Generating Android Malware Profiling Rules From Simple to Extract Metadata

1st Fabian Haugen
IDA
Linköping University
Linköping, Sweden
fabha972@student.liu.se

2nd André Willquist
IDA
Linköping University
Linköping, Sweden
andwi954@student.liu.se

***Abstract*—The aim of this research is to derive rules for profiling malware in Android applications, using data that could be easily extracted from the hosting marketplaces and data intrinsic to the downloaded APKs themselves. An automatic system for scraping metadata from the marketplace website, extracting intrinsic metadata and submitting APKs to Virustotal is set up. Based on the dataset generated from this system, analysis is done using decision trees and association analysis. Because of the discovery that the decision trees created were heavily biased towards rules for profiling benign APKs the strategy of the research was changed for the association analysis, modifying the dataset in such a way as to increase the likelihood of extracting rules which profile malware.**

From both of these analyses a sets of rules were found. The rules were in general rather intuitive and reasonable, with a few outliers which were harder to interpret. A discussion of these rules were performed where their meanings were analysed.

***Index Terms*—Malware Profiling, Android Security, Rule Extraction, Machine Learning**

I. Introduction

Given the prevalence of malware in mobile applications shared on the web, Android as a platform should in no way be excluded from modern security considerations. Among applications offered on various marketplaces targeting Android users, many are disguised as benign while in reality exhibiting malicious behaviour when being run. As of 2018, 26.6 million applications were detected to contain malware [1]. This is however still just a small part of all of the android apps available for download. Thus when considering all applications currently in circulation, how does one determine which can be trusted and which to avoid? Scanning every application for malware with well known malware detection tools may be a good idea, but might not be feasible with the amount of APKs available. The aim of this research is to generate rules from easy to extract metadata available on an Android marketplace, to aid security analysts in profiling malicious applications. This task motivates the following question formulation:

- What rules are likely to be most effective for deciding whether a given app is malicious?

II. Background

Terms in need of explanation will be defined here along with a presentation of previous research lightly illustrating concepts discussed in this report, mostly relating to machine learning, data mining and Android security.

A. APK

An Android package, or APK, is a file compressed with the format used to distribute applications on the Android platform. APKs contain everything that is needed for an app to be run on an android phone. The relevant files for feature extraction are as follows:

- **The Android Manifest** defines a lot of the aspects which make up the metadata of the app, such as the name, the components of the app, the permissions it requires to be run and the hardware and software features it requires. [2]
- **The .dex Files** contain the application's program code.
- **CERT.RSA** contains certificate information of the application issuer.
- **The assets/ and res/ Directories** contain data used by the application, such as images or text data.

B. Description of Relevant Android Permissions

- **BROADCAST_STICKY** - Allows an application to broadcast sticky intents. These are broadcasts whose data is held by the system after being finished, so that clients can quickly retrieve that data without having to wait for the next broadcast. Messages are sent between components of an application. Protection level: normal
- **BLUETOOTH_ADMIN** - Allows applications to discover and pair Bluetooth devices. Protection level: normal
- **BLUETOOTH_ADMIN** - Allows applications to connect to paired Bluetooth devices. Protection level: normal
- **CALL_PHONE** - Allows an application to initiate a phone call without going through the Dialer user interface for the user to confirm the call. Protection level: dangerous
- **CHANGE_NETWORK_STATE** - Allows applications to change network connectivity state. Protection level: normal
- **INTERNET** - Allows applications to open network sockets. Protection level: normal
- **MODIFY_AUDIO_SETTINGS** - Allows an application to modify global audio settings. Protection level: normal

- **READ_PHONE_STATE** - Allows read only access to phone state, including the phone number of the device, current cellular network information, the status of any ongoing calls, and a list of any PhoneAccounts registered on the device. Protection level: dangerous
- **RECEIVE_SMS** - Allows an application to receive SMS messages. Protection level: dangerous
- **READ_SMS** - Allows an application to read SMS messages. Protection level: dangerous
- **READ_EXTERNAL_STORAGE** - Allows an application to read from external storage. Protection level: none
- **WRITE_EXTERNAL_STORAGE** - Allows an application to write to external storage. Protection level: none
- **PROCESS_OUTGOING_CALLS** - Allows an application to see the number being dialed during an outgoing call with the option to redirect the call to a different number or abort the call altogether. Protection level: dangerous

C. Tools:

- **Virustotal** is a website providing services for scanning files for malware with several different engines such as Avast and F-secure. Virustotal also provides an API for automating the scanning of files via scripting. The Virustotal API can be accessed for free, but with some restrictions, which is in part enforced by requiring users to create an account on the Virustotal website. [3]
- **NinjaDroid** is a tool which automatically extracts most of the existing information from a given APK, such as file sizes, name, version, requirements, et cetera [4]
- **Weka** is an application which contains several tools for machine learning and data mining. It is GUI based and allows the user to apply many different algorithms on provided data. One of the tools is specialized in performing association analysis and allows the user to perform such analysis with minimal knowledge of the underlying functionality and algorithms.
- **Selenium** is a tool for testing web applications. It uses a browser process, a web driver, to interact with a webpage using given commands. In this case, google chrome was used together with selenium's python implementation.

D. Classification

Classification of data in a machine learning sense means that, given measurements of different attributes of the sample that is to be classified, the model will try to find patterns so that it can predict a class of the object based on the attributes. An example of this would be that given the amount of downloads for an app, the model will try to predict whether the app is malware or not.

E. Selected machine learning algorithms

While technologies such as Neural Networks (often Deep Neural Networks also known as Deep Learning) and Support Vector Machines (with non-linear kernels) are usually more in the public eye as to regarding machine learning nowadays these techniques do have some weaknesses. One of these is the reliance on transforming the data into a feature space to help ease the classification process and to enable non-linear classification. Being able to perform non-linear classification is essential to being able to handle complex data, and thus the conversions to feature space are essential for these techniques to function properly. These feature spaces do however come with some drawbacks, one of the largest of these being a lack of interpretability. It can be extremely hard to tell exactly which properties of the input data that has affected the outcome. Since it is hard to tell which inputs lead to which output it is also hard to create rules connecting input to output which humans can understand based on the trained models.

Because of these problems in order to extract the rules based on metadata for deciding whether an app is malware or not this report resorts to other machine learning approaches which can provide these rules. These approaches will be introduced in the following sections.

1) Decision trees: If you were to imagine the simplest classifier possible it would probably be to classify based on one value in one dimension, for example if the amount of downloads are less than 10,000 then the app is assumed to be malware, otherwise it is assumed to be safe. Such a classifier is called a decision stump. While decision stumps might work on the simplest of data it would obviously not be that good as a classifier for more complicated data, even such a simple relation as y=x could not be modelled. However, if we combine several of these decision stumps, looking at different dimensions and having different thresholds on the values a much better classifier could be created. This is what is called a decision tree and what will be used as a classifier in this study.

The biggest advantage of decision trees compared to other relevant machine learning approaches is that the classification rules are immediately extractable from the trained model.

2) Association Analysis Rules: Association analysis is the theory of finding rules for the different classes in the already existing data, and is thus extremely relevant for this report. The rules are found by looking at the dataset and looking at which patterns are often present in the interesting samples. Such a pattern could be a correlation between malware classification and what average review score a given app has. Since it would be hard and take way to long time to generate all rules, there are algorithms limiting the generation of rules to the ones which are considered interesting, based on the rule's support and confidence.

The support of a rule is how often it occurs in the database, for example if low review score and malware occurs together in 1% of the entries in the data then the support for this combination of attributes would be 1%.

The confidence of a rule is what fraction of cases where the left hand side of the rule (the consequent) of the rule occurs that the right hand side (the antecedent) also occurs. For example if in 70% of the cases where we have classified an app as malware we also find that the app has less than 10,000 downloads then the confidence of the rule is 70%.

The algorithms for rule generation does require that the data is discrete, and thus require the data to be discretized before the algorithm can be run, meaning that the parameters selected can severely affect the outcome of the algorithm.

F. Malware Classification on Application Metadata

In [5] it has been shown that it is possible to classify android APKs as either malware or benign based on easily extracted metadata features, such as the number of required permissions and the time since the latest update. This does lend credence to the idea of finding rules for classification of malware based on such easily extracted metadata features. It has also been shown that certain categories of apps are more likely to contain malware, once again supporting the hypothesis that such a classification would be possible.

G. Motivation of Collected Features

In this section initial motivation for the choice of attributes to analyze is presented arguing for why they are considered important in the context of this report.

The expectations when setting out with the project is that:

- There will be a difference in malware density among the different categories, as shown in previous studies. [5]
- Apps with more permission requirements are more likely malware since the goal of the malware is likely to gain access to as many of the phones functionalities as possible.
- Apps with lower user review score are more likely malware since if users discover that malware is present, then they are likely to give the app a lower review score.
- Apps with a higher version number is probably less likely to be malware, this since it is unlikely that malware will get continuous support and updates after release.
- The proportion of the amount of URL:s and the size of the app might be indicative of malware. While it is likely that larger apps have more URL:s it is possible that malware have a higher number of URL:s compared to their size.

III. Methods

In this section the general workflow of the project is presented. This includes the fetching of apps, labeling of the apps (as malware or benign) as well as rule extraction from the collected data.

A. Choosing Markets

To select a suitable marketplace for downloading apps a few different criteria were taken into account. The most important were:

- The amount of free apps should be larger than 10,000
- The likelihood of viruses to be present on the site should not be too low

The chosen marketplace was Anzhi.com, on which both stated criteria were fulfilled [6]. Furthermore the site had relevant metadata readily available on every app page and had a structure that could be traversed without too much trouble (for more information on structure and traversal see section III-D).

B. Challenges and Rationale of the Program Structure

A challenge with this project is the amount of data which needs to be downloaded and processed. Assuming that the maximum size of each app is 30MB, which is the maximum which we can submit to Virustotal, and that the goal of the report is to download and process about 10,000 apps this would mean that in the worst case an approximate 300GB of data would be required to be downloaded and stored. It would therefore be preferable to instead manage the downloading and processing of apps in smaller batches. This would allow for the apps within a batch to be removed when processed so that only the unprocessed apps need to be stored on the computer, greatly reducing the required space. This does however increase the complexity of the programming task.

Another challenge is the amount of time it takes to process all of the apps. Since Virustotal restricts the allowed amount of API calls to 4 per key per minute this means that the expected time to process 10,000 apps would range from two to three days. This is quite a long time to expect a single process to manage to run uninterrupted, especially when the process requires access to the internet to function. This means that it would be preferable to take care to save state as often as possible so that the process can be paused at any point, or even stop it entirely, without having to restart the downloading and processing process. This becomes harder when the downloaded apps are removed after they have been processed.

C. Structure of the Data Collection Code and Execution

The data collection code is made up of four different main components, these are:

- A module for downloading APKs and scraping metadata
- A module for submitting APKs to Virustotal for scanning
- A module for extracting the intrinsic metadata from the APKs
- A module for removing the already processed apps

These four components communicate with each other by using files which they write to and read from. For example when the downloading module has downloaded an APK it writes this to two different files to notify the submitting module and the extracting module that the APK is available for processing.

The entire system is also built up in a cyclical manner where the processes do all of the work available to them in one cycle before restarting to check if any new work is available. For example the submitter submits all of the APKs available to it in one cycle to then wait until the next cycle starts before checking if any new apps are available. This is done so that the cleaner has some time where all processes have stopped during which it can remove APKs safely.

The system was parallelized between 6 computers, all running on a given index.

D. Traversing the Website

For the site traversal the choice was made to fetch the app pages from an index page found on the site. All the apps

considered for download could be reached by incrementing a number in the URL of this page and following the app links listed there. The computers running the program were given indices spread out over the whole set.

E. Downloading the Metadata

Application metadata was obtained by loading the app-specific page on the marketplace and then scraping the relevant information from it (using the python package BeautifulSoup), which is then saved to disk in a CSV file for future analysis. In the case that one of the apps were missing some of the expected metadata a default value was instead used.

The chosen data was:

- The app name
- The review score
- The app category
- The number of comments

F. Downloading the APKs

The APKs were downloaded by clicking a download link on their respective app pages. Since the apps could not simply be downloaded by fetching from a specific URL, but instead by executing some JavaScript code, a web driver was needed to initiate downloads. The python package selenium was used with a chrome driver to click on the download buttons and run downloads in the browser. Since the filenames where randomly generated by chrome, the file creation time was used to associate files with fetched data, which meant that the downloading process needed to be serial.

G. Extracting the Intrinsic Metadata

To extract intrinsic metadata from the APKs, the APK parser tool NinjaDroid was used. The apps marked ready for parsing was input, with JSON formatted data being output from NinjaDroid. From these the relevant data was extracted and saved in a CSV file for future analysis. The chosen data was:

- The version number
- The app size
- The number of URLs in the APK (extracted from the .dex files)
- The number of shell commands used (extracted from the .dex files)
- The permissions required

H. Submitting the APKs to Virustotal

The submitter sends APKs to Virustotal for malware scanning. It is made up of one main thread which loads in all of the apps and makes sure that all other subprocesses finish properly as well as one new thread for each of the APKs which are to be processed.

As stated for each of the APKs which are to be submitted there is one new thread spawned which is responsible for taking care of the entire process of scanning that APK. This process is made up of several different steps which also vary depending on the responses received from Virustotal. The first thing which is done for each of the APKs is to check if the APK is already present on Virustotal. Is this the case then the next step is to check if the current report on Virustotal is recent enough to use, in this case recent enough is defined as made within the last 180 days.

Is the report not present on Virustotal or if the existing report is too old then the APK is submitted to Virustotal for scanning. After an APK has been submitted to Virustotal the thread responsible for the APK sleeps for five minutes waiting for the result. When the thread wakes up it checks if the results are available or not, if they are then the thread is done and outputs the results, if the results are not available, then the thread sleeps for another three minutes.

The reason for the threaded module is to enable the possibility of having more than one app being processed at a time, this means that if the Virustotal submitter has to wait for results for one of the apps then it can still process the others, making sure to fully utilize the four available requests per minute.

The restriction of four requests per minute gets somewhat trickier when having a thread per APK however, since all of the living threads must share the available requests. This is however accomplished by python's condition functionality. The condition functionality allows having a restriction where the threads can only make a request if a counter is above a threshold value, this counter is protected by a lock. A separate thread can then update the counter with four new requests every minute as well as updating all of the waiting threads when the counter is updated.

I. Extracting Malware Recognition Rules

For the rule extraction the R programming language is used. The data is loaded in and combined into one large data frame. After the data has been loaded it is then used to create a first classification tree. After this, cross validation is used to decide upon the optimal tree depth and the tree is pruned to this level iteratively.

For the association analysis Weka is used to generate the rules. To be possible to be used in weka the data needs to be converted to an .arff format, this can, however, easily be done in weka. After converting the data the association is performed by selecting the association option from the interface. From this the algorithm is run and the rules extracted. Because of how the association analysis works, not all APKs can be used for this. If all of the APKs are included then the benign APKs far outnumber the malicious APKs, and thus the rules found relate only to the benign APKs. Because of this the data was pre processed in such a way that there were approximately the same amount of benign APKs as there was malware. This was done by first dividing the data up into malware and benign APKs. After the APKs were divided up approximately the same amount of benign APKs were taken at random from the set of benign APKs as there were malware APKs. These two equal sized sets were then combined into one final set which was then processed. The reason it is required to have benign APKs in the set, which is processed with the association analysis, was that otherwise there occurs problems with the

confidence of the rules, since there exists no benign APKs for which the rules could also hold.

When doing the intrinsic metadata extraction such as it was done in this project, each of the possible permissions of an app is represented as one binary variable in the data regarding the APK. This means that each app was represented by 170 different variables in total. Because of both the time and the memory complexity of the Apriori algorithm, as a function of the amount of variables in the data, it becomes unfeasible to perform the algorithm on the data when keeping all of the variables. This means that some of the variables has to be removed. This removal of variables was done manually and worked in such a way that if there were a variable where less than 100 APKs fell into either of the categories then the variable was removed. This means that if, for example, less than 100 out of the APKs used a permission then the variable representing that permission was removed before the algorithm was run. This also means that all of the variables that were removed were those with the lowest amount of variance.

IV. Results

In this section the generated decision tree and rules are presented to the reader.

A. Model and Extracted Rules

In the following section the results regarding both the generated tree classifier and the generated rules are presented. Regarding the decision trees only the final tree for the finally selected virus threshold is presented, there were however trees generated for each of the integer thresholds in the range [1,20], this was done to be able to compare the misclassification rates of the trees.

B. Selection of Threshold

As earlier presented one challenge which presented itself in this project was to decide upon a threshold as to how many of the Virustotal engines had to mark an APK as a virus for it to be considered a virus. To help with this decision two figures were generated, one presenting how the proportion of viruses compared to overall APKs changed for different values of the threshold(Figure 1), the other was an image showing the misclassification rate of the decision tree as a function of the selected threshold(Figure 2).

Based on these figures the values of the threshold for the decision trees was set at 13. The threshold for the association rules was also set at 13, which is the value for which about 10% of the data is virus. The reason that 20 activations were not selected, even though it apparently has a lower misclassification rate is that when the threshold is increased the density of malware in the data is decreased. This decrease of malware density results in a better misclassification rate, even if the model is not better at profiling viruses, just because there are fewer to profile.

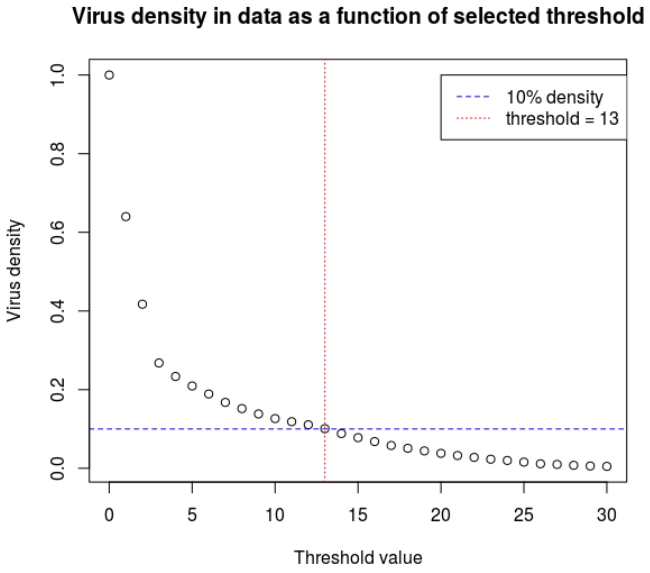

Fig. 1. Virus density as function of threshold

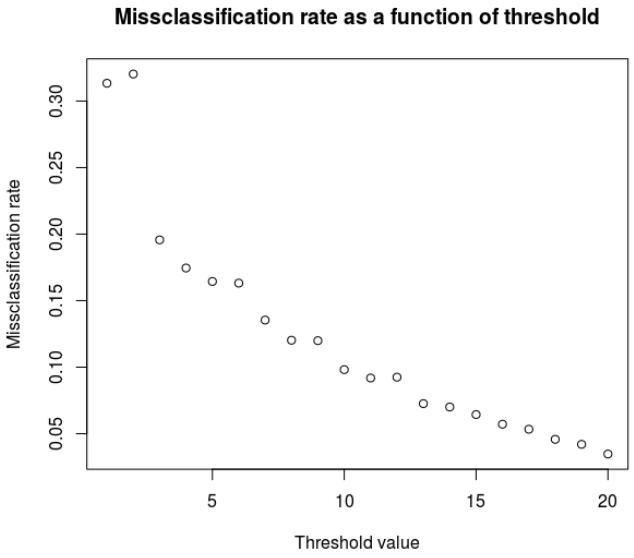

Fig. 2. Misclassification rate as related to threshold

C. Generated Tree Models

Based on the selected threshold of 13 engine activations for the APK to be classified as a virus the following tree model was generated. The actual tree model is presented in Appendix A, since it was too large to include in this text. Because of problems with the graphical representation the tree model is included in text form.

D. Rules Extracted via Association Analysis

When the rules were generated two sets of rules were generated, one with rules only related to the classification of viruses and one where all rules were included. Here only the virus classification rules are shown.

Only two of the rules generated are actually represented since the length of the rules otherwise quickly make the report rather unreadable. The two that were selected were two out of the top four. The reason that not just the top four were selected is that the rules seem to come in pairs, where every second rule is just a slight modification of the first rule. In all of the cases it is also such that both the right and the left hand side include exactly the same amount of samples in both of the rules, meaning that both of the rules relate to exactly the same sample of apps. Because of this it seemed more relevant to include rules which differed more, even if they were not specified as the rules with the highest confidence.

It should be noted that even with this exclusion the rules which are included are still quite similar, but have neither side covering the exact same amount of apps, and also have differing confidence. More differing rules could likely be found if more rules were to be generated, doing this is only prevented by a lack of time. The rules could easily be generated by the authors upon request.

Any further generated rules would have a lower confidence than the ones presented in this report.

The rules are in format

$$A, B, C, relevant for D samples \rightarrow E, F conf(G)$$

To read this format it is important to know:

- A, B, C are the attributes that all apps in the rule have
- D is the amount of samples which show all of these attributes
- E is the result of the rule (in this report that the samples are virus)
- F is how many of the D samples which are actually virus
- G is the confidence of the rule based upon D and F

It is also the case that if the rules say A=0 or A=1 this means that A is either false or true respectively for the samples. For example if A is a permission this means that the samples either have or do not have the permission. On the other hand if A is the status of virus then it means that the samples either are not viruses or are viruses.

1) BLUETOOTH=0,
   BROADCAST_STICKY=0,
   CALL_PHONE=0,
   CHANGE_NETWORK_STATE=0,
   INTERNET=1,
   MODIFY_AUDIO_SETTINGS=0,
   READ_PHONE_STATE=1,
   RECEIVE_SMS=0,
   relevant for 1105 samples
   $\rightarrow$ virus=1, 791 conf:(0.72)

2) BLUETOOTH=0,
   BROADCAST_STICKY=0,
   CHANGE_NETWORK_STATE=0,
   INTERNET=1,
   MODIFY_AUDIO_SETTINGS=0,
   READ_EXTERNAL_STORAGE=0,
   READ_PHONE_STATE=1,
   READ_SMS=0,
   WRITE_EXTERNAL_STORAGE=1,
   relevant for 1098 samples
   $\rightarrow$ virus=1, 785 conf:(0.71)

V. Discussion

In this section the results are discussed, were they similar to the expected results? Do they have good support and confidence? Do they make sense?

A. Rule Analysis - Tree Algorithm

While analysing the resulting tree, it should be noted that the feature on the level closest to the root is the most important, in this case the READ_PHONE_STATE permission. This being the most important feature also seems rather logical. Data about the phone state would be useful in developing effective malware, which is further backed by the fact that the permission has the "dangerous" protection level. Not having the READ_PHONE_STATE permission reduces the amount of malware in the population from about 10% to about 2.7% which is a rather significant shift, almost a factor 4.

Attributes on the second level are also rather important, which would include the update_date and the amount of shell commands. The role of these attributes is however not possible to determine from the tree. This is partly due to the fact that the rules represented by these nodes are an intersection of several different constraints, and thus not only dependent on these attributes. Another reason as to why it is not possible to extract any immediate meaning as to why the attributes are important is that the intermediate nodes do not represent any complete rules, neither do they represent any finished profiling of the apps as malware or benign. That said there are some interesting complete rules ending in leaf nodes which can be discussed. The rules which are most interesting to discuss are the ones that profile malware with a reasonable certainty, as well as those that profile benign apps with an extraordinary certainty, these are the ones that will be presented.

First amongst the rules is the rule ending in node 4), profiling benign apps with a 99.8% certainty. This rule holds for apps without the READ_PHONE_STATE permission which had their latest update more than about 2.5 years ago. This is states that apps which are somewhat older almost entirely needed the READ_PHONE_STATE permission in able to function as malware, increasing the likelihood that the READ_PHONE_STATE permission at least used to be an extremely important indicator of malware.

The next rule, the one ending in node 21, does however present a somewhat interesting counterpoint to this, it states that apps that do not have the READ_PHONE_STATE permission, but that are updated more recently than about 2.5 years ago, which also have less than 13.5 shell_commands and are larger than 6MB, have a 70% probability of being malicious. This seems to indicate that some way of performing malicious activities without the READ_PHONE_STATE permission has been introduced recently. It also seems likely that the code required to perform this malicious activity is likely rather large, leading to the larger size requirement for the rule. Based on the previous assumptions that malware in general have a higher amount of shell commands the final requirement does however seem somewhat unintuitive. Overall however this rule seems quite intuitive based on these explanations. This rule does raise some questions as to what allowed the malware to bypass the requirement of having the READ_PHONE_STATE permission however. A potential explanation for this could be the emergence of lower level exploits, such as Meltdown and Spectre [7] which could allow applications to bypass the protections of the operating system, even without special permissions.

The third interesting rule is the rule ending with node 11. This is one of the most unintuitive rules in the generated tree. It states that if the app does not have the READ_PHONE_STATE permission, it updated more recently than 2.5 years ago, does also have more than 13.5 shell commands, then it is with 99.7% certainty a benign app. This rule is not nearly as intuitive as most other rules, but there are two possibilities that can be seen as to why this would be possible. The first of these possibilities is once again that some important feature is missing or some unsuitable simplification has been made in the model. The other option that can be seen is not that there is something special with the benign apps having more than just 13 shell commands, but rather that the malware presented in the previous rule, at node 11, have extraordinary few shell commands. If we are to assume this then it might be so that if the malware profiled in rule 11 use some exploit to bypass the restrictions of the operating system (thereby allowing for direct execution of instructions), then the shell commands might be less necessary than for any usual app. This combined with the assumption that malware applications likely have less effort put into them, thus requiring a less than average amount of shell commands than normal to perform their other functionality, then this might result in this specific class of malware having less than the average amount of shell commands overall. This would somewhat explain the shell command part of the rules, both for this rule and for the previous rule, ending in node 11.

The final rule, ending in node 222), is rather long. It states that, if the app uses the READ_PHONE_STATE permission, has more than 30 shell commands, has more than 120 URLs, has the ACCESS_COARSE_LOCATION permission and has a version number between 64 and 115, then it is with 84% certainty a virus. This rule has many parts to go through. Starting with the READ_PHONE_STATE permission it has earlier been stated that this is a likely predictor of malware, meaning that it is not a surprise to see it included in the rule. Along with this the ACCESS_COARSE_LOCATION permission is also included, this allows the app to access the phones approximate location. Just as the READ_PHONE_STATE this is a “dangerous” permission, meaning that it is reasonable to find it amongst the permissions which help profile malware. The location of a phone is valuable information that might be of high interest to potential malware creators, if possible to get a hold of. As to the amount of URLs it is in no way unlikely that malware would have a higher amount of URLs in the app, since all of the locations to both download malicious information and all the locations to upload the user information would need to be included in the app. Finally the version number presented in the rule raises some questions, as it was assumed that malware would not be updated frequently. What is of note however, is that the version number is gotten from the android manifest, and can thus be set at will by the application developer. This means that the version number does not necessarily have a direct correlation to the actual version of the app. What this part of the rule might indicate is that the app developers set the version number intentionally so that it does not seem like the app is as new as it might in fact be. The reason for this might be to give the users a false security, assuming that the app is safe since it has been around and been updated while this is not, in fact, actually true.

Further research would be required to confirm the validity of these explanations. The applications could be examined to check for the presence of exploits bypassing the operating system, and given such a presence its effect on apk size would also be considered.

B. Rule Analysis - Association Analysis

The rules generated seem to be logical, with a clear correlation between sets of permissions involving one with a “dangerous” (as defined by [8])or undefined protection level (READ_PHONE_STATE and WRITE_EXTERNAL_STORAGE), some general permission (INTERNET) and positive virus profiling. The inverse is in part true with permissions of a “normal” protection level, such as BLUETOOTH and BROADCAST_STICKY, indicating negative malware profiling. Though permissions related to making calls and sending SMS messages, which have a “dangerous” protection level, are constrained to be missing from the permission set in the presented rules. One explanation could be that these particular sets of rules are given with respect to particular malware families, in which no such

permissions are used. The permissions just mentioned would then have to be present to differentiate from other malware families. Analysis of the code of some of the applications from the supposed malware families would be helpful in determining whether or not such patterns actually exist in the dataset.

Finally the rules chosen for presentation are similar, which is not unexpected if the variables correlate with the malicious truth value on their own.

C. Rule Analysis - Algorithm Comparison

A comparison of the rules generated by the algorithms, and thereby also of the algorithms themselves, would have been interesting. Performing such a comparison was also the plan at the start of the project. Currently it is however impossible to do such a comparison. This is due to the fact that the algorithms did not run on the same data. The data that the tree did run on was 90% benign apps and 10% malware, while the association analysis did run on a dataset made up of 50% benign apps and 50% malware. The fact that the algorithms ran on different sets of data means that it is impossible to fairly compare the resulting rules as well as comparing the algorithms themselves.

D. Method Criticism

Amongst the rules a few unintuitive rules were found one option for why this happened is that some important underlying features were not collected. This possibility would be rather reasonable since only easily extracted metadata from the marketplace and from the APK were used. If this is the case then there might be a clear pattern in the data, based on some of missing feature(s), that the rules try to mimic based on the currently available features. If this is the case then it might, by happenstance, be such that a seemingly random feature is the best available approximation of the missing feature(s). If this is the case then the model would then select this approximative feature in lack of a better option, leading to seemingly random features and rules being included in the final model.

As previously stated most of the rules generated for the association analysis were rather similar. This means that it is possible that only the first of these rules would have been needed for a relevant profiling. This could probably have been helped by the generation of more rules, since it is unlikely that all of the rules are this similar, it is probably just the case with the rules which have the highest confidence, since they likely profile a similar set of APKs. Unfortunately there was not enough time to generate these rules to see if there were any differences amongst later rules.

Along with this another problem is that the association analysis was only run on about 3,000 apps. This is since as stated in the method III-I there needed to be a balance between benign and malicious APKs for the algorithm to function correctly.

The reason to change the number of APKs for the association analysis was to find relevant rules, it should however be noted that it should be possible to find the rules even if the APKs are not removed, just that they will be present later in the list. The problem that occurred however was that running the algorithm long enough to try to find these rules was not feasible on the available hardware as the algorithm was run for five hours to then finally crash due to using more than 12GB of ram memory.

It should however be noted that changing the proportions of the APKs will, as previously stated, change the confidence of the rules, since it is possible that there would be more benign APKs who would also have the same permissions, or lack of permissions, as is presented in the rules. This would mean that out of all of the APKs which have all of the features presented in the rules a smaller proportion of them would be malware and thus the confidence would be lower. The only two other options to fix the presented problem, however, is to either buy better hardware or to reduce the amounts of viruses as well to keep the proportion until rules were found, neither of which were deemed as doable or a better solution.

There was also some bias towards benign apps in the decision tree, only a few of the rules ended up profiling malware, most only profiled benign apps. The reason why the data was not changed for the decision tree as well was since there did actually did exist some rules for profiling malware meaning that it was deemed worse to risk reducing the generalizability only to increase the amount of rules which profiled malware.

VI. Conclusions

This study contributes some possible rules to help profile malware. These rules can potentially be used by security analysts to help profile malware and identify which groups of apps might be most relevant as to finding new malware. These rules can also help profiling zero day exploits where the tools to actually identify the malware are not yet developed. Only a smaller set of rules were extracted for this report but the frameworks for extracting more, as well as for basing the rules on more malware, are set up and tested. Making it easy to extract more rules if required.

Most of the rules generated seemed intuitive, with a few that were harder to interpret. This shows that likely the method in general is valid, but with some minor aspects which could be improved. Seeing as the result would likely improve given a larger set of features, it can be concluded that profiling based on easily extracted features is possible, but not optimal.

References

[1] "Development of new android malware worldwide from january 2011 to march 2018," https://www.statista.com/statistics/680705/global-android-malware-volume, accessed on 2019-04-12.

[2] "App manifest overview," https://developer.android.com/guide/topics/manifest/manifest-intro, accessed on 2019-04-05.

[3] "About us," https://support.virustotal.com/hc/en-us/categories/360000160117-About-us, accessed on 2019-05-05.

[4] P. Rovelli, "Ninjadroid," https://github.com/rovellipaolo/NinjaDroid, May 2017, accessed on 2019-05-05.

[5] A. Muñoz, I. Martín, A. Guzmán, and J. A. Hernández, "Android malware detection from google play meta-data: Selection of important features," in *2015 IEEE Conference on Communications and Network Security (CNS)*, Sep. 2015, pp. 701–702.

[6] Y. Ishii, T. Watanabe, F. Kanei, Y. Takata, E. Shioji, M. Akiyama, T. Yagi, B. Sun, and T. Mori, "Understanding the security management of global third-party android marketplaces," in *Proceedings of the 2nd ACM SIGSOFT International Workshop on App Market Analytics*. ACM, 2017, pp. 12–18.
[7] "Meltdown and spectre," https://meltdownattack.com/, accessed on 2019-05-16.
[8] "Permissions overview," https://developer.android.com/guide/topics/permissions/overview, accessed on 2019-05-14.

APPENDIX

Because of issues with the graphical representation of the tree that could be generated, the tree is presented in text form, because of this some explanation of the format might be needed, the generated explanation is also presented in the first two lines of the tree.

Each row of the tree represents a split in the tree, starting from the root node. Each of the lines are presented on the form of:
Node Number), split condition, amount of data points in node (and potential children), classification value of data points in the node (only definitive if leaf node), (percentage of data points in node that are not viruses, percentage of data points in node that are viruses)

To note is that the update date feature means the number of days between the year 2000 and the date the app was last updated this means that a larger number indicates a more recent update, this was done to get a good and fair comparison between different apps for the model to process. This structure was also good since it meant that the date value of a given app would not change from day to day, which is important since the downloading process took several days. The size is given in number of bytes.

```
node), split, n, yval, (yprob)
      * denotes terminal node

  1) root 9192 0 ( 0.899043 0.100957 )
    2) READ_PHONE_STATE < 0.5 2394  0 ( 0.972013 0.027987 )
      4) update_date < 6395.5 1864   0 ( 0.998391 0.001609 ) *
      5) update_date > 6395.5 530  0 ( 0.879245 0.120755 )
       10) shell_commands < 13.5 144  0 ( 0.562500 0.437500 )
         20) size < 6.0739e+06 57   0 ( 0.964912 0.035088 ) *
         21) size > 6.0739e+06 87  1 ( 0.298851 0.701149 ) *
       11) shell_commands > 13.5 386   0 ( 0.997409 0.002591 ) *
    3) READ_PHONE_STATE > 0.5 6798 0 ( 0.873345 0.126655 )
      6) shell_commands < 231.5 3850 0 ( 0.810130 0.189870 )
       12) shell_commands < 30.5 277  1 ( 0.469314 0.530686 ) *
       13) shell_commands > 30.5 3573 0 ( 0.836552 0.163448 )
         26) urls < 120.5 2530 0 ( 0.881423 0.118577 )
           52) VIBRATE < 0.5 1108 0 ( 0.805957 0.194043 )
            104) update_date < 6394.5 698  0 ( 0.883954 0.116046 )
              208) update_date < 5011 277  0 ( 0.758123 0.241877 ) *
              209) update_date > 5011 421  0 ( 0.966746 0.033254 ) *
            105) update_date > 6394.5 410  0 ( 0.673171 0.326829 )
              210) update_date < 6420 304  0 ( 0.572368 0.427632 ) *
              211) update_date > 6420 106   0 ( 0.962264 0.037736 ) *
           53) VIBRATE > 0.5 1422  0 ( 0.940225 0.059775 ) *
         27) urls > 120.5 1043 0 ( 0.727709 0.272291 )
           54) ACCESS_COARSE_LOCATION < 0.5 390  0 ( 0.961538 0.038462 ) *
           55) ACCESS_COARSE_LOCATION > 0.5 653  0 ( 0.588055 0.411945 )
            110) version < 64 352  0 ( 0.863636 0.136364 )
              220) shell_commands < 153 67   0 ( 0.507463 0.492537 ) *
              221) shell_commands > 153 285  0 ( 0.947368 0.052632 ) *
            111) version > 64 301  1 ( 0.265781 0.734219 )
              222) version < 115 254  1 ( 0.153543 0.846457 ) *
              223) version > 115 47   0 ( 0.872340 0.127660 ) *
      7) shell_commands > 231.5 2948 0 ( 0.955902 0.044098 ) *
```