Android Application Classification Tool using Machine Learning

ADRIJA NAG, Arizona State University, USA
ALEX GAOUETTE, Arizona State University, USA
JACOB WATSON, Arizona State University, USA

As Android phones continue to dominate the market, these devices have become a big source of security vulnerability to the users. Malicious apps are proliferating putting users' sensitive and private information at risk. Existing methods for detecting malware utilize both static and dynamic features of an app. In this paper, we present three machine learning based approaches to classify Android applications into the categories of benign and malicious. Each approach uses a distinct set of features extracted from the the Android Manifest file of the app. The first approach uses permissions requested by an app, the second uses the intent actions while the third uses metadata from the manifest file to determine if a certain app is benign or malicious. On each feature set, we train three different classification models - Random Forest, Decision Tree and Support Vector Classifiers, and then compare results. Our evaluation of F1-scores for all three models on all three feature sets exceeds 80% by a large margin, with Random Forest model trained on the user permission based feature set giving the highest F1-score of 94.96%.

Additional Key Words and Phrases: malware, static analysis, dynamic analysis, cross-validation

ACM Reference Format:
Adrija Nag, Alex Gaouette, and Jacob Watson. 2022. Android Application Classification Tool using Machine Learning. 1, 1 (December 2022), 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Over the past decade, the use of smartphones has grown rapidly, ultimately making these devices an indispensable commodity among people. In 2021, over 1.4 billion smartphones were sold globally [6]. Of these, more than 70% run on the Android operating system [7]. With mobile computing gaining momentum, Google Play Store houses an ever growing number of applications for services ranging from banking to shopping to entertainment. In order to work as intended, these applications collect an array of sensitive user information. Given, its popularity and reach, this makes the android app market an easy target for cyber-criminals to spread malicious software.

Malware can manifest itself in many forms - adware, ransomware, scareware or smsmalware, but the end objective is to gain access to data or permissions that were otherwise not supposed to be accessed by said malicious app and then later exploit the data collected. As this breach of security has the potential to cause tremendous damage to the user, significant amount of research has been done on ways to mitigate this problem. Existing approaches of detecting malware in an application include the static analysis [10] and dynamic analysis [8].

Static analysis relies upon files in the Android package to classify the app, i.e., the app need not be run to determine its category. This method, thus, is fast and involves fewer steps [15]. Dynamic detection, on the other hand, executes

Authors' addresses: Adrija Nag, anag9@asu.edu, Arizona State University, Tempe, Arizona, USA; Alex Gaouette, agaouett@asu.edu, Arizona State University, Tempe, Arizona, USA; Jacob Watson, jrwats10@asu.edu, Arizona State University, Tempe, Arizona, USA.

the app on an emulator or a device to detect malicious behaviour. As a result, this approach takes more time than the former.

Machine learning techniques can be used in tandem with both of these approaches to automate the process of malware detection or app classification [9]. In this paper, we have used static features found in the Android package (APK), particularly in the *AndroidManifest.xml* file to train several machine learning classifiers and compares the results. The models we have used are Random Forest, Decision Tree and Support Vector Classifiers. The working theory behind each of these models will be detailed in the next section.

2 BACKGROUND

In this section, we will discuss malware, the structure of an android app, the *AndroidManifest.xml* file and the features in there that are relevant to our paper.

2.1 Malware

A malware is a piece of software intentionally created to cause a change in a computer system that it is otherwise unauthorized to carry out. It could be in the form of unauthorized access to sensitive user data, disruption of computer network or erasing all data from a computing system, just to name a few. There are several different types of malware including viruses, worms, Trojan horses, spyware, adware, ransomware and scareware [13]. Whatever form it may manifest itself, a malware has the potential to put the user's computing system, an Android device in this case, at a great security risk.

2.2 Structure of an Android application

It is essential to understand the key components in an Android app along with other important files that are part of an Android package. The four key components in any Android app are Activity, Service, Broadcast Receiver and Content Provider [4].

Activities set the user interface of the app and lay out screens with buttons, input blocks and text blocks, through which the user can interact with the app. Services are components that run in the background for longer or time bound operations. They do not offer any user interface. Broadcast receivers are used for delivering events between applications or activities. These allow Android apps to respond to external events. Content providers manages the shared app data which can be stored in databases, on the web, in a local file system or someplace else.

Except for services, all the other components use intents for their activation. Intents are responsible for binding all components together at runtime.

2.3 The Manifest File

Every app needs to have an *AndroidManifest.xml* file in order for the system to know about a certain component that an app demands to be started. All components being used in the app needs to be declared in this file and the file itself needs to be in the root directory of the project [4].

In addition to keeping track of the components used in the app, the manifest file is also responsible for identifying all the permissions that the app requires. It also has to state the hardware and software requirements of the apps.

This file is of particular interest for our paper, as it contains the static information we need to form our feature sets.

3 METHODOLOGY

3.1 Data Collection and Pre-processing

For the purpose of our project, we sampled 1000 APKs, of which 500 were benign and 500 were malicious. As we were interested in viewing the manifest file, we first reverse-engineered the APKs using APKTool's decode functionality [3].

Once we had the decompiled applications, we parsed through all the manifest files. Inside the manifest file, we were interested in the 'uses-permission' nodes, the 'action' nodes under intent-filters, and the 'metadata' nodes. Three separate lists were then formed comprising of all the unique values for each set of features. Those values were then one-hot encoded to form columns of the respective dataframes.

The target vector or the label for each of the three dataframes is a column named 'malicious' which took the value 1, if the app was malicious, otherwise 0. Finally all three dataframes were saved to csv files - permissions.csv, actions.csv, and metadata.csv for training and testing our machine learning classifiers.

All the NaN values in the dataframe were filled with 0s, meaning that the particular feature was absent in the app's manifest file. For example, if a sample has 0 under one of the permission columns, it means that said app does not request that particular permission. Any row with zeroes for all columns was dropped. Any column with all zero values was dropped too.

3.2 Classification models

The classification models used in our project are - Random Forest, Decision Tree and Support Vector Machine. These models were implemented in the project using the scikit-learn library. The following sections describe the basic functionality of each of these:

3.2.1 Decision Tree Classifier: As the name suggests, decision tree classifiers form a tree like structure from the training data. At each node of the tree, there is a decision to be made which determines the branch that will be taken next [14]. The input vector traverses down the tree and at each node the split is done in a way that the resulting groups are as different from one another as possible. Finally, at the bottom of the tree, a class prediction is made for the input. A simple decision tree classifier has been shown in Figure 1 [12].

3.2.2 Random Forest Classifier: This is an ensemble classifier that employs multiple decision trees, each of which is responsible for predicting a class. When classifying a new object, its features are passed through each of these trees which then all predict a suitable class for the input vector. In random forest, the class with the most number of votes is finally assigned to the new input. A random forest classifier has been shown in Figure 2 [1].

3.2.3 Support Vector Classifier: The idea behind this model is to find a hyperplane that divides the training samples into classes while maximizing the margin, i.e. the distance between data points belonging to both classes. The points closest to the hyperplane are also points lying on the margin, and these are called support vectors. A support vector classifier has been shown in Figure 3 [2], depicting the margins on each side of the hyperplane and the support vectors that lie on those margins.

3.3 K-fold Cross Validation

To validate the performance of our models and ensure good generalization, we used k-fold cross validation with $k = 5$. The way k-fold cross validation works is that it randomly splits the given samples into k equal parts, and then uses k-1 parts as the training group and the remaining 1 part as a testing group. It then rotates the data, so that in the

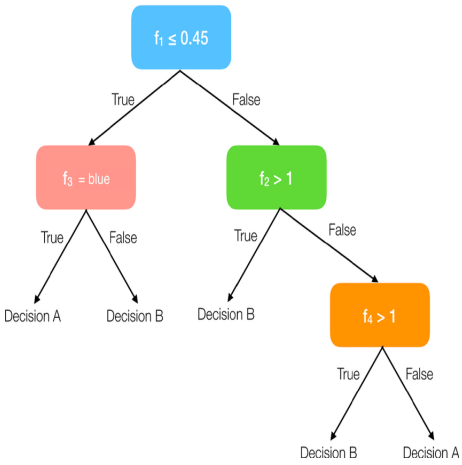

Fig. 1. A Simple decision tree classifier

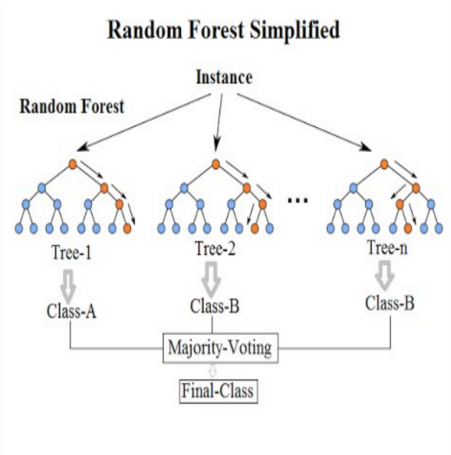

Fig. 2. Random Forest Classifier

next iteration, a new fold is used for testing and the other k-1 folds (which now includes the fold that was used for testing in the previous iteration) are used as training samples. The process repeats for k times, at the end of which every subsample has had the chance to be in the test set [5]. The working of k-fold cross validation has been shown in Figure 4.

In our case, since we used $k = 5$, we retained $k - 1 = 4$ subsamples for training and 1 subsample for testing, in each iteration. This corresponds to a $80 - 20$ split of data, where 20% of data is reserved for testing.

Scikit-learn's cross validation method was used for the purpose of fitting the models to our data and getting the relevant scores for our model. We collected four scores for each model's performance on each dataset - F1-score, Accuracy, Recall and Precision.

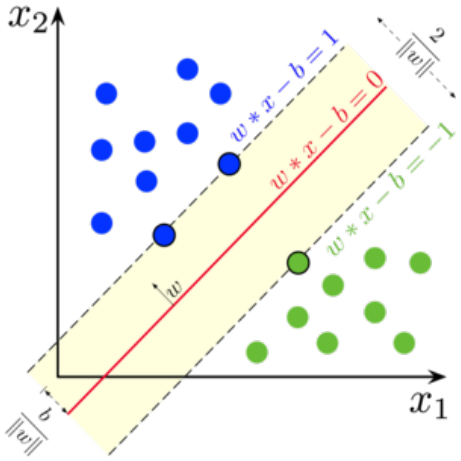

Fig. 3. Support Vector Classifier

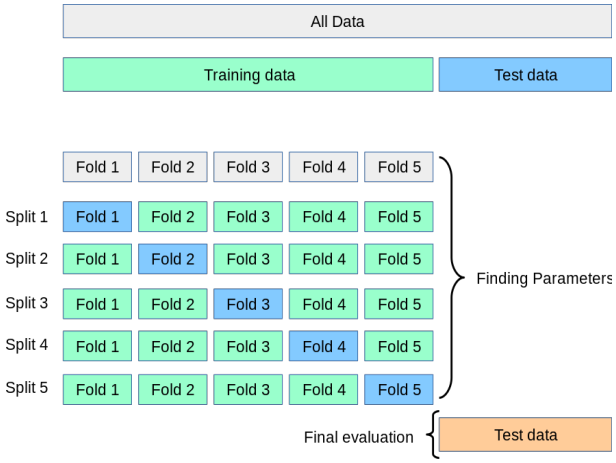

Fig. 4. K-Fold Cross Validation

3.4 Data Visualization

A confusion matrix was used for visualizing the models' performance after they made predictions on the test dataset. A confusion matrix or an error matrix lays out the data in a grid form such that each row represents the instances belonging to the actual class whereas each column represents instances belonging to the predicted class.

In our project, we label malicious apps as belonging to class 1 and benign apps as belonging to class 0. So, accordingly, each cell in the confusion matrix would be -

- Top left (True negative) - app was predicted benign and is actually benign
- Top right (False positive) - app was predicted malicious but is actually benign
- Bottom left (False negative) - app was predicted benign but is actually malicious
- Bottom right (True positive) - app was predicted malicious and is actually malicious

Table 1. Model performance comparison on permission feature set

Scores	Random Forest	Decision Tree	Support Vector
F1-Score	94.96%	92.45%	94.73%
Accuracy	94.96%	92.49%	94.75%
Precision	93.52%	91.00%	93.57%
Recall	96.64%	94.12%	96.02%

Using the above definitions, the F1-score, accuracy, precision and recall were calculated as per the following equations:

$$
\begin{aligned}
Recall &= \frac{True\ Positive}{True\ Positive + False\ Negative} \\
Precision &= \frac{True\ Positive}{True\ Positive + False\ Positive} \\
Accuracy &= \frac{True\ Positive + True\ Negative}{True\ Pos. + True\ Neg. + False\ Pos. + False\ Neg.} \\
F1 - score &= 2 \times \frac{Precision \times Recall}{Precision + Recall}
\end{aligned}
\tag{1}
$$

4 RESULTS AND ANALYSIS

In this section we will compare the results obtained after running three classification models on each of the three feature sets - permissions, actions and metadata.

4.1 User Permissions

Table 1 compares the results obtained by the three models on the user permission feature set. As is evident, Random Forest gives the highest scores among the three. The confusion matrices for all three models are shown in following figures.

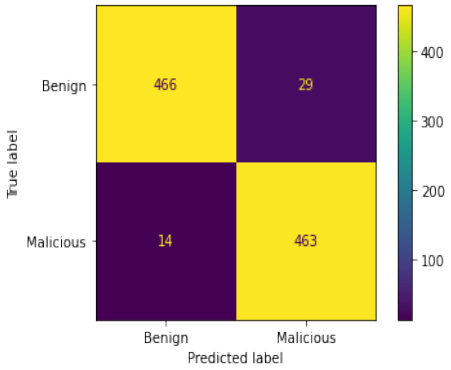

Fig. 5. Random Forest Classifier using permission based feature set

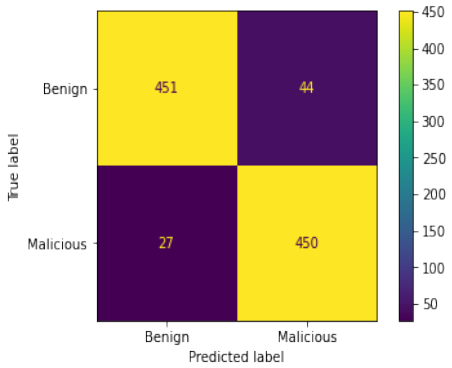

Fig. 6. Decision Tree Classifier using permission based feature set

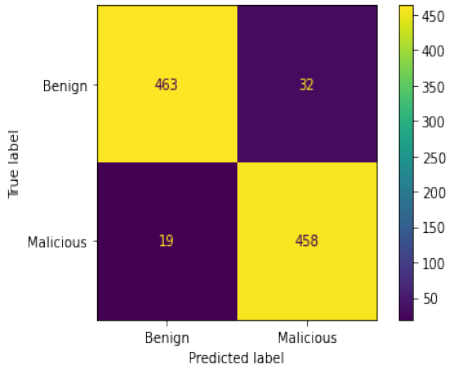

Fig. 7. Support Vector Classifier using permission based feature set

Table 2. Model performance comparison on intent action feature set

Scores	Random Forest	Decision Tree	Support Vector
F1-Score	92.23%	90.85%	91.29%
Accuracy	92.89%	91.56%	92.08%
Precision	99.27%	96.94%	98.78%
Recall	86.16%	85.53%	84.91%

4.2 Intent-Filter Actions

Table 2 compares the results obtained by the three models on the intent action feature set. Again, Random Forest gives the highest scores among the three. The confusion matrices for all three models are shown in figures 8, 9 and 10.

4.3 App Metadata

Table 3 compares the results obtained by the three models on the metadata feature set. This time Decision Tree gives the highest scores among the three. The confusion matrices for all three models are shown in the figures 11, 12 and 13.

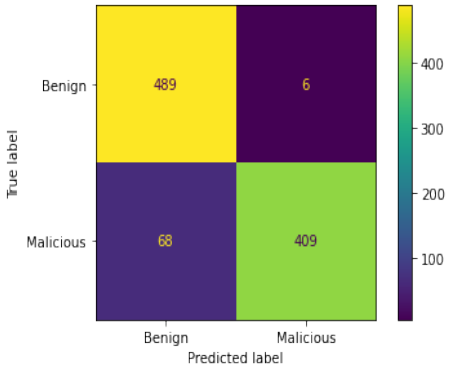

Fig. 8. Random Forest Classifier using intent action based feature set

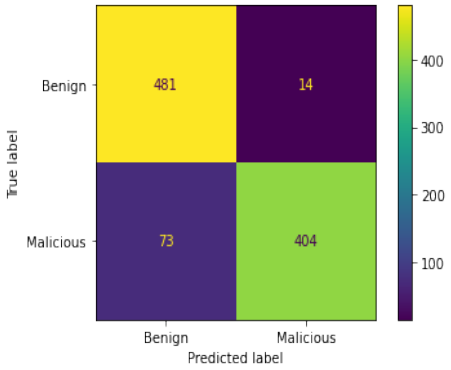

Fig. 9. Decision Tree Classifier using intent action based feature set

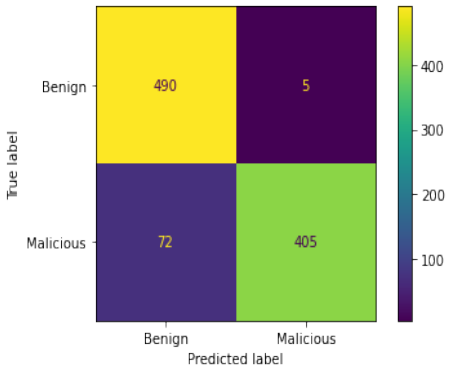

Fig. 10. Support Vector Classifier using intent action based feature set

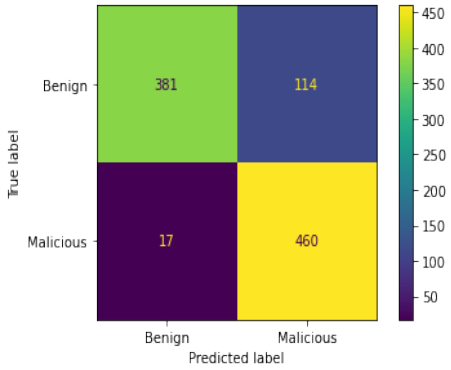

Fig. 11. Random Forest Classifier using metadata based feature set

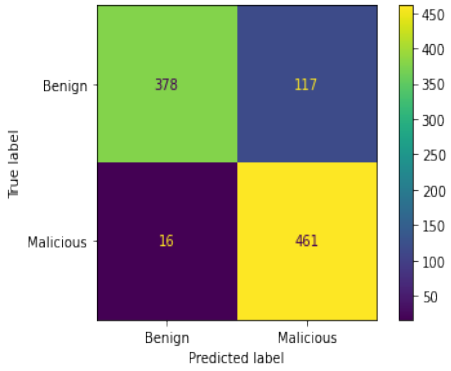

Fig. 12. Decision Tree Classifier using metadata based feature set

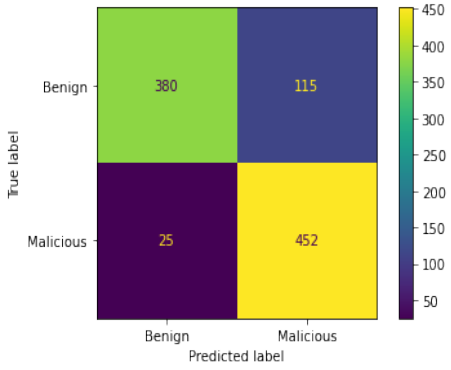

Fig. 13. Support Vector Classifier using metadata based feature set

4.4 Analysis

Of the three classification models that we have compared in this paper, Random Forest gave the highest F1-score on the user permissions and intent actions datasets. It was a close second to Decision Tree when run on the metadata dataset. Thus, Random Forest classifier gave generally good results on all three feature sets.

Table 3. Model performance comparison on metadata feature set

Scores	Random Forest	Decision Tree	Support Vector
F1-Score	87.28%	87.35%	86.60%
Accuracy	87.11%	86.21%	85.60%
Precision	79.88%	79.78%	79.86%
Recall	96.44%	96.64%	94.76%

Table 4. Comparison of results obtained in this paper and in Milosevic et al.

Model	This project	Milosevic et al.
Decision Trees	92.45%	82.7%
Random Forest	94.96%	86.50%
Support Vector	94.73%	87.90%

Out of the three feature sets used, user permissions was the best at classifying apps as malicious or benign as indicated by the high F1-score achieved by all three classification models on that dataset. On the other hand, metadata classified apps with the lowest accuracy and F1-scores.

5 RELATED WORK

In their article, Milosevic et al. [11] have performed a machine learning based classification of apps based on user permissions and source code analysis. The models they have compared are Decision Trees, Random Forest, Bayes Networks, Support Vector Machines, and Logistic Regression among others. They trained their models on the MODroid dataset that contains 200 of each benign and malicious datasets.

Although, our project uses a completely different dataset, we have compared the same models that the authors of the aforementioned paper have used for comparison. Furthermore, one of the feature sets - user permissions - is common to both papers. We can use this knowledge to compare results obtained in both papers. The comparison of F1-scores between the two papers have been shown in Table 4.

As is evident from the table, F1-scores improved upon existing work for all three models on our dataset when permission based features where used for classification.

6 CONCLUSION

In this project, we created an automated, machine learning based classification tool that is capable of classifying Android applications into the categories of benign and malicious with decent accuracy and F1-score. For this, we parsed through the manifest files of the apps and extracted out three sets of unique features which we then used to train our classification models.

We compared the performance of Random Forest Classifer, Decision Tree Classifier and Support Vector Classifier and came to the conclusion that Random Forest overall gave higher scores with the highest being 94.96% for the permission based feature set. The lowest F1-score achieved was by Support Vector Classifier on the metadata based feature set at 86.6%.

Overall, the user permission feature set resulted in the most accurate classification of apps whereas the metadata feature set resulted in the least accurate.

For further research, we may extend our project to use the CIC-AndMal2017 dataset, perform classification tasks on those samples, and compute resultant scores.

REFERENCES

[1] 2017. Random Forest. https://en.wikipedia.org/wiki/Random_forest.

[2] 2018. Support Vector Machine. https://en.wikipedia.org/wiki/Support_vector_machine.

[3] 2022. APKTool Documentation. https://ibotpeaches.github.io/Apktool/documentation/.

[4] 2022. Application Fundamentals. https://developer.android.com/guide/components/fundamentals.

[5] 2022. Cross-Validation. https://en.wikipedia.org/wiki/Cross-validation_(statistics).

[6] 2022. Global smartphone sales to end users 2007-2021. https://www.statista.com/statistics/263437/global-smartphone-sales-to-end-users-since-2007/.

[7] 2022. Market share of mobile operating systems worldwide 2009-2022. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/.

[8] Haipeng Cai, Na Meng, Barbara Ryder, and Daphne Yao. 2019. DroidCat: Effective Android Malware Detection and Categorization via App-Level Profiling. *IEEE Transactions on Information Forensics and Security* 14, 6 (2019), 1455–1470. https://doi.org/10.1109/TIFS.2018.2879302

[9] Rajesh Kumar, Zhang Xiaosong, Riaz Ullah Khan, Jay Kumar, and Ijaz Ahad. 2018. Effective and Explainable Detection of Android Malware Based on Machine Learning Algorithms. In *Proceedings of the 2018 International Conference on Computing and Artificial Intelligence (ICCAI 2018)*. Association for Computing Machinery, 35–40. https://doi.org/10.1145/3194452.3194465

[10] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. 2017. Static analysis of android apps: A systematic literature review. *Information and Software Technology* 88 (2017), 67–95. https://doi.org/10.1016/j.infsof.2017.04.001

[11] Nikola Milosevic, Ali Dehghantanha, and Kim-Kwang Raymond Choo. 2017. Machine learning aided Android malware classification. *Computers Electrical Engineering* 61 (2017), 266–274. https://doi.org/10.1016/j.compeleceng.2017.02.013

[12] Ioannis Mollas, Nick Bassiliades, and Grigorios Tsoumakas. 2022. Conclusive local interpretation rules for random forests. *Data Mining and Knowledge Discovery* 36 (07 2022). https://doi.org/10.1007/s10618-022-00839-y

[13] Mohd Faizal Ab Razak, Nor Badrul Anuar, Rosli Salleh, and Ahmad Firdaus. 2016. The Rise of Malware. *J. Netw. Comput. Appl.* 75, C (2016), 58–76. https://doi.org/10.1016/j.jnca.2016.08.022

[14] Lu Y. Song YY. 2015. Decision tree methods: applications for classification and prediction. *Shanghai Arch Psychiatry* 27, 2 (2015), 130–5. https://doi.org/10.11919/j.issn.1002-0829.215044

[15] Tang Y. Yuan, H. 2020. MADFU: An Improved Malicious Application Detection Method Based on Features Uncertainty. *Entropy (Basel, Switzerland)* 22, 7 (2020), 792. https://doi.org/10.3390/e22070792

Received 7 December 2022