# Security Risk Assessment Using Smart Edge Computing for Sets of Android Applications

Keywords: Android Operating System, Machine Learning, Edge Computing, Security Risk, Android Security

Abstract: Mobile devices are a widely used vector for cyberattacks in current times and the Android Operating System has a high market share of current devices. One such attack is when two applications, seemingly benign on their own, can communicate themselves and gather and send important information to a third party. We addressed the problem of analyzing groups of applications that communicate with each other from a security risk perspective. We created a comparative analysis of different machine learning algorithms that can estimate the risk an application poses and we integrated it into a usable tool that groups installed packages and extracts the input features based on the values in the AndroidManifest.xml file. We made use of Edge Computing as all the computations for the risk analysis take place locally on the mobile device, without communicating with a server or requiring internet access, thereby avoiding network interception and reducing data consumption.

## 1 INTRODUCTION

In today's world, smartphones are integral to daily life, with the Android Operating System(Google Corporation, ) commanding 70% of the global market share(Statcounter GlobalStats, ). This highlights the urgent need for effective tools to protect users from malware. Cyberattacks via malicious apps often compromise sensitive data such as banking details and credentials. Additionally, inter-app communication can enable coordinated malicious activities, where seemingly harmless data exchanges can become significant security threats.

This paper aims to enhance user awareness of mobile device cybersecurity by clarifying possible app interactions and estimated security risks. Motivated by the need for accessible tools for non-experts, we analyzed AndroidManifest.xml files, grouped apps, assessed risk via machine learning, and combined these features into a single tool.

By creating an Android application that executes all necessary computations on the device, without any server communication, we ensure that the data we collect cannot be intercepted by malicious third parties as it is never stored or sent over a network. As demonstrated in (Holbrook and Alamaniotis, 2019), the increase of Internet of Things (IoT) devices has led to an increase in network security attacks and breaches. This structure is also conducive to the edge-computing model, as can be seen by the fact that the computations are done on the device on which our application is installed.

While there have been other papers that have treated the subject of static analysis of mobile applications from a security perspective, the majority either use tools that aren't available on mobile platforms (Feizollah et al., 2017) (Khariwal et al., 2020) or do not specify what they used for extracting data from apks (Idrees et al., 2017), (Shrivastava and Kumar, 2019). Another branch of study is the grouping of applications into communicating sets (Jing et al., 2016). These two types of studies don't overlap, to the extent of our knowledge. This paper aims to bridge the gap between the two subdomains in order to give the user a better understanding of the apps installed on their device.

The rest of the study is split into sections as follows. We review works related to static analysis of Android applications and the grouping of them in Section II. In Section III, we discuss the proposed solution, including the comparative study of algorithms and the creation of the mobile application. In Section IV, we present out conclusions and plans for future work.

## 2 RELATED WORK

In this section, we review works that approach the subjects of grouping applications and of malware detection on the Android OS.

### 2.1 Intent Space Analysis

In the article(Jing et al., 2016), authors Jing et al. constructed a holistic reachability graph to represent

intent-based communication. The authors modelled different security filters implemented by the operating system with transfer functions with Intent Space Algebra. The experiments, run on 4 Android based operating systems, indicated how dense the connection between applications using the global clustering coefficient: for Stock Android it was 0.990 including parallel edges. The authors also tested the equivalence between no permissions and no privilege apps, fine-grained domain isolation, enumerating multi-app workflows, and discovery of permission redelegation paths.

## 2.2 Malware Detection

There are a number of papers that approach the topic of malware detection on Android devices. The AndroDialysis(Feizollah et al., 2017) authors analysed the utility of implicit or explicit intents and permissions in malware detection by conducting a comparative study. The authors used a Bayesian Network for the decision maker module and the BeautifulSoup(BeautifulSoup, ) package from Python(python, ) for the extractor sub-module. The experiments run resulted in a true positive rate for intent-based malware detection of 89-91%, varying only by the different algorithms used (K2, Genetic Search, Hill Climber, LAGD Hill Climber).

The AndroPIn(Idrees et al., 2017) method tested various malware detection algorithms using permissions and intents. The dataset used consisted of 1745 applications, with a bias towards malicious apps. The authors do not mention which tools or original methods they used to extract the permissions and intents for each application. The proposed method extracts the input features, checks whether they are in the lists that correspond to malicious intents or permissions, compares the result of the aggregation to a threshold for classification. This approach achieved a true positive rate of 98%.

Shrivastava et al. proposed the SensDroid(Shrivastava and Kumar, 2019) method to classify apps using permissions and intents into different risk categories (low, medium, and high risk). They used a dataset of 8653 applications, with about 65% being malicious. In the study, the authors do not mention what specific tool or method they used for extracting the intents and permissions. The authors defined their own rules based on the frequency of permissions and intents in either malware or innocuous apps. The total accuracy was 98.65%.

The authors of the IPDroid(Khariwal et al., 2020) approached the classification of Android applications in malware or benign by finding the best set of permissions and intents by using Information Gain to rank them. The size of the dataset was of 3131 applications, a split of roughly 45-55% between benign and malicious apps. The features were extracted from the manifest using Apktool(Apktool, ) and Python scripts. The authors tested using only permissions, only intents, or combining both features using 3 different Machine Learning algorithms (Support Vector Machine, Naive Bayes, Random Forest). The best performing was the combined feature collection paired with Random Forest for an accuracy of 94.72%.

A comparison between our method and those of the studies presented in this section will be detailed in subsection 3.4.

# 3 STUDY DESIGN

In this section, the proposed solution will be described, starting with the steps needed to create the Machine Learning model and ending with the edge computing tool that could be used on the Android platform. We divided the work into subtasks that made the overall solution more approachable. This section is split in accordance to those subtasks.

## 3.1 Data Acquisition

In order to create a ML model, we needed a proper dataset on which we could train and test different algorithms. We decided to use the AndroZoo(Allix et al., 2016) dataset that contained Android applications (.apk format) from multiple sources, such as the Google Play Store(Google, ) and the AppChina market(AppChina, ). To use the dataset, we downloaded the corresponding metadata csv file which contained the following fields: `sha256`, `sha1`, `md5` are the values for checksums using different hash functions, `apk_size` which represents the size of the application package, `dex_size` which is the size of the classes.dex file, `dex_date` which is the date attached to the dex file in the archive(zip), `pkg_name` which is the name of the Android package and `vercode` is the version code of it, `vt_detection` and `vt_scan_date` which pertain to the VirusTotal(VirusTotal, ) detection of malware, specifically the number of times it had been detected as malware and the date at which that had happened, and finally `markets` which is a list of the markets on which the apk could be found.

The original csv file from the AndroZoo dataset was over 5.6GB, which was difficult to use and contained many applications which couldn't be downloaded locally due to storage concerns. We decided

to extract a fraction of the file and separate the data into four different files (low risk, medium risk, high risk, and test) for ease of use when downloading the applications and ultimately parsing them. We elected to separate the test data from the training data at this stage, so that it would be untouched by any processing done on the other files. The filtering algorithms copied the header of the main file in order to ensure a consistent structure in all the input files that were created. Following, the decision to classify an app in a certain risk category was done in accordance to SensDroid(Shrivastava and Kumar, 2019). The low risk category had the `vt_detection` score between $[0,3]$, medium risk had the values between $[4,7]$ and the high risk category anything larger than 7. Alongside the requirements for the risk category, the following clauses were added to ensure that the different categories are roughly the same size: the size of the apk must be within a certain range and the application must be available on the Google Play Store. All of these filters were based on metadata of all applications in the dataset that were available in the large csv file. The clauses were decided based on trial and error in order to obtain a balanced dataset where all of the types were represented in roughly the same proportion and the scripts used to process the data were written using the awk(GNU, ) language.

After the extraction of the metadata of the applications into csv files, we began the process of downloading the actual apks. The number of low risk entries was 9731, for medium risk 10136, for high risk 10215, and for the test category 4497 which in total sums up to 34579 applications to be downloaded and from which to extract features. We followed the recommendation of the authors of the AndroZoo and used the az tool(ArtemKushnerov, ) which downloads a number of apks that satisfy specified criteria from the dataset hosted on the AndroZoo server.

The az tool, by default has a mechanism to avoid overwriting a downloaded apk and creates a copy instead. For our use case, this feature was more of a detriment as we did not want duplicates of the same apk. We then decided to modify the tool to better suit our needs: saving each application only once from the input files. In order to optimize the tool, we decided to omit the random selection of the apks in favor of a sequential approach and make sure to save each file uniquely.

The tool attempts to load in memory the original metadata file and then download apks matching the search criteria, which in our case took close to an hour for 100 apks. By extracting separate metadata files for each risk class and modifying the tool's source code, we improved the processing speed to download all the apks from one class (approx. 10000 apks) in one hour.

After downloading all of the applications and saving their metadata, the dataset has been completely created and can be used in the following tasks.

## 3.2 Processing and Interpreting the Data

After obtaining the apks corresponding to the filtered metadata, we transformed them into something that a Machine Learning algorithm could use as input, since apks are large files and are difficult to parse. We created a pipeline for extracting the data and processing it, we used Kotlin(JetBrains, ) for this, so that the code could be easily ported to an Android mobile device. The main obstacle was unpacking the apks and extracting the features, which had to be thought of in a way that could be run on a mobile device, not just on a computer. This rules out apktool(Apktool, ) which could unpack the files, but could not be used on a mobile device.

The file from which we extract features is AndroidManifest.xml, which is in each apk and is stored as bytecode. In order to obtain the file, we created our own algorithm to unpack an apk file, obtain the manifest file, and decode its bytecode so that it may be parsed. Unpacking the apk file is done using the ZipFile class which gets the input stream of any entry in the file. Afterwards we extract the bytecode from the file by using an array of bytes to save all the data inside the file. Decompressing the bytecode and obtaining the actual manifest file is done by parsing the array of bytes. We use a greedy like implementation where we take each byte sequentially, save them in a buffer until the contents of the buffer make sense in the context of a manifest file. We attempt to match tags, the properties set, attributes, resource indexes etc.

Once we had obtained the content of the manifest file, we decided what relevant features can be extracted from it and how to select an output value. We tested multiple iterations of input data, in order to obtain the best possible combination for risk analysis. We started from the baseline of using the official permissions and intent actions from the Android documentation and iteratively added more features over the course of 3 additional tests in order to evaluate if the performance improved. The second test added the amount of custom permissions created and the intent tags that could be defined (data, category, extra), along with the number of times each intent tag appears. The third test added the properties from the application tag which were also taken from the Android documentation. The last test added the differ-

ent hardware or software features that an application could require.

Each feature corresponding to an official permission is extracted as a binary value (0 - the permission is not requested by the application, 1 - the permission is requested in the manifest file). In the case of user-defined permissions, considering that each developer is able to create permissions with custom names, we elected to create one feature containing the number of user-defined permissions requested by the application. The intents are represented as positive values which correspond to the number of times a certain intent property has been defined in the manifest file. The intents are extracted by storing them in a map in which the key is the intent name and the value is the total number of occurrences in the manifest file. The application properties are numerical values, as per the documentation. The corresponding features are either the configured values found in the manifest file or they are the defaults found in the documentation. The device features are also taken from the documentation and are saved with values of 0 (not declared), 1 (declared, but not required), or 2 (declared, and required). The final list of features contains 1152 elements.

The flow of the data acquisition is as follows: an application's manifest file is extracted, the input data (permissions, intents, application properties, device features) is saved using several helper methods, one for each of the different categories of features, all of the elements are concatenated and saved to an input csv file that can be fed into Machine Learning algorithms. After extracting the data, we split it into either a classification or a regression problem, in order to test which method is advantageous in retrieving a correct prediction of security risk.

#### 3.2.1 Classification

For classification algorithms, the output must be of a categorical type, which in this case are the values 0, 1, and 2 representing low, medium, and high risk respectively. When discussing about the input data, we must also take into consideration the fact that some of the applications may have the same set of feature values when parsed. This could potentially lead to instances where two distinct applications A and B, that belong to different risk categories, produce two records with the same feature values. After analysis, we developed three distinct ways to handle such cases: saving all of the applications' features as they are, saving the unique records with the output as the risk category that appeared the most amount of times for that set of feature values (majority case), and saving only unique records and attaching the highest risk category found for any occurrence of that particular set of feature values (dubbed the worst case scenario).

By saving all of the data entries, we have a bloated dataset that may confuse the ML algorithms since the same input data may be attributed to multiple output classes. By saving the data with the class that was found most often to be attributed with it, the outliers may be neglected by the ML algorithm. When saving the input data with the worst case scenario, if there were multiple applications with the same feature values, we save the output class with the highest risk found. In that case, the algorithm would skew the data and decide that more applications are considered malware than not.

The total size of the processed data for all of the data was 28750, for duplicate matching with majority case criteria 4049, and for duplicate matching with the worst case option 4049. We can estimate the amount of applications that use the same configurations, which in turn translates into duplicated records, by comparing the original dataset with the other two, pre-processed ones. The datasets that have been processed are the ones corresponding to the fourth test described previously, in which all the permissions, intents, application properties, and device features are used as features, making it the most diverse from a feature selection point of view. The final dataset composition was balanced from a risk category perspective (low - 9532, medium - 9573, high - 9825).

#### 3.2.2 Regression

In the case of regressors, the output must be a numerical value. For our dataset, the value is the *vt_detection* field from the metadata of an application. As presented earlier, the dataset may contain duplicated records when it comes to the feature values that are extracted. Similar to the classification case, we can process them in two manners: in one scenario, all the input data is saved as is, and in the second scenario, each unique record is paired with the average of the detection rates found for all applications using the corresponding feature values. The potential issue with the first approach is that, despite the size of the dataset, the duplicated records may negatively influence the ML model. The second approach may address this issue however, the averaging function may impede the identification of outliers. The output of the each regressor would be a numerical value like the *vt_detection* score, but it would be interpreted as a risk category based on the rules defined previously (low risk in the interval $[0,3]$, medium in $[4,7]$, and high score larger than 7).

In the case of regression, similar to the classification case, the input file that contains all the applica-

tions is of size 28750 and the one containing the averages of the applications is of size 4049. In both cases, a negligible amount of records from the original set were dropped, as the manifest file was improperly defined and could not be parsed.

### 3.3 Experimental Results

Once all the data had been collected, we ran tests with different Machine Learning algorithms to analyse which is the best to include in the mobile application. The algorithms we tested were a Support Vector Machine(Awad and Khanna, 2015) and multiple ensemble learners: Random Forest(Parmar et al., 2019), Gradient Boosting(Bentéjac et al., 2021), and XGBoosting. In order to implement all the models, we used the Python(python, ) programming language and the scikit-learn(scikit-learn, ) open source library used for machine learning.

When testing the ensemble algorithms, we measured the effect on performance that the number of estimators has. The number of estimators is a hyperparameter used by the Random Forest, Gradient Boosting, and XGBoosting models. We evaluated the performance on the training data and the test data in order to see if the models can generalise to new data or they overfit. We also measured the overall accuracy and the accuracy per the different risk categories.

In order to calculate the accuracy we use the following formula:

$$A = \frac{CP}{TP} \tag{1}$$

where $A$ represents the accuracy, $CP$ corresponds to the number of correct predictions, and $TP$ is the total number of predictions.

All of the following tables of results contain the abbreviations algorithm(Alg.), number of estimators(Est.), Random Forest(RF), Gradient Boosting(GB), Extreme Gradient Boosting(XGB), and Support Vector Machine(SVM).

The general structure of the experiments run is: training the machine learning algorithms selected using the specific input file (all input data, majority case data, worst case data, average case data), and then running the algorithm to predict the values of the test data and calculating the accuracy. Since we ran 4 tests for both classification and regression algorithms with multiple input data files, the entire table of results will not be shown here, only the best representative data is discussed. The complete results of all the tests can be seen in (The authors, ).

After evaluating the classification algorithms using the different tests which represented the inclusion of more features to give a better context to the algorithm, we concluded that the final test was the best performing one. In the chart in Figure 1, we can see the overall accuracies on the test dataset for the following classification algorithms: RF (Random Forest with 250 estimators), GB (Gradient Boosting with 250 estimators), XGB (Extreme Gradient Boosting with 250 estimators), and SVM (Support Vector Machine). In this manner, it is much easier to see the evolution of the algorithms throughout the four tests. It is clear that the RF and SVM algorithms don't perform as well as the algorithms based on gradient boosting. It can be observed that with the addition of more features over the 4 tests, the performance of all models has increased, consolidating our hypothesis that it is possible to improve on the results obtained by using just permissions and intents.

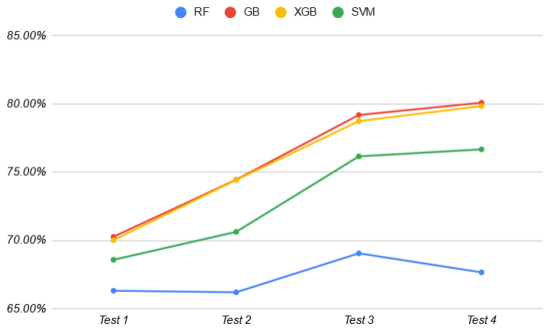


Figure 1: Progression of overall accuracy values of the classification algorithms over the course of tests

Table 1 displays the accuracy data for all the experiments run in the final test. There are several items worth noting, such as the overfitting of the models used with either the majority case or worst case data. When using the majority case data as input for the training, we can see the overfit by comparing any algorithm's accuracy on the training data with the one on the test data. For example, when using a Gradient Boosting algorithm with 200 estimators, the train accuracy was 98.96%, while the on the test data it was 77.78%. One possible explanation this happens is reducing all the applications with the same input features are reduced into a single category based solely on the amount of times it was classified in that category, meaning that it doesn't truly catch the values that don't coincide with the majority case.

Another observation from the Table 1 is the overfit on the worst case dataset, which when using the Gradient Boosting algorithm with 250 estimators, the train accuracy was 99.28%, while the test one was 63.17%. This behaves similarly to the majority case, however the difference is the model doesn't properly classify elements that aren't in the highest risk category per input features.

The final conclusion from Table 1 is that using the complete input data, unprocessed, has the highest accuracy rates, while majority and worst case data tend to have lower accuracy rates. The best algorithm trained during the classification tests is Gradient Boosting with 250 estimators: which garnered the best results for each dataset, but for the input data all the rate was 80.09%.

In the case of regression, the accuracy values on the test

Table 1: Accuracy values in the Classification Test 4

| Alg. | Est. | Test 4 | | | | | |
|---|---|---|---|---|---|---|---|
| | | All data | | Majority Case Data | | Worst Case Data | |
| | | Train Acc (%) | Test Acc (%) | Train Acc (%) | Test Acc (%) | Train Acc (%) | Test Acc (%) |
| RF | 100 | 73.71 | 68.36 | 72.68 | 57.51 | 71.50 | 58.63 |
| | 150 | 74.17 | 69.08 | 72.64 | 64.54 | 71.05 | 58.67 |
| | 200 | 75.09 | 70.29 | 73.06 | 64.27 | 71.23 | 58.22 |
| | 250 | 73.71 | 67.66 | 73.08 | 66.90 | 71.30 | 61.15 |
| GB | 100 | 82.38 | 79.78 | 95.90 | 74.09 | 94.27 | 61.87 |
| | 150 | 82.63 | 80.02 | 97.55 | 74.67 | 95.97 | 61.48 |
| | 200 | 82.75 | 79.98 | 98.96 | 77.78 | 98.30 | 60.85 |
| | 250 | 82.92 | 80.09 | 99.68 | 77.89 | 99.28 | 63.17 |
| XGB | 100 | 82.05 | 79.89 | 93.13 | 73.28 | 92.37 | 61.06 |
| | 150 | 82.32 | 79.75 | 95.18 | 75.87 | 94.94 | 61.06 |
| | 200 | 82.46 | 79.82 | 96.34 | 76.58 | 95.90 | 61.19 |
| | 250 | 82.62 | 79.84 | 97.26 | 76.92 | 96.81 | 61.35 |
| SVM | N/A | 77.01 | 76.67 | 71.35 | 55.71 | 71.35 | 58.79 |

dataset increased similarly to the classification case based on the different sets on input data, thus demonstrating the necessity of the addition of more features.

In the case of the fourth test for regression, the accuracy values are displayed in Table 2. When analysing the results, we can see that by using the average dataset we overfit on the training data, for example, the Extreme Gradient Boosting algorithm with 250 estimators has an 91.48% accuracy rate for the train data, but a 68.07% rate for the test data. This discrepancy is due to the fact that by computing the average value, we create values that may be either on the edge of two categories or the average could be skewed either to the highest risk category or the lowest one.

Another observation is the generally better accuracy rate of the algorithms that were trained on all the data without processing. The best performing algorithm in the fourth test was Extreme Gradient Boosting with 200 estimators with a test accuracy rate of 78.22%. The same algorithm when trained on the set of data that had been averaged resulted in an accuracy value of 68.18%.

After aggregating all the results, we decided that the best algorithm to include in the mobile application was from the classification set, the fourth test, the Gradient Boosting algorithm with 250 estimators. The choice was between the two best performing algorithms from the respective categories: regression or classification. The regression Extreme Gradient Boosting algorithm from the fourth test with 200 estimators was the best performing from that set. When comparing the accuracy rate, the classification algorithm had an accuracy rate of 80.09%, while the regressor had an accuracy rate of 78.22%. Besides using accuracy as a measure for usability, we also decided to use an algorithm that would be small enough to be ported to a mobile application and could return results quick enough so that the user wouldn't wait a long amount of time. This immediately excluded Support Vector Machines from the available options, since it took very long (upwards of 30 minutes compared to other methods which took about 15 minutes) to compute the values even with a small input.

Among the tested algorithms, Random Forest was the least accurate ensemble learner, regardless of estimator count. Gradient Boosting algorithms achieved the highest accuracy with more estimators for classification. Support Vector Machine performed poorly and required the most time for training and testing.

Table 2: Overall accuracy values for Regression Test 4

| Alg. | Est. | Test 4 | | | |
|---|---|---|---|---|---|
| | | All Data | | Average Data | |
| | | Train Acc (%) | Test Acc (%) | Train Acc (%) | Test Acc (%) |
| RF | 100 | 76.25 | 74.00 | 73.60 | 56.65 |
| | 150 | 76.26 | 73.89 | 73.60 | 56.67 |
| | 200 | 76.35 | 74.18 | 73.57 | 56.54 |
| | 250 | 76.49 | 74.38 | 73.94 | 56.79 |
| GB | 100 | 79.57 | 76.47 | 83.53 | 59.64 |
| | 150 | 80.05 | 76.90 | 86.89 | 61.21 |
| | 200 | 80.41 | 77.39 | 87.68 | 62.25 |
| | 250 | 80.44 | 77.69 | 88.69 | 65.66 |
| XGB | 100 | 80.31 | 77.48 | 88.42 | 66.76 |
| | 150 | 80.89 | 77.91 | 90.29 | 67.64 |
| | 200 | 81.07 | 78.22 | 90.71 | 68.18 |
| | 250 | 81.13 | 78.09 | 91.48 | 68.07 |
| SVM | N/A | 74.00 | 73.80 | 69.55 | 61.08 |

Gradient Boosting algorithms provided balanced accuracy across categories, unlike Random Forest and Support Vector Machines, which favored the low-risk category while underperforming in others. Although these latter algorithms might excel in binary classification, they were suboptimal for this task. Gradient Boosting and Extreme Gradient Boosting achieved better balance and higher overall accuracy, as shown in Tables 1 and 2.

Table 3: Comparison of results with other works in the field

| Solution | Classification Type | Dataset | Best Results |
|---|---|---|---|
| AndroDialysis (Feizollah et al., 2017) | binary | 7406 apps (1846 benign, 5560 malware) | 95.5% |
| AndroPIn (Idrees et al., 2017) | binary | 1745 apps (445 benign, 1300 malware) | 98% |
| SensDroid (Shrivastava and Kumar, 2019) | risk categories | 8653 apps (2972 low, 2556 medium, 3125 high) | 98.65% |
| IPDroid (Khariwal et al., 2020) | binary | 3128 apps (1414 benign, 1714 malware) | 94.74% |
| Our Solution | risk categories | 28750 apps (9352 low, 9573 medium, 9825 high) | 80.09% |

## 3.4 Discussion of Results

As seen in Table 3, the total sizes of the datasets did not exceed 10000 applications, in some cases they were below 5000. As the datasets were much smaller than our own, the other methods may not generalize the data as well as ours does. Another aspect worth mentioning is that in some papers(Feizollah et al., 2017), (Idrees et al., 2017) the datasets used were skewed with a bias towards malware, which could lead to a higher rate of false positives. In the case of SensDroid(Shrivastava and Kumar, 2019), they used a statistical approach that did not integrate Machine Learning, instead opting to create a set of rules based on their own classification of permissions and intents as malicious or clean. Our method used Machine Learning algorithms so that we could create a more robust and adaptable solution as we would not need to create or modify several rules when the underlying data changes. As our approach classified applications into risk categories, rather than a binary classification, and used a very large and balanced dataset for training purposes, the lower accuracy rate is justified.

## 3.5 Final Tool and Grouping Algorithm

The tool we developed did the following tasks: extract all the installed packages, parse the manifest file, group applications by the queries tag in the manifest, extract input features from the manifest that are used with the ML model. By grouping the applications based on the queries tag, we collected all applications that have access to each others' different actions, therefore they could be used in a malicious manner.

The grouping algorithm, written in Kotlin, uses the PackageManager class to extract installed applications and their metadata. After extracting the applications, we parsed the list, identified which ones queried others, and created an input array from the manifest file to group applications based on their interactions.

In order to do this, we created the function extractQueriedPackages that receives as input the PackageInfo of the current application, an XmlPullParser for the parsing of the manifest file, and a string in which the contents of the manifest file are stored. We then found all the $< queries >$ tags from the manifest and added the name of the package that is being queried to a list for the current application. The list of package names is then returned for further use.

After saving all the packages each application queried, we began properly grouping the applications by applying a closure algorithm on the sets of queried applications. The algorithm can be split into the following steps:

1. Initializing a final set of groups as an empty set, and a previously computed set of groups as an empty set and acquiring the map of packages and their set of queried packages;
2. Checking whether changes have occurred or if this is the first iteration of the computation of the set of groups, if the check doesn't pass, the flow goes to step 8;
3. Setting the flag for changes to false;
4. Going through the map of queries, adding the first entry if the current set of groups is empty and starting this step again, if not the flow continues to step 5;
5. Checking whether the current application is contained in the set of queried packages of a different application and then adding the application to the set along with all the packages it queries and marking the changes flag as true;
6. In the alternative case, the applications from the set of queried packages are checked and added in a similar method as in step 5;
7. After parsing all of the queries in the map, the current set of groups is compared to the previously computed one, if they are not the same, the algorithm returns to step 2, otherwise it continues;
8. The current set of groups is returned.

## 3.6 Edge Computing Risk Assessment

After calling the function that computes the groups of applications using the previously stated algorithm, we continue with the gathering of the input data for the ML model.

The input data was extracted using the same method as the training pipeline and was aggregated for parsing the groups. For standard permissions, we applied a bitwise OR to determine if a permission existed in any group application. Custom permission values were summed across applications to reflect the total for the group. Intent values were similarly summed, while application properties used a bitwise OR to indicate presence rather than frequency. For device features, we selected the maximum value, reflecting the highest access level within the group.

By integrating the best-performing machine learning algorithm, we developed an edge computing tool that processes data locally, without server interaction or network traffic, ensuring complete offline functionality. This is ideal for monitoring suspicious apps, as it allows offline analysis and prevents further data transmission. Additionally, we added a feature to scan only non-system apps by filtering out package names containing *com.google* or *com.android*, aligning with Play Store Developer Program Policy restrictions.

# 4 CONCLUSION

In conclusion, we proposed a solution that aims to give an Android device user a better knowledge of the security risks different applications or groups of applications pose. Since there are many malware applications publicly available on the Google Play Store(Google, ), the necessity for such a solution to raise awareness is clear.

We developed a pipeline to process over 30,000 applications by unpacking APK files and extracting their AndroidManifest.xml files. From the manifest, we gathered features, specifically: permissions, intents, application properties, and device features. To refine feature selection, we created four separate datasets: in the case of classification, we further interpreted the sets as unprocessed data, majority-case data (most frequent risk category for duplicates), and worst-case data (highest risk category for duplicates); and for regression, we interpreted them as unprocessed and averaged data (mean risk for applications with identical feature values).

We evaluated several machine learning algorithms for security risk analysis, including Random Forest, Gradient Boosting, Extreme Gradient Boosting, and Support Vector Machine, testing ensemble learners with 100–250 estimators. The fourth dataset, containing all features, yielded the best results for classification and regression. Gradient Boosting with 250 estimators performed best for classification, while Extreme Gradient Boosting with 200 estimators excelled in regression. Since the classifier outperformed the regressor (80.09% vs. 78.22% accuracy), we selected it for the tool.

We developed a method to group applications by connections using the manifest's queries tag and integrated it into the mobile tool alongside the feature extraction and selected model. With the chosen model, the tool operates entirely on-device, aligning with edge computing principles. This ensures all computation and data storage remain local, eliminating the need for server access or internet connectivity, thus preventing third-party interception.

Our future plans include adding Principal Component Analysis to explain why an application/group of applications was classified into a specific risk category. We aim to improve accuracy by identifying features most indicative of risk and incorporating implicit intents through Java bytecode analysis.

# REFERENCES


Allix, K., Bissyandé, T. F., Klein, J., and Le Traon, Y. (2016). Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th of MSR '16*, pages 468–471. ACM.

Apktool. Apktool. https://apktool.org/. Online; accessed 4 March 2024.

AppChina. AppChina.com. http://www.appchina.com/. Online; accessed 9 April 2024.

ArtemKushnerov. az. https://github.com/ArtemKushnerov/az. Online; accessed 1 April 2024.

Awad, M. and Khanna, R. (2015). *Support Vector Machines for Classification*, pages 39–66. Apress.

BeautifulSoup. Beautiful Soup Documentation. https://beautiful-soup-4.readthedocs.io/en/latest/. Online; accessed 4 March 2024.

Bentéjac, C., Csörgő, A., and Martínez-Muñoz, G. (2021). A comparative analysis of gradient boosting algorithms. *Artificial Intelligence Review*, 54:1937–1967.

Feizollah, A., Anuar, N. B., Salleh, R., Suarez-Tangil, G., and Furnell, S. (2017). Androdialysis: Analysis of android intent effectiveness in malware detection. *Computers & Security*, 65:121–134.

GNU. The GNU Awk User's Guide. https://www.gnu.org/software/gawk/manual/gawk.html. Online; accessed 1 April 2024.

Google. Google Play. https://play.google.com. Online; accessed 16 March 2024.

Google Corporation. Android Open Source Project. https://source.android.com/. Online; accessed 27 February 2024.

Holbrook, L. and Alamaniotis, M. (2019). Internet of things security analytics and solutions with deep learning. In *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 178–185.

Idrees, F., Rajarajan, M., Chen, T. M., Rahulamathavan, Y., and Naureen, A. (2017). Andropin: Correlating android permissions and intents for malware detection. In *2017 8th IEEE Annual IEMCON*, pages 394–399.

JetBrains. Kotlin Programming Language. https://kotlinlang.org/. Online; accessed 24 March 2024.

Jing, Y., Ahn, G.-J., Doupé, A., and Yi, J. H. (2016). Checking intent-based communication in android with intent space analysis. In *Proceedings of the 11th ACM on Asia CCS'16*, page 735–746.

Khariwal, K., Singh, J., and Arora, A. (2020). Ipdroid: Android malware detection using intents and permissions. In *2020 Fourth WorldS4*, pages 197–202.

Parmar, A., Katariya, R., and Patel, V. (2019). A review on random forest: An ensemble classifier. In *ICICI 2018*, pages 758–763. Springer.

python. Welcome to Python.org. https://www.python.org/. Online; accessed 8 April 2024.

scikit-learn. scikit-learn: machine learning in Python. https://scikit-learn.org/stable/index.html. Online; accessed 8 April 2024.

Shrivastava, G. and Kumar, P. (2019). Sensdroid: Analysis for malicious activity risk of android application. *Multimedia Tools and Applications*, 78(24):35713–35731.

Statcounter GlobalStats. Mobile Operating System Market Share Worldwide. https://gs.statcounter.com/os-market-share/mobile/worldwide. Online; accessed 2 March 2024.

The authors. Comparative Analysis for Android Risk Analysis. https://tinyurl.com/comparativeandroid. Online; accessed 13 July 2024.

VirusTotal. VirusTotal - Home. https://www.virustotal.com. Online; accessed 8 March 2024.