5, S6, S7, S8	DNN	Feature grouping and feature selection

Table 6: Results on AndroZoo Dataset.

Model	**Accuracy**	**F1**	**FNR**
Best architecture	0.97701	0.98288	0.01776

and 70 false negatives (which corresponds to 1.77% of FNR against 0.53% with Drebin dataset).

In Figure 7, the most important feature families are presented, and we can see that the added feature (Suspicious actions) has the most important weight, seconded by the permissions, while it is the permission set which has the heaviest weight with Drebin dataset. This probably means that we could significantly increase the results of the first model (Drebin dataset) if the "Suspicious actions" features have been added.

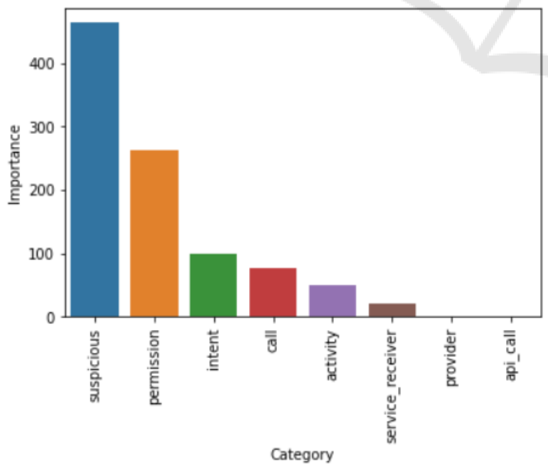

Figure 7: Subset Importance for AnrdoZoo Dataset.

To analyse the feature importance of the deep learning model fed by AndroZoo dataset, we generated a SHAP diagram that shows the feature importance. We can see in Figure 8 the red dots that indicate how much the feature is present, and the blue dot means the contrary, and as for the previous SHAP diagram, the further the dot is on the right the more this feature is likely to represent a malware, and on the left benign. Thus we can see that the presence of features 'READ_PHONE_STATE' or 'com' implies that the application is more likely to be a malware. And the presence of the feature 'RecyclerView' indicates that the application is more likely to be benign.

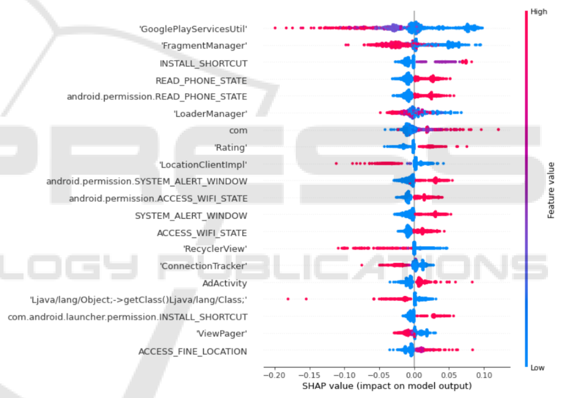

Figure 8: Feature Importance for AndroZoo Dataset.

During the experimentation phase, the hardware characteristics used are the following for the training of the models without the feature selection: Processor Intel(R) Core(TM) i3-10100F CPU 3.60 GHz, 16 Gb RAM, GPU NVIDIA GeForce GTX 1650 SUPER. And for the training of the models with the features selection, the hardware characteristics used are: Processor Intel(R) Xeon(R) CPU E5-2678 v3 2.50 GHz, 12 Gb RAM, GPU.

5 CONCLUSIONS

To conclude this study, our work contributes to the research field of malware detection for Android application by improving the detection rate. We based our work on the research paper of Drebin (Arp et al., 2014), and we achieved 99.31% of accuracy, superior than the highest rate with exclusively the same dataset from other related work. In addition, we extended our working dataset with more recent data extracted from AndroZoo APKs, and we improved the accuracy by using deep learning techniques and by the extraction of multiple and additional features from bytecode and the AndroidManifest.xml file. Our methodology has proven to be effective with an accuracy of nearly 97.7% in detecting recent Android malware by binary classification. A dataset consisting of features extracted from nearly 80,000 recent applications with about 30,000 malware will be made available on the Internet, as well as the script to extract these features from a raw AndroZoo dataset.

Different areas of improvement can be studied, such as optimizing hyperparameters, exploiting a greater mass of applications from the AndroZoo dataset, and extending the extracted features from bytecode to improve the model. Ongoing work on multi-class classification to better categorize Android malware families is actually carried out. It is also interesting to study the cases of APK predicted as false positives, to understand why they were tagged as malware. Manual reverse engineering techniques could eventually reveal unknown attacks that were not detected by classical antivirus.

REFERENCES

Allix, K., Bissyand and, T., F., Klein, J., and Le Traon, Y. (2016). AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 468–471, New York, NY, USA.

Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., and Siemens, C. (2014). Drebin: Effective and explainable detection of Android malware in your pocket. In *Network and Distributed System Security Symposium*, volume 14, pages 23–26, San Diego, California, USA.

Backes, M. and Nauman, M. (2017). Luna: Quantifying and leveraging uncertainty in Android malware analysis through Bayesian machine learning. In *IEEE European Symposium on Security and Privacy (EuroS P)*, pages 204–217, Paris, France.

Dong, Y. (2017). Android malware prediction by permission analysis and data mining. In *PhD*, University of Michigan-Dearborn.

Ganesh, M., Pednekar, P., Prabhuswamy, P., Sreedharan, D., Park, Y., and Jeon, H. (2017). CNN-based Android malware detection. San Diego, CA, USA.

Hou, S., Saas, A., Ye, Y., and Chen, L. (2016). Droiddelver: An Android malware detection system using deep belief network based on api call blocks. In *International Conference on Web-Age Information Management*, volume 9998, pages 54–66, Nanchang, China.

Kapratwar, A., Di Troia, F., and Stamp, M. (2017). Static and dynamic analysis of Android malware. In *International Conference on Information Systems Security and Privacy*, pages 653–662, Porto, Portugal.

Kim, T., Kang, B., Rho, M., Sezer, S., and Im, E. G. (2019). A multimodal deep learning method for Android malware detection using various features. *IEEE Transactions on Information Forensics and Security*, 14(3):773–788.

Li, D., Wang, Z., and Xue, Y. (2018). Fine-grained Android malware detection based on deep learning. In *IEEE Conference on Communications and Network Security (CNS)*, Beijing, China.

Liu, K., Xu, S., Xu, G., Zhang, M., Sun, D., and Liu, H. (2020). A Review of Android Malware Detection Approaches Based on Machine Learning. *IEEE Access*, 8:124579–124607.

Lundberg, S. and Lee, S.-I. (2017). A unified approach to interpreting model predictions. Seattle, WA.

Naway, A. and Li, Y. (2018). A Review on The Use of Deep Learning in Android Malware Detection. In *International Journal of Computer Science and Mobile Computing*, volume 7, pages 42–58.

Nix, R. and Zhang, J. (2017). Classification of Android apps and malware using deep neural networks. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 1871–1878, Anchorage, Alaska, USA.

Pektaş, Abdurrahman, and Acarman, T. (2020). Deep learning for effective Android malware detection using API call graph embeddings. *Soft Computing*, 24(2):1027–1043.

Shiqi, L., Shengwei, T., Long, Y., Jiong, Y., and Hua, S. (2018). Android malicious code classification using deep belief network. *KSII Transactions on Internet and Information Systems*, 12(1).

Sood, G. (2017). *virustotal: R Client for the virustotal API*. R package version 0.2.1.

StatCounter (June 2021). Operating system market share worldwide". https://gs.statcounter.com/os-market-share.

Wang, W., Zhao, M., Gao, Z., Xu, G., Xian, H., Li, Y., and Zhang, X. (2019). Constructing Features for Detecting Android Malicious Applications: Issues, Taxonomy and Directions. volume 7, pages 67602–67631.

Wu, D.-J., Mao, C.-H., Wei, T.-E., Lee, H.-M., and Wu, K.-P. (2012). Droidmat: Android malware detection through manifest and API calls tracing. In *Seventh Asia Joint Conference on Information Security*, pages 62–69, Tokyo, Japan.

Yuan, H., Yang, Z., Chen, X., Li, Y., and Liu, W. (2018). URL2Vec: URL modeling with character embeddings for fast and accurate phishing website detection. pages 265–272, Melbourne, Australia.