UNIVERSITÀ DI PISA

Dipartimento di Ingegneria dell'Informazione

Master of Science in Artificial Intelligence and Data Engineering

Data Mining and Machine Learning

Android Malware Detection Using Machine Learning

Student: TEMFACK DERICK

Examiners: Prof. Francesco Marcelloni, Alessandro Renda
University of Pisa, Italy

Academic Year 2023-2024

Contents

1. Introduction

The Android operating system (OS), powered by Google, has dominated the mobile platform since 2012, surpassing 70% of smartphone shipments since 2017 [1]. With iOS, they account for over 99% of the mobile OS market. Android's widespread use and vast data make it a prime target for cyberattacks. Despite malware attacks stabilizing since a 2016 spike, new threats, including ransomware, emerge daily [2, 3]. Over 480,000 new Android malware samples were detected monthly in 2020 [4], but undetected malware is likely more prevalent [5]. The malware even infiltrates Google Play, affecting millions of devices [6]. Despite security measures like Google Play Protect and Samsung Knox, malware creators still bypass defences [7, 8]. However, machine learning (ML) has shown great promise in detecting even sophisticated malware, including zero-day, repackaged, and obfuscated types.

This project aims to build an effective classification model to classify a mobile application as **Benign** or **Malware**. To do so, we'll evaluate multiple classification models using different metrics and select the best model with better performance for our dataset.

2. Exploratory Data Analysis

2.1 Dataset Overview

The dataset used in this project, hosted on FigShare, contains feature vectors of 215 distinct attributes gathered from 15,036 mobile applications-5,560 classified as malware from the Drebin project and 9,476 as benign. It is structured with 215 columns and 15,036 rows, designed for binary classification where the target variable differentiates between Malware (S) and Benign (B) apps. Each attribute is encoded in binary format: 0 indicates an attribute's absence, while 1 denotes its presence. The class distribution is presented in Figure 2.1.

The 215 features of our dataset are divided into four different categories: **API Call Signature, Manifest Permission, Intent, Commands signature** (See Figure 2.2).

2.2 API Call Signature

There are 72 features in the category API Call Signature. It represents the signature of a method in the Java API. An example is 'android.os.Binder' which is a fundamental class in Android development used for Interprocess Communication (IPC). It allows different processes to communicate with each other, enabling one app to call methods on another app or service. Another example is 'Ljava.lang.Class.getResource' which is used to locate a resource with a given name. This method returns a URL object that represents the resource, or null if the resource cannot be found.

2.3 Manifest Permission

There a 113 features in this category and are declared in the manifest file (AndroidManifest.xml). These permissions are used to control access to protected parts of the system or other apps. Declaring permissions ensures that the app has the necessary access to perform its functions while maintaining security and privacy. For example, SEND_SMS permission in Android allows an app to send SMS (text) messages on behalf of the user while ACCESS_FINE_LOCATION permission enables an app to access precise location data of the phone from location providers such as GPS.

2.4 Intent

There are 23 features in this category. An Intent in Android is a messaging object that is used to request an action from another app component. It's a fundamental part of the Android system that helps different components of an application (or even different applications) interact

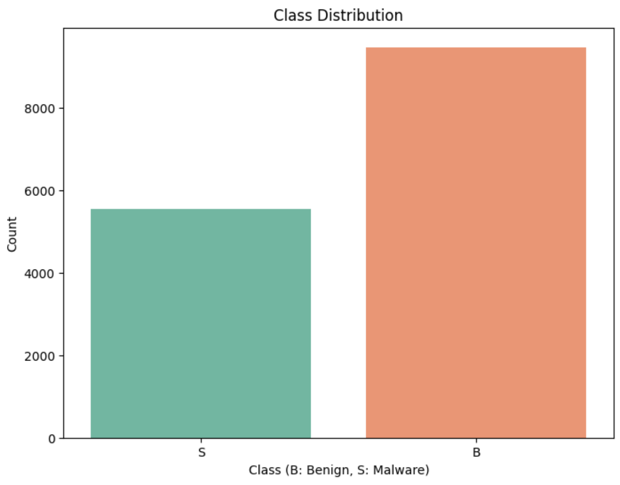


Figure 2.1: Class Distribution

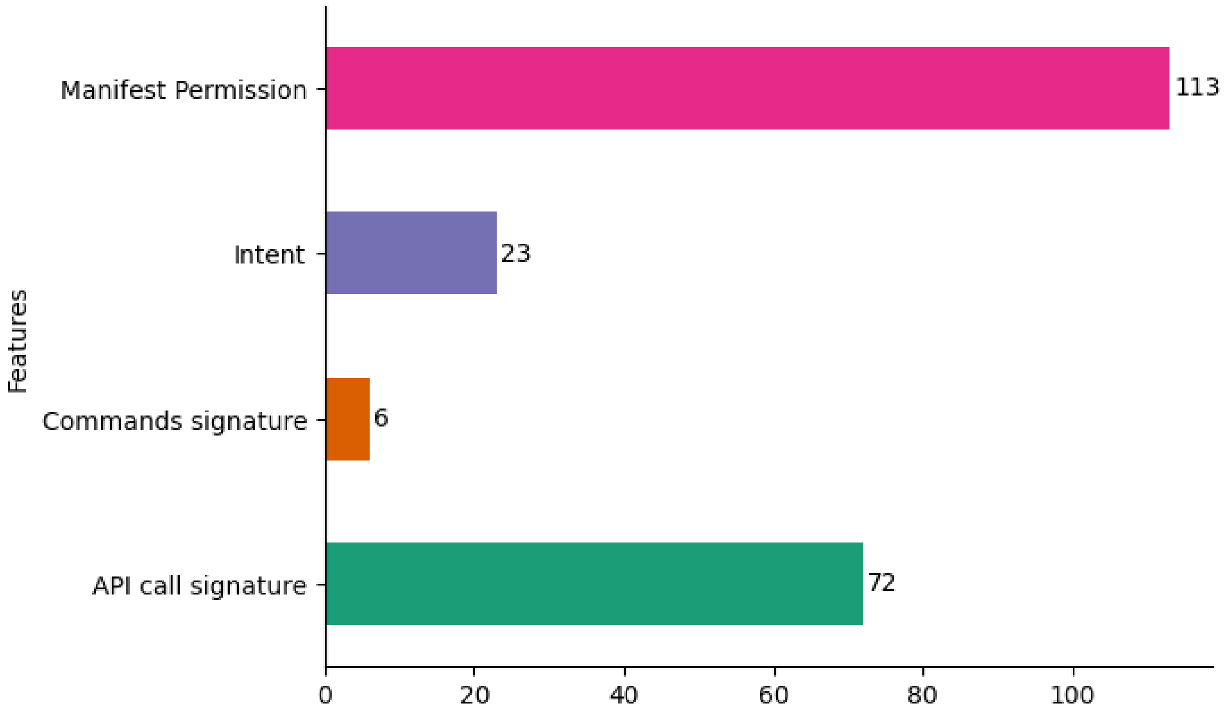


Figure 2.2: Group of Features

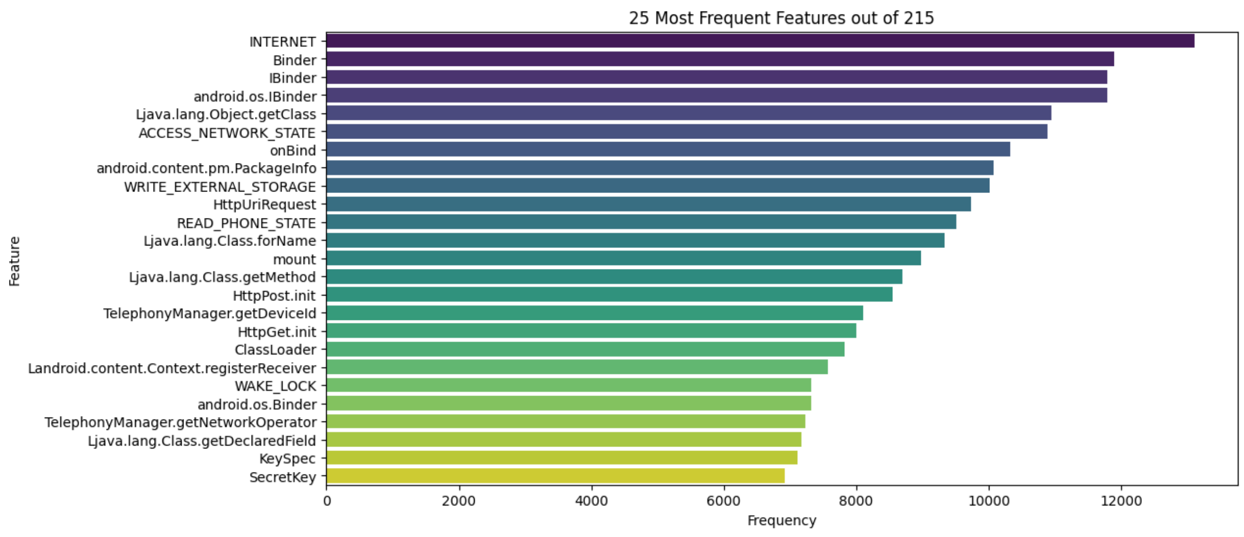


Figure 2.3: Top 25 most frequent features

with each other. For instance, the intent android.intent.action.BATTERY_LOW is an intent action that indicates the battery level of the device is low. When the system broadcasts this intent, it signals that the device's battery is running low and the user should take action to preserve battery life, such as reducing power consumption or charging the device. Another intent android.intent.action.NEW_OUTGOING_CALL is used in Android to broadcast when a new outgoing call is being placed. This broadcast can be intercepted by other apps to monitor, modify, or block outgoing calls.

2.5 Commands signature

There are 6 features in this category. They are commands used in the Android system. For instance, chmod Changes the permissions of a file or directory while chown Changes the ownership of a file or directory.

2.6 Top 25 most frequent and less frequent features

Figure 2.3 presents 25 most frequent features while figure 2.4 presents 25 less frequent features.

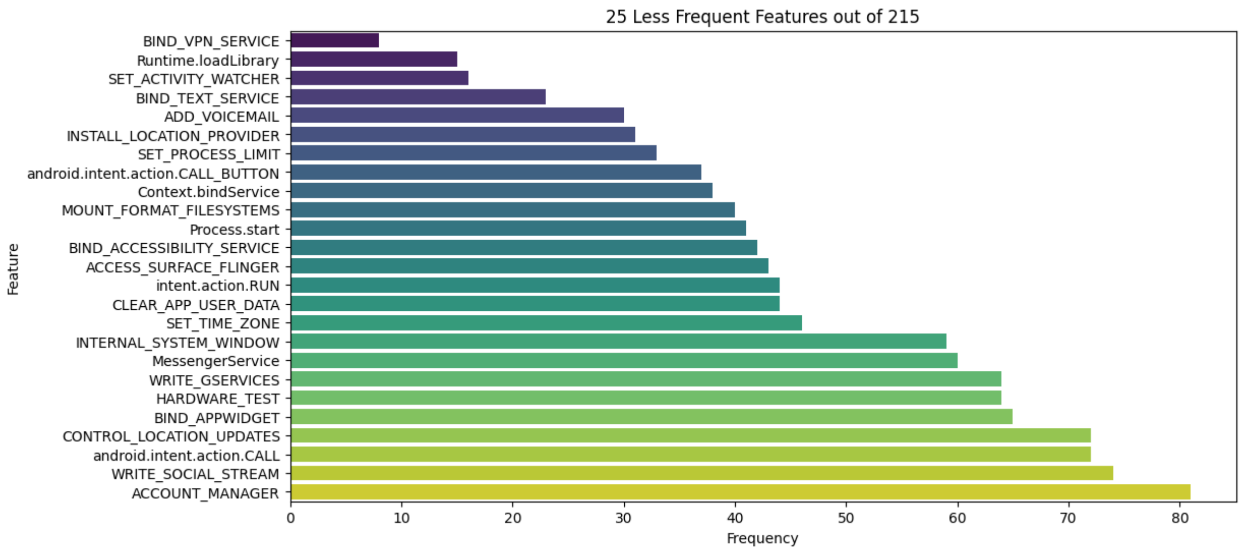


Figure 2.4: Top 25 less frequent features

3. Data Cleaning, Transformation and Splitting

3.1 Data Cleaning and Transformation

To clean and transform the dataset, we began by examining the data types, noting that most columns were of type `int64`. We identified columns with `object` types, specifically TelephonyManager.getSimCountryIso and `class`. While `class` being an `object` type was expected, TelephonyManager.getSimCountryIso had inconsistent values, including `'0', '1', '?', 1, 0`, indicating a mix of strings and integers, with some missing data represented by the placeholder `'?'`. We addressed this by filtering out rows with missing data in `TelephonyManager.getSimCountryIso` since they were minimal, and then converting the remaining values to integers. We also transformed the `class` column from categorical values (`'B'` for benign and `'S'` for malware) to numerical values, with `'B'` mapped to 0 and `'S'` mapped to 1, to make it suitable for modelling. The dataset was then verified to ensure that these transformations were applied successfully, leaving it clean and ready for further analysis.

3.2 Data Splitting

Before performing further operations on our dataset, we divide it into a training set and a testing set, ensuring the testing data remains unseen during model training. We allocate 80% of the data for training and 20% for testing to maintain a clear separation between the training and evaluation phases. This split helps prevent data leakage and allows us to evaluate model performance accurately on unseen data, contributing to a more robust and reliable model for deployment. After the split, we verified the shape of each dataset portion, confirming the division into a training set with 12,024 samples and a test set with 3,007 samples across 215 features, aligning with the desired 80-20 ratio.

4. Models comparison

Several models were cross-validated to evaluate the performance of various machine learning models for Android Malware classification using a 5-fold RepeatedStratifiedKFold. The models tested included:

- Random Forest
- XGBoost
- LightGBM
- Extra Tree Classifier
- Logistic Regression
- Support Vector Machine
- AdaBoost
- Decision Tree
- Bagging
- Bayesian

4.1 Before Under-sampling

The table 4.1 presents the performance of each model across the different metrics.

Rank	Model	Accuracy	Precision	Recall	F1 Score	ROC AUC
1	XGBoost	0.988662	0.988112	0.981055	0.984571	0.998568
2	LightGBM	0.988495	0.987516	0.981206	0.984351	0.998513
3	Extra Tree Classifier	0.988440	0.991681	0.976846	0.984208	0.998383
4	Random Forest	0.987830	0.991592	0.975267	0.983362	0.998149
5	Support Vector Machine	0.982008	0.985944	0.964968	0.975343	0.997424
6	Logistic Regression	0.976852	0.975803	0.961058	0.968375	0.996192
7	Bagging	0.981759	0.980030	0.970305	0.975144	0.995173
8	AdaBoost	0.964959	0.960943	0.943317	0.952049	0.993118
9	Decision Tree	0.973109	0.958166	0.969403	0.963752	0.972323
10	Bayesian	0.708943	0.560313	0.978800	0.712663	0.776001

Table 4.1: Comparison of Model Performance Metrics without Sampling

Additionally, Figure 4.1 presents the confusions matrices for all our models before sampling and Figure 4.2 presents the ROC Curves.

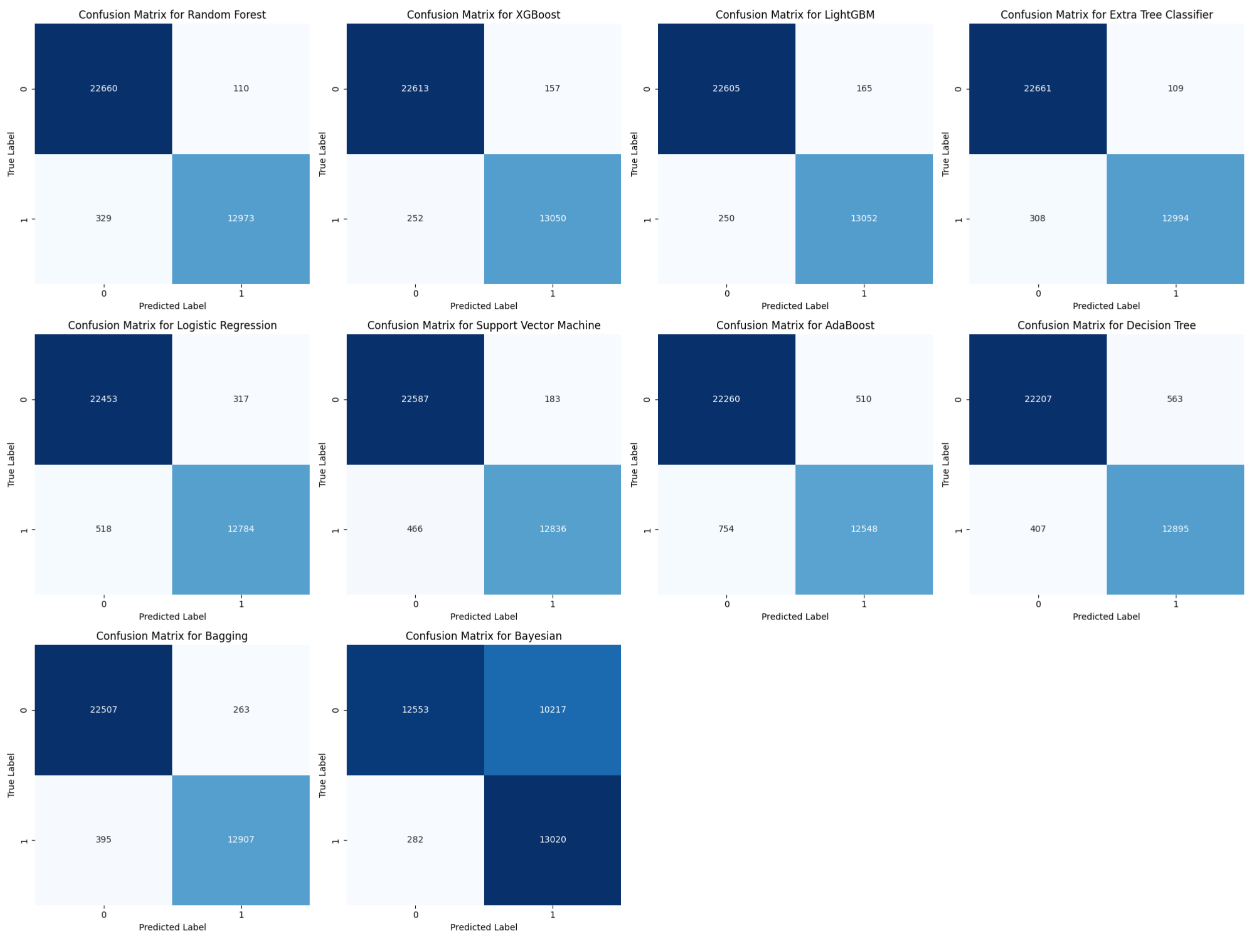


Figure 4.1: Confusion Matrix of Models without Sampling

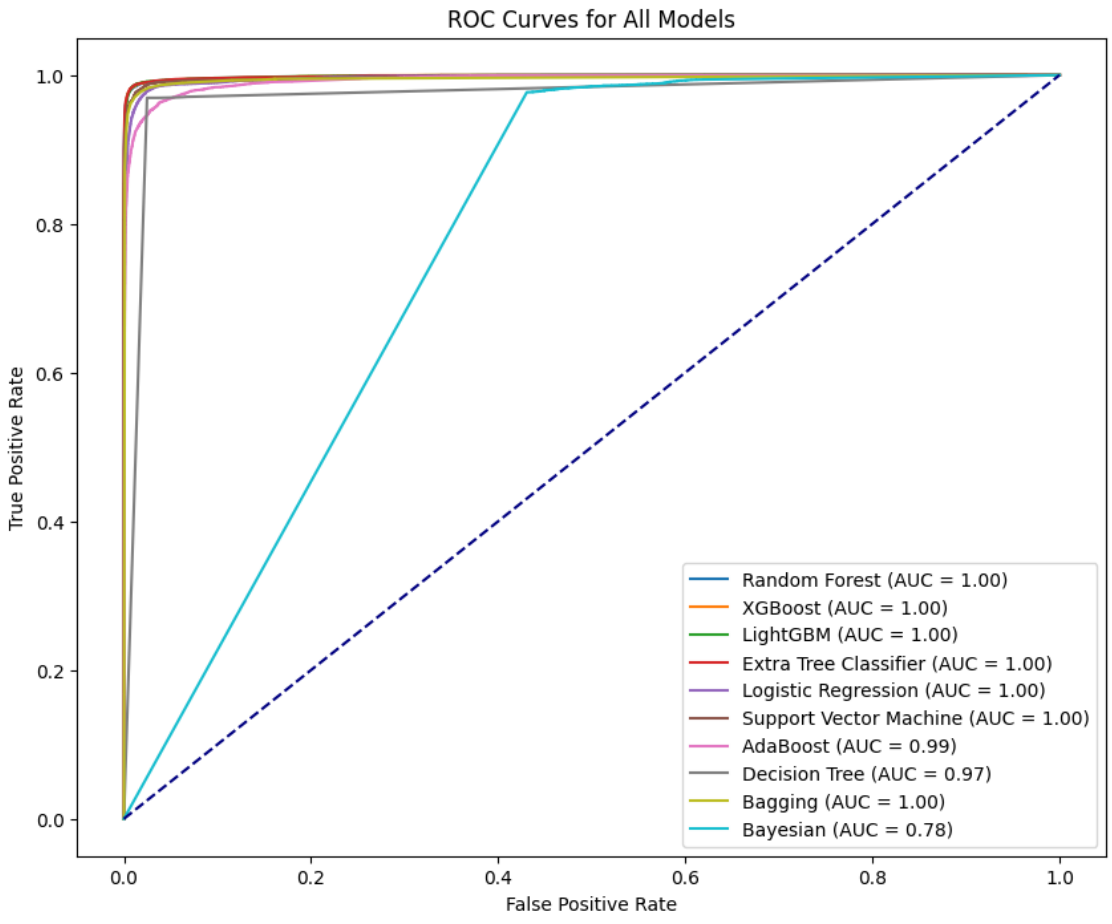


Figure 4.2: ROC Curves for All Models Before Sampling

4.2 After Under-sampling

The table 4.2 presents the performance of each model across the different metrics after the dataset has been under-sampled.

Rank	Model	Accuracy	Precision	Recall	F1 Score	ROC AUC
1	XGBoost	0.986468	0.987862	0.985040	0.986449	0.998527
2	LightGBM	0.986280	0.989186	0.983311	0.986239	0.998391
3	Extra Tree Classifier	0.986506	0.992841	0.980078	0.986418	0.998062
4	Random Forest	0.986167	0.992461	0.979777	0.986079	0.997941
5	Support Vector Machine	0.980567	0.988238	0.972711	0.980413	0.997605
6	Logistic Regression	0.974816	0.979502	0.969929	0.974692	0.996493
7	Bagging	0.976808	0.980020	0.973463	0.976730	0.994427
8	AdaBoost	0.962186	0.965546	0.958578	0.962049	0.993927
9	Decision Tree	0.968727	0.964882	0.972861	0.968855	0.969079
10	Bayesian	0.767779	0.688087	0.979627	0.808375	0.780607

Table 4.2: Comparison of Model Performance Metrics after Under-Sampling

Additionally, Figure 4.3 present the confusions matrices for all our models before sampling and Figure 4.4 present the ROC Curves.

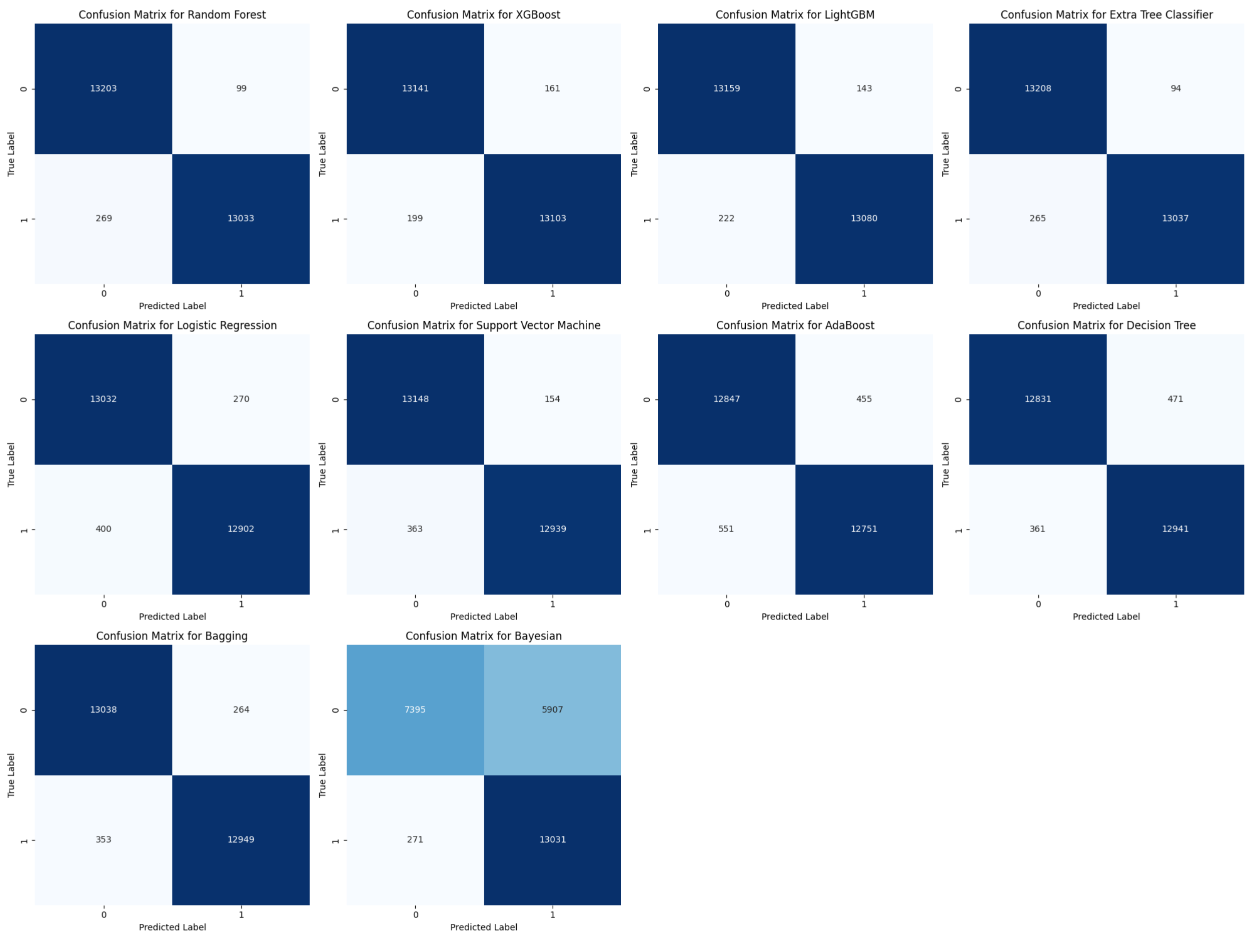


Figure 4.3: Confusion Matrix of Models After Under-Sampling

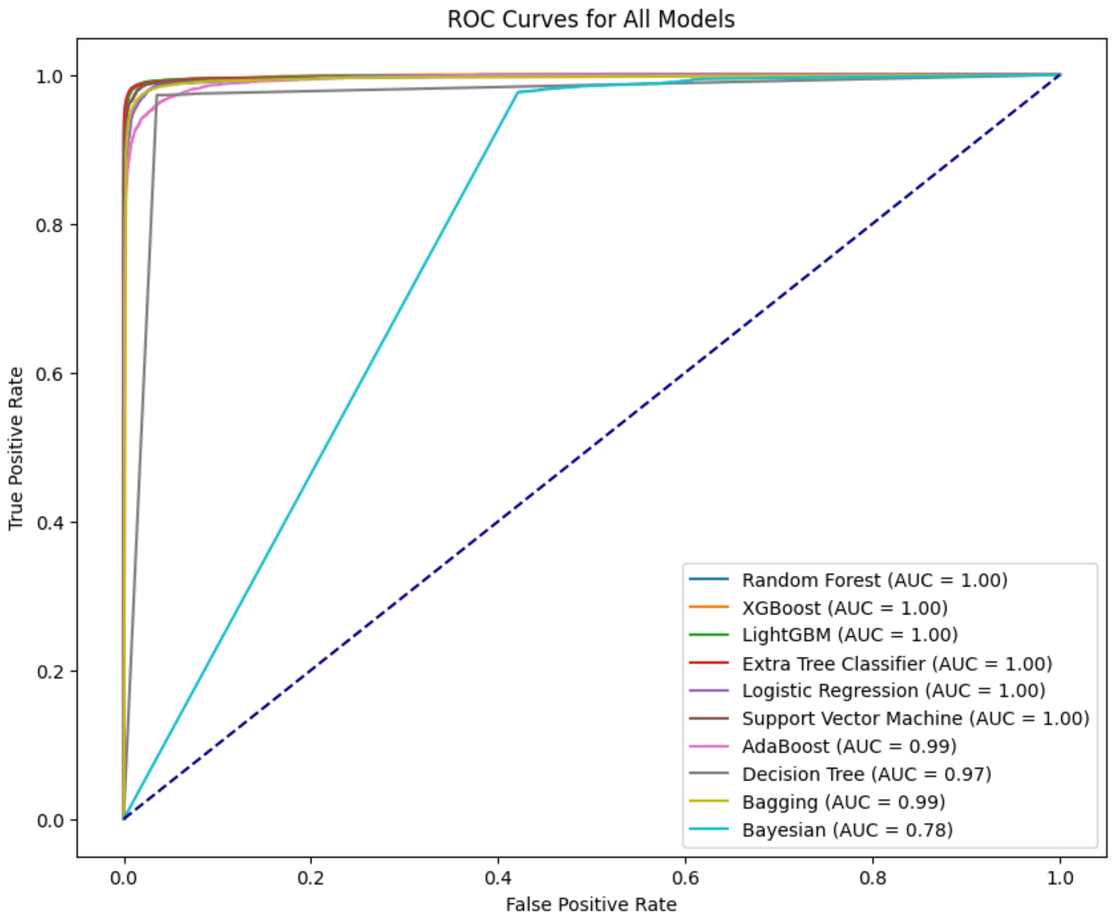


Figure 4.4: ROC Curves for All Models after Under Sampling

5. Model Selection and Hyper-parameter Tuning

5.1 Model Selection

After examining the performance metrics after the k-fold cross-validation, both before sampling and after under-sampling, we can conclude that XGBoost archives the best performance in both scenarios followed by LightGBM and Extra Tree Classifier. So we decide to continue with XG-Boost.

5.1.1 Evaluation of XGBoost without Sampling. After selecting XGBoost, we trained and evaluated it and there the results are presented in table 5.1 and the confusion matrix and ROC Curve are available in figure 5.1.

Model	Accuracy	Precision	Recall	F1 Score	ROC AUC
XGBoost Training	0.998087	0.999095	0.995715	0.997402	0.999825
XGBoost Test	0.987695	0.990054	0.976806	0.983386	0.998446

Table 5.1: Performance metrics for XGBoost model on Training and Test sets Before Sampling.

5.1.2 Evaluation of XGBoost After Under-sampling. We also trained and evaluated XGBoost after under-sampling the dataset. The results are presented in table 5.2 and the confusion matrix and ROC Curve are available in figure 5.2.

5.2 Hyper-Parameter Tuning

5.2.1 Model Improvement with Hyperparameter Tuning. A hyperparameter tuning process was conducted using GridSearchCV to improve the performance of the XGBoost model. The

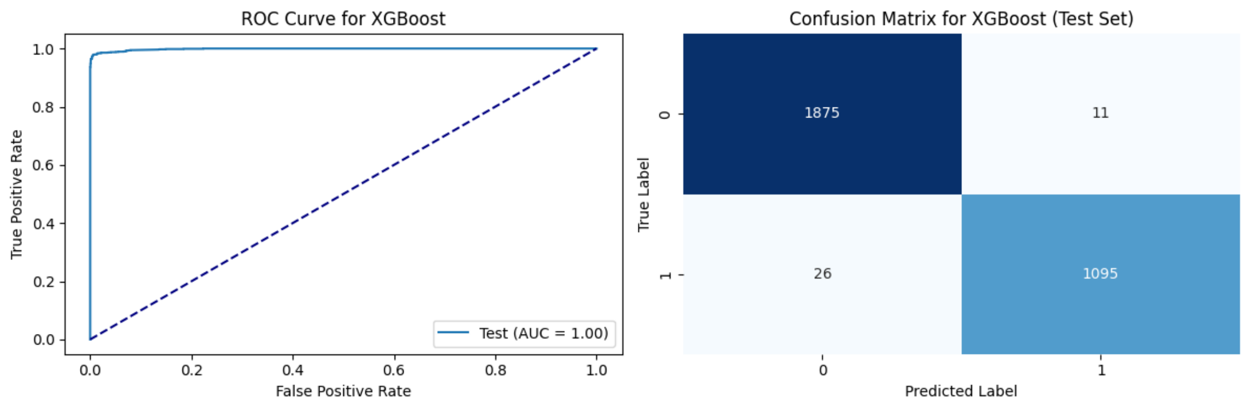


Figure 5.1: ROC Curve and Confusion Matrix Before sampling

Model	**Accuracy**	**Precision**	**Recall**	**F1 Score**	**ROC AUC**
XGBoost Training	0.99501	0.990579	0.995940	0.993252	0.999570
XGBoost Test	0.98437	0.976909	0.981267	0.979083	0.998569

Table 5.2: Performance metrics for XGBoost model with undersampling on Training and Test sets.

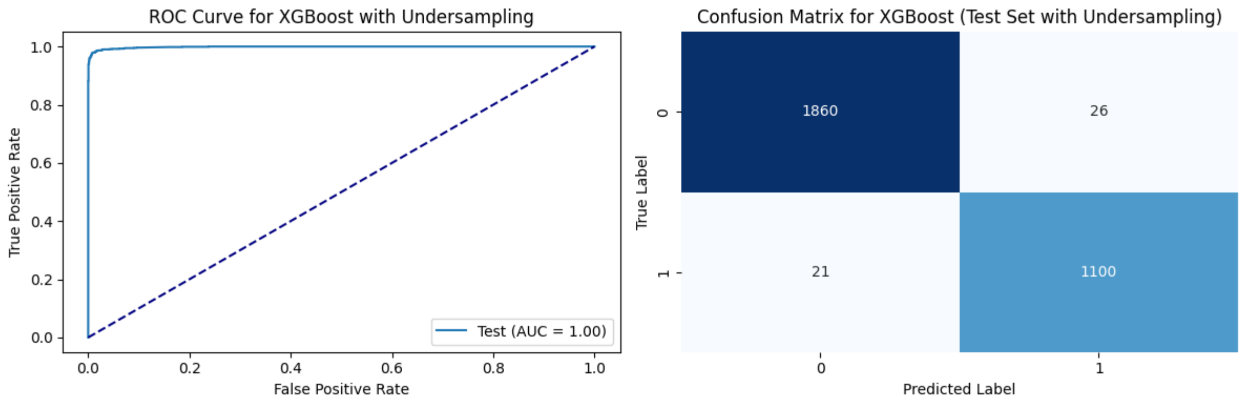


Figure 5.2: ROC Curve and Confusion Matrix After Under-Sampling

following hyperparameters were optimized:

- **n_estimators:** The number of trees in the forest.
- **max_depth:** The maximum depth of the tree.
- **learning_rate:** The step size shrinkage used in the update to prevent overfitting.
- **subsample:** The fraction of samples for each tree.
- **colsample_bytree:** The fraction of features to be used for each tree.

The grid search was performed with multiple scoring metrics: ROC AUC and Recall. The best hyperparameters were selected based on the highest recall score. The fine-tuned XGBoost model was then retrained on the entire training set and evaluated on the test set. The evaluation metrics for the test set are presented in the table 5.3. Additionally, the ROC Curve and the confusion matrix are presented in Figure 5.3.

Table 5.3: Evaluation Metrics for Fine-Tuned XGBoost Model

Model	**Accuracy**	**Precision**	**Recall**	**F1 Score**	**ROC AUC**
XGBoost Fine-Tuned	0.986698	0.98914	0.975022	0.982031	0.998764

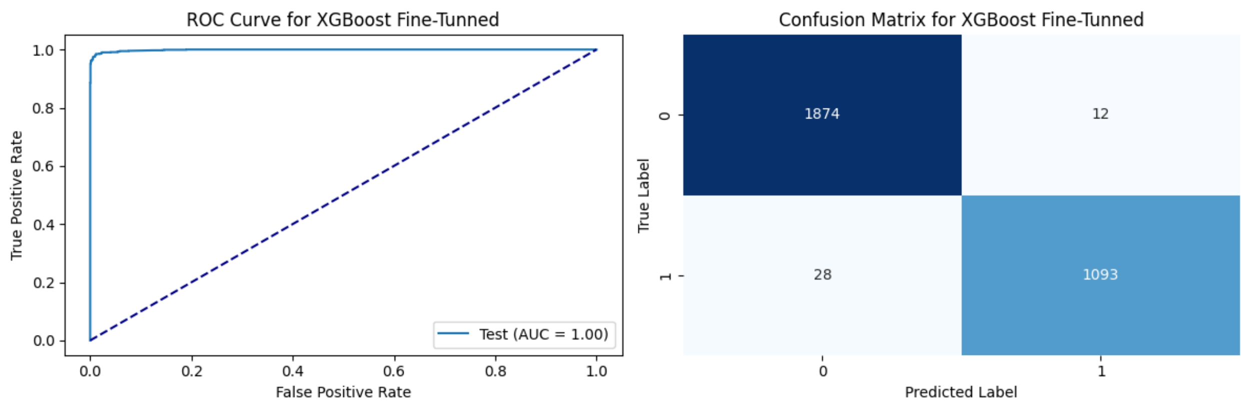


Figure 5.3: ROC Curve and Confusion Matrix XGBoost Fine-Tuned

6. Deployment

To deploy our model, we package everything within a Docker container and expose the model as an API. When a user wants to make a prediction, they submit an APK to the API. The first step in the process involves reverse-engineering the APK to extract all the features necessary for the prediction. These features are then used to determine the status of the application. The complete workflow is illustrated in Figure 6.1. Figure 6.2 and 6.2 present the executions of the WhatsApp APK using our app.

To have access to the application, you have to follow the following steps:

1. Have Docker installed on your computer.
2. Run the following command: `docker run -p 8080:8000 tderick/android-malware-detection`
3. Go to `http://localhost:8080/docs` to test the application.

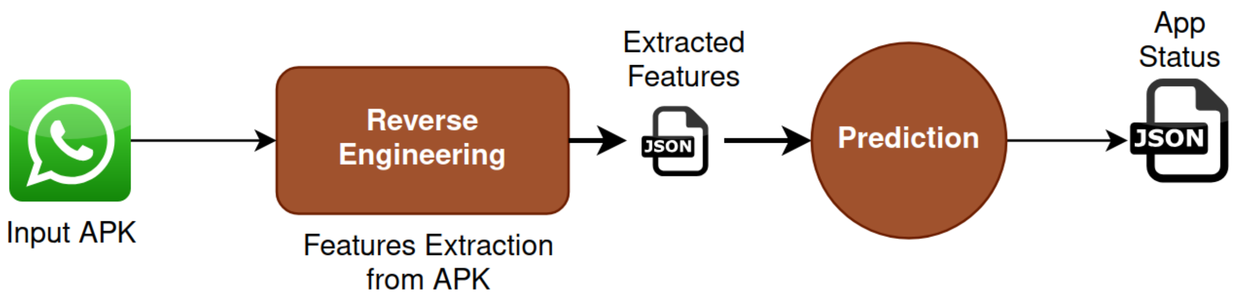
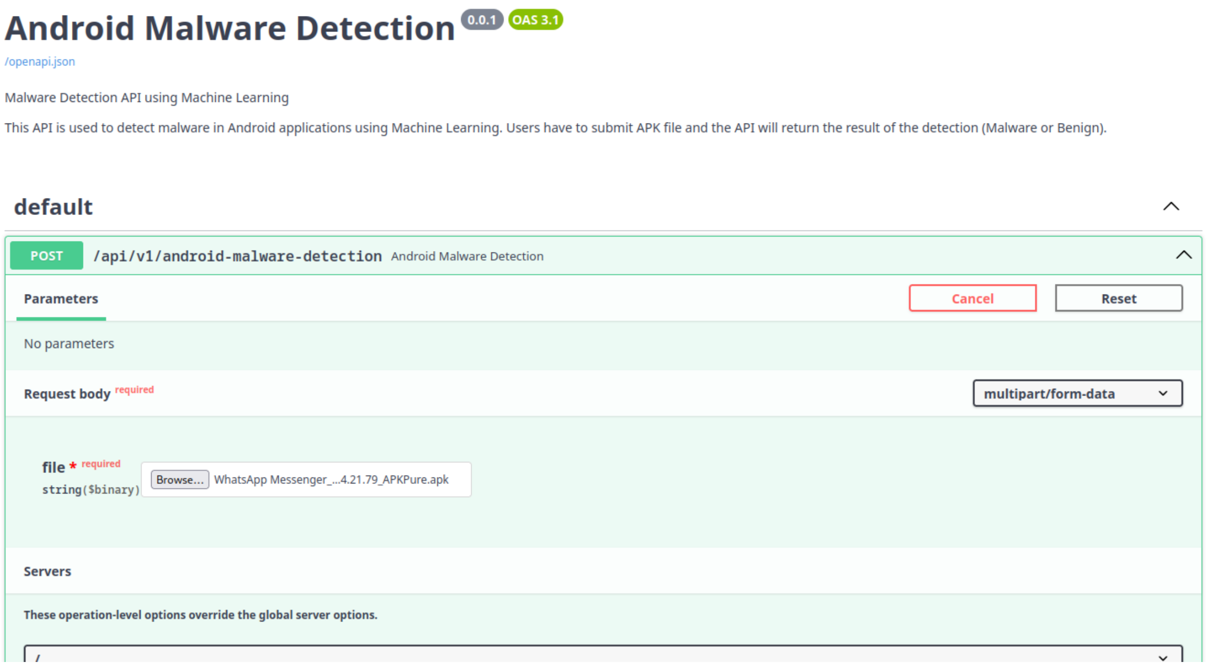


Figure 6.1: Prediction Workflow



Android Malware Detection 0.0.1 OAS 3.1

/openapi.json

Malware Detection API using Machine Learning

This API is used to detect malware in Android applications using Machine Learning. Users have to submit APK file and the API will return the result of the detection (Malware or Benign).

default

POST `/api/v1/android-malware-detection` Android Malware Detection

Parameters

Cancel Reset

No parameters

Request body required

multipart/form-data

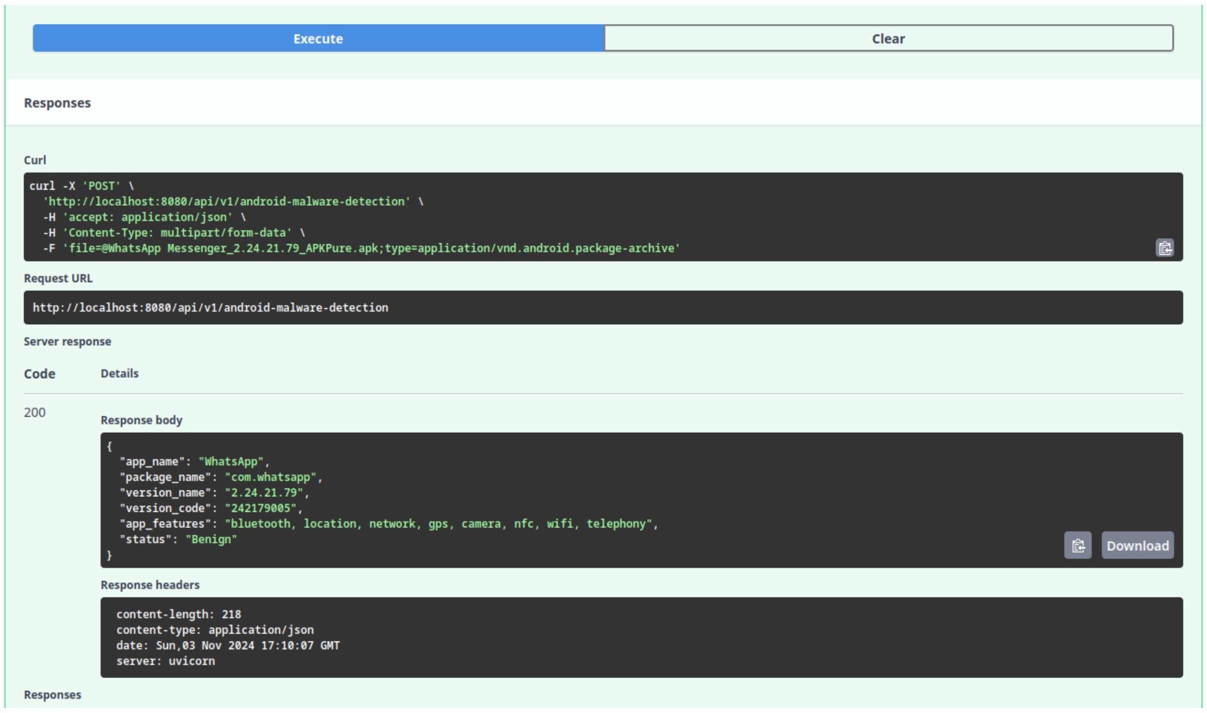
file * required
string($binary)

Browse... WhatsApp Messenger_...4.21.79_APKPure.apk

Servers

These operation-level options override the global server options.

Figure 6.2: WhatsAPP APK Analysis



Execute Clear

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:8080/api/v1/android-malware-detection' \
  -H 'accept: application/json' \
  -H 'Content-Type: multipart/form-data' \
  -F 'file=@WhatsApp Messenger_2.24.21.79_APKPure.apk;type=application/vnd.android.package-archive'
```

Request URL

```
http://localhost:8080/api/v1/android-malware-detection
```

Server response

Code Details

200

Response body

```
{
  "app_name": "WhatsApp",
  "package_name": "com.whatsapp",
  "version_name": "2.24.21.79",
  "version_code": "242179005",
  "app_features": "bluetooth, location, network, gps, camera, nfc, wifi, telephony",
  "status": "Benign"
}
```

Download

Response headers

```
content-length: 218
content-type: application/json
date: Sun,03 Nov 2024 17:10:07 GMT
server: uvicorn
```

Responses

Figure 6.3: WhatsAPP APK Analysis

7. Conclusion

In this project, we trained a classification model to determine whether an Android application is malware. To achieve this, we evaluated a variety of machine-learning classification models and ultimately chose XGBoost as the best option for our dataset. We deployed our model as a REST API for production use and packaged everything inside a Docker container.

Bibliography

[1] Statista, 2021a. Global market share held by mobile operating systems since 2009. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009

[2] Chebyshev, 2019. Mobile malware evolution 2019. https://securelist.com/mobile-malware-evolution-2019/96280

[3] Microsoft, 2020. Sophisticated new Android malware marks the latest evolution of mobile ransomware. https://www.microsoft.com/en-us/security/blog/2020/10/08/sophisticated-new-android-malware-marks-the-latest-evolution-of-mobile-ransomware/

[4] Statista, 2021b. Global Android malware volume. https://www.statista.com/statistics/680705/global-android-malware-volume

[5] Broersma, 2020. Sophisticated Android malware. https://www.silicon.co.uk/workspace/android-sophiticated-malware-344222

[6] McGowan, 2020. New malware apps on Google Play. https://blog.avast.com/new-malware-apps-on-google-play-avast

[7] Cimpanu, 2019. Gustuff Android banking trojan targets 100 banking, IM, and cryptocurrency apps. https://www.zdnet.com/article/gustuff-android-banking-trojan-targets-100-banking-im-and-cryptocurrency-apps

[8] Lakshmanan, 2020. Joker Android mobile virus. https://thehackernews.com/2020/07/joker-android-mobile-virus.html