A Pragmatic Android Malware Detection Procedure

Paolo Palumbo[1], Luiza Sayfullina[2], Dmitriy Komashinskiy[1], Emil Eirola[3], and Juha Karhunen[2]

[1] F-Secure Corporation, Helsinki, Finland
[2] Department of Information and Computer Science, Aalto University, Finland
[3] Arcada University of Applied Sciences, Helsinki, Finland

Abstract. The academic security research community has studied the Android malware detection problem extensively. Machine learning methods proposed in previous work typically achieve high reported detection performance on fixed datasets. Some of them also report reasonably fast prediction times. However, most of them are not suitable for real-world deployment because requirements for malware detection go beyond these figures of merit.
In this paper, we introduce several important requirements for deploying Android malware detection systems in the real world. One such requirement is that candidate approaches should be tested against a stream of continuously evolving data. Such streams of evolving data represent the continuous flow of unknown file objects received for categorization, and provide more reliable and realistic estimate of detection performance once deployed in a production environment.
As a case study we designed and implemented an ensemble approach for automatic Android malware detection that meets the real-world requirements we identified. Atomic Naive Bayes classifiers used as inputs for the Support Vector Machine ensemble are based on different APK feature categories, providing fast speed and additional reliability against the attackers due to diversification. Our case study with several malware families showed that different families are detected by different atomic classifiers. To the best of our knowledge, our work contains the first publicly available results generated against evolving data streams of nearly 1 million samples with a model trained over a massive sample set of 120,000 samples.

Keywords: android, malware detection, static analysis, machine learning, classification, ensemble learning, feature selection

1 Introduction

The importance of the Android platform in the mobile operating system space is a well-known fact. During the first half of 2015, the Android platform represented 49.47% of the total mobile operating system market according to data provided by NetMarketShare [1], making it the most widely used mobile operating system

in the selected time frame. A mobile operating system with such a wide adoption rate clearly is at risk of becoming the target of criminals and other threat actors; according to the data collected by F-Secure Corporation [2], the first half of 2015 has seen hundreds of thousands of detections reported by mobile antivirus clients. This represents a clear continuation of the trend that was observed in 2014 and previous years.

Starting with the very first samples of Android malware discovered back in 2010, the Android platform has been under attack by increasingly sophisticated new malware. Polymorphic structure, the use of encryption, dynamic protection techniques and increasingly sophisticated communication techniques with attacker-controlled Command and Control servers are just some of the issues that contribute to the growth in complexity of Android malware. The reasons for this evolution of malware are numerous. For example, malware authors are always trying to counteract the detection capabilities developed by security vendors; similarly to legitimate software, malware is software that is continuously developed and maintained; finally, malware needs to take advantage of the latest vulnerabilities and security flaws to infect, spread and persist. Additionally, this increase in sophistication clearly reflects the perceived value that Android devices have to the attacker's eyes, and as a result, this new malware is increasingly more complicated to detect in a generic fashion, for example through the use of machine learning-based approaches. Difficulties in detecting the malware reflect in malware that is able to survive longer before being detected. As a result it is able to carry out its purpose for longer; and malware that is able to sneak past the security countermeasures of App stores, including the official Google Play store, and therefore is able to reach a much wider pool of potential victims.

To address these problems with Android malware detection several approaches have been tried in the past. However, observation of the security for the last years has shown that even a company like Google [3], which has often been at the forefront of research and has a concrete interest in keeping the Android ecosystem free of malware, has failed to prevent malware from sneaking into its own marketplace [4–7]. This situation made us believe that the problem of automatically detecting Android malware is still far from being solved.

This paper makes the following contributions:

1. **Presenting an approach to automatic malware detection that is based on a number of atomic classifiers for specific features extracted from Android APK files which are then combined by an overall classifier in a typical ensemble-classification model.** This approach is designed to be more effective both against new variants of existing Android malware and against samples belonging to new, previously unseen families, while at the same time minimizing the amounts of false positives. Although ensemble methods are not novel, we highlight the importance of using different feature groups separately by each atomic classifier to improve robustness. Our proposed approach, thanks to its low computational complexity, is very easy to deploy; we consider this to be an extremely important asset for using in real-world operations. We compared the False Positive rate

(FPR) with the one obtained using the approach outlined in the selected baseline paper both on the evolving streams of data and on fixed test set and observed significant decrease of FPR. The model is described in Section 5.

2. **Providing clear usage scenarios of automatic malware detection systems and defining the associated requirements**. These requirements include complexity, acceptable False Positive and False Negative rates, low dependency on external tools, easy retraining and robustness towards new malware families and obfuscation; several considerations about obfuscation are made in Section 8. We show that our whole approach from data collection to prediction satisfies most of these requirements and is particularly useful for filtering system that prioritizes what will eventually reach a human analyst. The details are provided in Section 2.
3. Highlighting the importance of being chronologically consistent when performing both training and evaluation. To properly simulate real world usage scenarios, it is clear that the process of training a classifier should include the selection of a specific date. **All the samples used for training should be anterior to this selected date and, correspondingly, all the samples used for testing should be posterior to this date.**
4. **Evaluating how the different types of features extracted from APK samples contribute to the overall detection capabilities.** We present a breakdown of the contribution of each feature type in Section 7.2.
5. **Providing detection results both for test data and for real, evolving data stream using a model trained on 120,000 samples with confirmed labels and over 1 million samples obtained via evolving data streams**. A FP rate of 4.07% achieved by testing on the evolving streams of data is acceptable as a filtering tool for experts tasked with analysing malware. Experimental results are presented in Section 7.

The rest of the paper is organized as follows: first we clarify the problem statement (Section 2); then we present a survey of existing approaches for Android malware detection and classification using machine learning techniques (Section 3). Later we describe the format of the data our approach is based on and explain how this data is collected (Section 4). Our method and its implementation are described next (Section 5), followed by extensive experimental results (Section 7) and conclusions.

2 Design requirements

When designing automatic malware detection systems, it is of high importance to understand how such systems will be used, especially in the context of a wider automation that is inevitably present when working at an industrial scale; it is critical to understand how such system could be contributing to the backend automation of an anti-virus (AV) vendor. Specifically, we can identify three different use cases for such systems:

1. **Detection.** The automatic system could be integrated as a black box that receives as an input a new, previously unseen object and is able to independently classify the object as malicious or not. The verdict produced by such system would then be unquestionably propagated to the object in question, and used directly for protecting users encountering such an application in-the-wild. It is clear that such a use case is an extremely challenging one, as it assumes that the verdict given by the system is completely reliable. In such a scenario, precision is the most important requirement. In case of false positives, user-experience is impacted: for example, users trying to install the application would be informed of its maliciousness and prevented from installing in fact a perfectly valid application. Moreover, application developers would also be impacted, as these wrong verdicts could result in missed revenues and reputational damage. Finally, there might be a severe reputational harm for the company that forces such a wrong verdict onto their users.
2. **Categorization.** Such an automatic system could be deployed as a black box that receives as an input a new, previously unseen object, and is able to independently classify the new object as belonging to a certain malware family or to a known group of specific applications. A mistake in such a scenario would result in the new application being reported as similar to a wrong group of existing applications, leading to false positives or false negatives. Additionally, the wrong labeling of the new application is likely to generate a cascade of different issues in the greater context of a wider automation; for example, the wrong labelling of a sample as belonging to a certain application group could erroneously trigger a set of expensive actions resulting in the degradation of performances for the overall backend automation. It is clear that before an automatic system can be used in this particular fashion, its precision must have been confirmed to be high enough, perhaps after the system has been successfully operating as a standalone classification system.
3. **Prioritization.** The automatic system could be integrated as a black box that receives as an input a new, previously unseen object, and is able to assess independently the maliciousness of the object. Such assessment would then be used to augment the information available to backend-side about the new object and to prioritize and optimize expensive human intervention and the usage of heavier, more computationally expensive automatic systems. Erroneous outputs in this situation are problematic, but somewhat less severe than in the two previous scenarios; in fact, wrong decisions in this situation could trigger the intervention of human analysts or more expensive automatic systems, therefore wasting resources without need. It is clear that, were these kind of mistakes to be frequent, there would be a significant degradation of the operations of the security company using such systems.

Additionally and independently of the particular use case, it is clear that such system should satisfy two additional generic requirements:

1. Such a system should be computationally efficient, both from training and of prediction points of view. Considering the case of an anti-virus company

using this system, such a company might receive well over thousands of new unknown objects every minute, meaning that prediction times need to be adequate. Similarly, to account for the changes in the threat landscape, the system will need relatively frequent re-training, and therefore a system that takes days to train might be unacceptable from an operational point of view. Similar considerations can be done for resource consumption and scalability.
2. Such a system should not rely heavily on the external dependencies. While there are available a number of very effective tools that can be used as part of an automatic system for malware analysis, these third party tools might be a liability in the long term. For example, the developer of the third party tool disappears leaving the tool broken or unable to cover the evolution of malware. Therefore it is preferable to be able to control all parts of the automatic system, for example by developing them in-house.

To this end, we adhere to a strict methodology when it comes to the design, implementation and testing of our proposed solution. First, a sizable amount of data is collected from a trustworthy data source, for which we possess verified labels, often the result of human categorization. Then we extract an extensive description of each application in our dataset using our own pre-processing tool. Later we select the most relevant features for malware detection and apply the improved Naive Bayes classification approach [8] in an ensemble-based classifier which satisfies the generic requirements described above. After that state-of-the-art testing methodologies are applied to our system and these results are discussed at length. Finally, the testing methodology is extended to include evolving data stream as we see from an anti-virus company submission feed.

3 Related work

Research efforts in automatic malware processing focus on a number of possible goals. The majority of the relevant literature has focused on malware detection [8–19], where the goal corresponds to the first and third use cases we discussed in the previous section (usually depending on the required precision). Given an object of interest (for example, a file object), it is necessary to make a decision on whether the object is malicious or not.

Some research has also been conducted in malware identification [20, 12], corresponding to the second use case from the problem statement; the main task of malware identification is to identify the specific malware family the object of interest belongs to.

A third goal is malware behaviour characterization [12], focusing on mining structural and behaviour patterns in order to precisely identify particular groups of objects of interest. For example, given a family of malicious objects, we want to understand what behavioural characteristics make this family of objects unique.

To reach these goals researchers use data that represent appropriately the objects of interest. There are two basic groups of techniques to extract and assemble data for this kind of representation – static and dynamic. Static approaches primarily focus on getting information that describes the objects' layout, structure

and content. One drawback of static techniques is that in certain situations they cannot provide a usable representation of the data: for example, if the code of the particular object has been obfuscated, static processing of the object will produce substantially weak data and therefore this data cannot guarantee a system to be reliable enough in terms of precision and recall. At the same time, there is a clear advantage when it comes to static processing: static techniques are often computationally light. When it comes to dynamic processing, these approaches focus on collecting information during the execution of the objects in proper environments (e.g., operating systems, virtual machines, sandboxes). Obviously, such data gives a better understanding of the objects' behavioural patterns that would not be completely collectable by using static analysis techniques. Unfortunately, dynamic analysis techniques have their own set of problems: first of all, developing tools that allow the dynamic analysis of malware is very challenging. Additionally, such techniques require extensive resources and often do not scale enough to be practical.

An Android package file is a compressed container of files (embedded items) located in a number of file directories, where Dalvik executable (classes.dex) and manifest (AndroidManifest.xml) files are the most interesting static data sources and represent the basic functionality of an application. Other items embedded in the APK container could be additional native executables or resource files. While being executed, the application interacts with a number of layered execution environments of Android operating system. The Dalvik executable is executed by Android runtime, which is in turn supported by the native execution environment and accompanying native libraries which in own turn use Linux kernel services. Correspondingly, the possible dynamic data sources giving behavioral information can be implemented at the level of these environments by adopting existing instrumentation and development tools available for Android and Linux kernels.

Most of the previous work has been based on static processing, the static data is extracted from mandatory items embedded in analyzed Android Packages. A dynamic approach is used only in some cases [14–16], where the particular data is collected by instrumenting operating system.

Most of the discussed research work relies on freely available tools created by the security community and academia. Particularly, tools like Androguard [21], Apktool [22] and Baksmali [23] are commonly used by many researchers in several contexts. All the tools rely on information provided by Google [3].

Among all papers reviewed, only a single one takes into account the problem associated with not considering the chronological order with the samples used for training and testing classifiers for malware detection [16]. The authors use Control Flow Graphs (CFGs) extracted from APK samples to demonstrate that using random selection to create a training and testing sets from a group of samples is an approach that introduces a significant bias. In order to reconstruct a chronology, the authors of [16] make use of the compilation time reported in APK container. In contrast, we use the date of first observation to sort the objects in this paper.

From the set of works that we reviewed, we observed the use of the following supervised learning approaches: k-nearest neighbours (k-NN) [9, 20]; various implementations of Decision Trees [9, 10, 12, 13, 18], including Classification and Regression Tree, Quinlan's ID3 and C4.5; Naive Bayes models [8, 18, 19]; and Support Vector Machines (often in classification context referred to as Support Vector Classification, SVC) [9–12, 17]. At the same time we see that ensemble learning approaches based on boosting (AdaBoost) and bagging (random forest, RF) are used also in [10, 12, 13, 18, 24]. When it comes to unsupervised learning techniques, we remark the use of k-Means clustering in [13] and [14]. The former paper uses the clustering means to divide the initial input data set into a number of groups of similar instances (thereby assigning the obtained cluster's labels to the input instances) and then uses supervised learning approaches. In the latter paper, the authors expectedly set the number of clusters k to 2 (malicious and benign) and use the obtained decision model for further detection of new malicious instances.

One of the most important aspects that defines the practical usefulness of proposed approaches is the quality of the data used during the training phase. There are two main points that have to be considered on the matter. First of all, the amount of available data has to be diverse and large enough to represent the objects of interest. This said, while considering the existing amount of Android applications and their families, we estimate the minimal acceptable size for the training data as tens of thousands of instances. Secondly, the sources of the information that are used to assign labels to the training objects have to be trustworthy. In this scope, such sources are antivirus vendors and specific research project databases. For example, in order to mitigate the labeling issue, some research teams use antivirus-based validation [8, 16, 17], reputation of Android markets [18] and various aggregating services like VirusTotal [25]. At the same time we still consider the practice of using various Android markets for labeling the training data risky, especially for the cases of preparing benign data sets. Out of investigated papers, the biggest datasets were used in [17] and [16], where the training set is of size 12,158 and 200,000 and test set is 135,792 and 200,000 respectively.

4 Dataset Description

According to the discussion in the previous section, one of the biggest challenges encountered in research work in the area is the unavailability of a big enough volume of recent and trustworthy data. In this study, we have access to a database of malicious and clean samples retrieved and processed by F-Secure Labs. The samples originate from different sources: for example they are gathered by employed investigators or submitted by users of F-Secure products. In subsequent processing, each sample is given one of three labels:

– "Knowingly benign" (a.k.a., "knowingly clean" or "clean") – this label is assigned only after human inspection verifies that the sample does not contain malicious or potentially unwanted functionality, or it is confirmed to

originate from a trustworthy manufacturer. As this label requires manual intervention, the cost of producing such labels is relatively high.
- "Knowingly malicious" (a.k.a., "malicious") – if the antivirus vendor's existing detection infrastructure identifies the sample as malicious, it is automatically labelled as such. The systems are executing human created rules based on previous observation.
- "Unknown" (a.k.a., "undetected") – this is the default label given to any instance not assigned to any of the previous labels, i.e., a sample that is not flagged by existing methods, and has not been manually checked. The majority of available samples fall into this category. Most of these samples can be expected to be benign, but a fraction is likely to represent undetected malware.

Table 1 provides a partition of the datasets used in the context of the study and highlights both the sizes of the datasets and their associated label. It is important to note that often in research studies, objects from the "Unknown" category are defined as benign. For obvious reasons, we consider this assumption risky for understanding and interpreting the results obtained in the context of these studies; we instead use entities with this label for estimating the real contribution of the approach we developed (i.e., the ability to detect new, previously unseen malicious objects and therefore the added value of the automatic system).

Table 1. The number of samples available in each category

	Label		
Dataset	**Knowingly benign**	**Knowingly malicious**	**Unknown**
Training	61,249	61,481	N/A
Testing	N/A	304,259	1,048,116

4.1 Dataset Collection

The training data set was collected during the time period preceding our experiments. The samples from the "knowingly benign" subset were collected according to their discovery date, starting from the time when the first android malware was observed up to the end of 2014. The samples belonging to the "knowingly malicious" subset were randomly chosen from the set of all malicious android samples discovered in the time interval beginning with the first of June 2014 and ending with the 25th of October 2014.

Both of the subsets used for testing purposes ("knowingly malicious" and "unknown samples") were collected between October 2014 and January 2015. The length of the data collection period and our ability to collect precise timestamps for samples in the testing set give us an opportunity to observe the evolution of the system's performances over time. Additionally, given the fact that

data included in the testing sets is collected after the conclusion of the training phase of our classifier, we have the unique and exciting opportunity of understanding how the system performs in a real life scenario. In order to emphasize this important aspect we contributed to the testing methodology, we denote the used testing sets as evolving data streams.

4.2 Feature Extraction

One issue, highlighted in the previous sections, is the heavy dependence of previous studies from the available set of external tools for processing input instances. In order to mitigate this aspect, we implemented our own APK processing tool that focuses on the types of features we were interested in. The tool goes through all items embedded in an object of interest, identifies their types and extracts available static data. For each embedded item, independently of its type, the tool collects its external attributes as filename, path, size and "sha1" hash.

A number of mandatory embedded items, such as Android manifest file *AndroidManifest.xml*, Dalvik executable file container *classes.dex* and resource file *resources.arsc* are completely parsed in order to extract additional information that is considered meaningful for our study.

In order to form the initial collection F of used features, we extract and use the ten following subsets of data from the items embedded in a generic Android package:

- Permissions (1) and other strings (2) from AndroidManifest.xml, denoted as F_{MFP} and F_{MFS} correspondingly
- Prototypes (3), types (4), methods (5), fields (6), names of classes (7) and other string (8) information from classes.dex, denoted as F_{DEXP}, F_{DEXT}, F_{DEXM}, F_{DEXF}, F_{DEXC} and F_{DEXS} correspondingly
- String (9) information from resources.arsc, denoted as F_{RSRC}
- Hash (10) values of all embedded items, denoted as F_{HASH}.

The described static information represents the main functional properties of the analysed objects, and therefore, indirectly characterizes their expected behavioural patterns. For example, permissions data indicate the main privileges the application requires in order to be executed by the Android operating system. Another clear example is the presence of hardcoded URI in classes.dex: such URIs usually indicate that an application is supposed to perform interaction with a specific network host or service. Using the variety of possible behavioural indicators along with the availability of substantial amount of training instances leads to making malicious APK detection systems fast, precise and generic. Such considerations explain the choice of feature sets we made. Further details on the meaning of the used data can be obtained from numerous data sources, for example [3].

5 Classification

In this section, we will describe the Machine Learning ensemble-based classification approach used for malware detection. To this end, we begin with the

baseline methodology using Naive Bayes classifier (NB), then describe in detail the suggested ensemble approach, which is based on atomic NB classifiers.

5.1 Naive Bayes

Naive Bayes [26] is a very computationally efficient linear model that is suitable for practical implementations, as well as fast in retraining. Naive Bayes classifier has been extensively used with high-dimensional datasets, especially for text classification tasks.

Our baseline method [8] uses a Normalized Bernoulli Naive Bayes classifier which implements the following modifications in contrast to the canonical Bernoulli NB classification approach: (1) the target likelihood is estimated without modelling absence terms and (2) the final sum of log factors is normalized. Given a file f, represented as a set of features, the ratio of the probability of it being malicious to the opposite is represented by the following expression:

$$\log \frac{p(label = malicious|f)}{p(label = benign|f)} = \log \frac{p(malicious)}{p(benign)} + \sum_{i \in f} \theta_i. \tag{1}$$

The used Laplace smoothing procedure adopts smoothing parameter $k = 1/N$ where N denotes the number of used training samples:

$$\theta_i = \frac{\frac{c(w_i, malicious)+k}{M+2k}}{\frac{c(w_i, benign)+k}{B+2k}}. \tag{2}$$

In Equation 2, the numbers of malicious and benign training instances having feature w_i are denoted as $c(w_i, malicious)$ and $c(w_i, benign)$. Correspondingly, M and B are the total number of malicious and benign training instances.

5.2 Ensemble learning

An ensemble of classifiers can be seen as a set of classifiers, whose individual predictions about the class are combined for making a final prediction [27]. Each separate i^{th} classifier makes its own hypothesis $H_i(x)$ about the class of the classified object x. One necessary condition for choosing atomic classifiers for the ensemble is their diversity, meaning that they make wrong predictions on different data points. The way to combine the outputs of separate classifiers can be diverse, including weighted average approach or majority voting.

One of the reasons to extend Normalized Naive Bayes with ensembles was to provide even more robust detection, where different groups of factors are taken into account separately. Another reason why ensembles are preferable for detection is partially dealing with obfuscation. Although we select the features by minimum frequency and most of the obfuscated features will not fall into the selected set of features, the classifier may rely on features that clearly are not associated with code, like APK permissions and resource strings.

5.3 Proposed approach

So far we have explained the baseline approach using the Normalized NB. We propose to use the Normalized NB as the separate classifiers for the ensemble. Each Normalized NB is trained on a different set of features and provides a confidence (probability) about the malicious class. Those confidences in turn are used in a Support Vector Machine classifier [28], which provides the final decision. Below we elaborate on our approach.

In the proposed approach a file instance f is represented by the feature spaces from the collection $F = \langle F_{MFP}, F_{MFS}, F_{DEXP}, F_{DEXT}, F_{DEXM}, F_{DEXF}, F_{DEXC}, F_{DEXS}, F_{RSRC}, F_{HASH} \rangle$ where each True value denotes a features existence and False its absence.

Particular representations $R_k(f)$ of file instances that are obtained during the objects' mapping procedure $R_k : f \to F_k \times \{malicious, benign\}$ to feature space $F_k \in F$ are then used to instantiate particular entities of Normalized Bernoulli NB classifiers NBC_k for each feature space.

The output values $c_k(f) = NB(R_k(f))$ representing confidences for malicious class obtained from each Normalized Bernoulli NB classifier form the second level representations $C(f) = \langle c_1(f), \ldots, c_k(f) \rangle$ of training instances that in turn are used for training the regularized Support Vector Machines classifier [28], which combines the results obtained via the ensemble of NB classifiers. To this end, the approach uses standard [29] implementation of SVM wrapped by scikit-learn library [30].

Figure 1 presents the main structural elements of the suggested classification scheme together with their connections.

6 Operational performance

As our system is designed to work at an industrial scale, precise running time measurements are difficult to retrieve; even if that would be possible, they would represent the performances of the system when implemented on specific hardware and processing flow. For completeness' sake, however, we include the data we collected when performing our tests on a dedicated single machine. Our test environment consisted of a test machine with 16 GB RAM, Intel Xeon E5430 quad-core CPU at 2.66 GHz, running on 64 bit Linux (Ubuntu). The tests were conducted with a single threaded toolchain that included:

1. APK parser (raw feature extractor) written in Java
2. Python-based learning / testing framework (based on scikit-learn [30]).

Based on the above scenario, we observed that the biggest amount of time is spent by the system when parsing an APK file; this processing took on average 3 seconds per sample and we will not consider time used for this task in the remainder of the performance analysis. When presented with the full training set, the system described above was able to complete the training procedure and produce an output model within 8 hours; with this model prepared the prediction

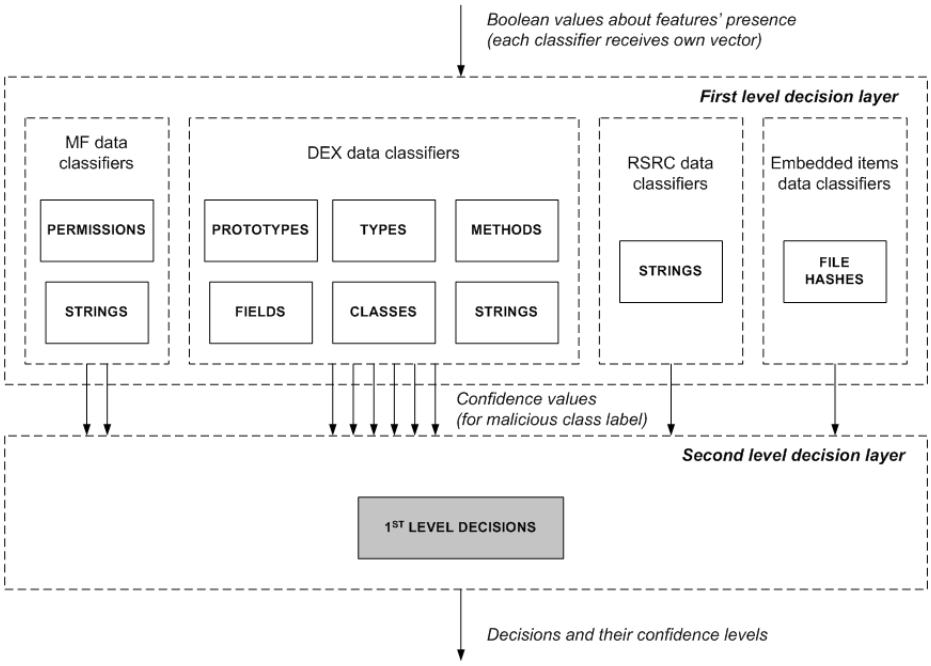

Fig. 1. Graphical representation of the suggested classification scheme.The white rectangles denote Normalized Bernoulli NB classifiers that are used at the first level of data processing; the text comments inside these items refers to the subsets of Boolean features that they use. The grey rectangle at the bottom denotes the Support Vector Machine classifier; its feature space is constructed with the confidence values produced by the first level classifiers. The output of the proposed approach is a tuple containing the predicted label and its confidence.

phase took on average 0.1 seconds per sample. Considering the numbers above, we can state that already a small system like the one we used for testing can process around 1,200 objects per hour from end to end. While this information is specific to our particular environment, it provides a lower bound when it comes to throughput expectations.

7 Experimental results

The development of the approach presented in the paper is primarily motivated by the practical study we performed against the baseline classifier. Having access to a big enough evolving data stream, we managed to establish a properly arranged testing process and found out that although the approach provides reasonable 90–98% of True Positive decisions, the amount of the produced False Positives is substantially high.

Analysis of these problematic situations allowed us to identify the root cause for such high false positive rates; the "knowingly clean" samples in the training

set do not provide enough coverage of Android applications using third-party, behaviourally neutral components that are also often used in malicious applications (for example, SDK components from Web analytics and advertisement manufacturers and software packers or protectors).

7.1 Feature selection

To ensure a fair and valid comparison between the two approaches, during data collection and the training of the classifiers we use functionally identical tools, training sets and feature spaces. In order to provide the latter aspect, the feature selection procedure $F_k \rightarrow F_k^{'}$ for both classification approaches is as follows:

$$F_k^{'} = \{w_i \in F_k \,|\, c(w_i, benign) \geq t \wedge c(w_i, malicious) = 0 \\ \vee\, c(w_i, malicious) \geq t \wedge c(w_i, benign) = 0\} \quad (3)$$

The numbers of malicious and benign training instances having feature w_i are denoted as $c(w_i, malicious)$ and $c(w_i, benign)$ whereas the selection threshold is denoted by t. In other words, features presented only in one class with more than a threshold t were selected.

The full dataset contains more than 40 million unique features, and thus in order to provide sufficient reduction to a more manageable number, the selection threshold t is set to 100. The procedure was performed against every feature subset except F_{MFP} (manifest permissions), which does not need reduction due to its initially reasonable size and the necessity to outline main functional characteristics, e.g. given privileges, of training instances. The results of feature selection procedure are presented in Table 2.

7.2 Evaluation of feature importance

To better estimate the contribution of each individual specific classifier towards the common goal, we performed a dedicated set of experiments. These experiments were conducted by selecting in each case a specific NB classifier and we evaluate its performances against the training set described before; the results include only those decisions made when the confidence level of the decision taken by the classifier was 0.99 or above. Table 2 presents the results of our experiments. Please note that the results are described in terms of feature types as opposed to classifiers; this is because each classifier works with a single specific feature space and therefore we can do this mapping. As it can be seen from Table 2, the most valuable feature type is F_{DEXS}, that encompasses all strings that are not specifically referenced otherwise in the DEX header; examples of DEX strings are hardcoded web links, command line strings to invoke additional programs, various debug messages and so on. We can make also an important consideration about APK permissions: in fact, while these are extremely useful as supporting features, the results of our evaluation clearly highlight why relying on them to properly distinguish between malware and non-malware is not enough.

Table 2. Ranking of individual features' usefulness for detecting malware with 0.99 confidence

Subset	Total number of features	Number of selected features	Percentage of detected malicious objects
F_{DEXS}	9,170,037	131,807	91.97
F_{MFS}	330,584	4,714	80.22
F_{HASH}	7,371,961	29,327	77.96
F_{DEXT}	5,696,265	81,244	76.83
F_{DEXM}	2,301,487	36,179	76.60
F_{DEXF}	5,788,696	79,769	66.38
F_{RSRC}	9,022,058	352,853	64.16
F_{DEXC}	569,925	14,229	41.93
F_{DEXP}	60,353	1,081	34.75
F_{MFP}	4,589	4,589	0

7.3 Classifier's evaluation against evolving data streams

In order to evaluate the classifier's performances using FPR and recall, we tested it for an extensive period of time. The experiments were conducted by first training the model with the previously described training set; this model was kept constant for the duration of the experiments. For each day during the testing period, we then fed the model with samples from the evolving data streams. At the end of the testing period, we aggregated the collected results. The information about obtained positive decisions is summarized and presented in Figure 2.

Figure 2 shows the results of the classifier's TPR measurements for the evolving data set of known malicious samples. During the specified time period, our classifier detected 259,630 unique malicious objects (85.33% of all instances from the data set).

If we focus on the linear trend, it can be observed that the recall for known malware samples decreases over time. This can be explained as follows: we can expect known families to give way to new, possibly more sophisticated ones that will be detected by anti-virus programs only after our training was completed; therefore we expect the capabilities of any system with a restricted amount of knowledge obtained during a training phase to decrease over time.

Figure 3 presents the results of applying the trained model to the evolving data set of unknown samples. The solid line shows the measured fraction of positive predictions (i.e., it indicates how many potentially malicious objects are present in the stream of unknown samples) performed for each day of our monitoring period, while the dotted one represents the linear trend. For the whole time period the classifier labeled 319,828 unique instances as malicious (30.51% of all instances from the data set). For the sake of clarity it is necessary to state

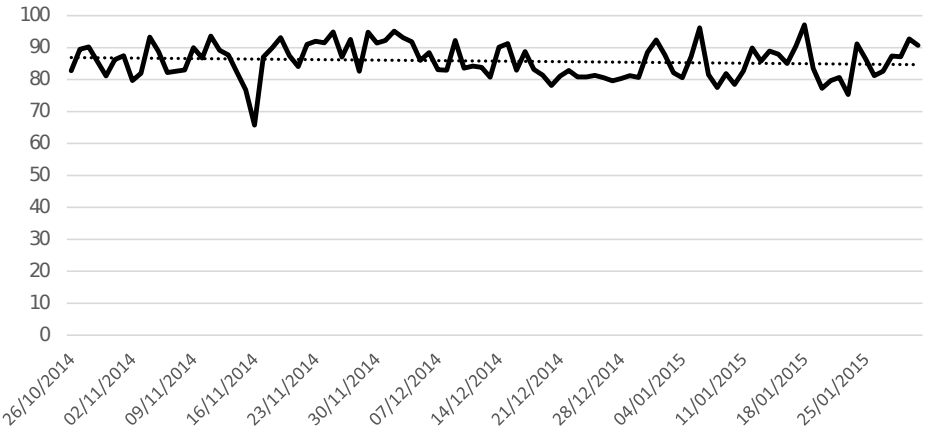

Fig. 2. Percentage of detected samples from the malicious evolving stream. The solid curve indicates results for each particular date starting with 26th October 2014 up to 31st January 2015. The dotted line represents the linear trend for the TPR. The recall for known malware samples decreases over time.

that the obtained numbers cannot be considered objective due to two reasons. First, this fact must be combined with additional information about the expected error rate (FPR) that is discussed in the next paragraphs. Secondly, even though the FPR is known, it is still necessary to validate some of the decisions with human analysts' assistance that is beyond this paper's scope. Overall, the presented results give us an indication of how many new, previously undetected malicious objects could be present in a stream of samples after scanning via traditional means, such as antivirus signatures and heuristic engines. Following these considerations, we define this rate as Unconfirmed True Positive Rate (UTPR).

If we again focus on the linear trend from Figure 3, we can observe an increase in the detection rate over time. The main explanation for this behavior is that in this particular timeframe the antivirus vendor's systems received an increasing amount of malware undetected by standard methods that uses techniques that are similar to those used by the samples our system was trained against. At the same time, another possible explanation is that this increase is justified by a growth in the false positive rate of the classifier as time goes by; in fact the features that were more prevalent in malware samples at the time of training might have been increasingly used in non-malicious applications produced after training. For example, if during the training phase a specific advertisement platform is used mostly by detected samples, there is the chance that as time goes by the same advertisement platform would become popular also in non-malicious applications.

In order to estimate the expected error rate for the positive decisions produced for the samples belonging to the evolving data set of unknown samples, we performed the same evaluation procedure that was described above. Thus, each

unconfirmed positive decision (malware detection from the unknown data set) was validated against the information available via the VirusTotal [25] service that aggregates most AV detection engines. To this end, each positive decision was confirmed so that if a given object is detected via the service by several engines, then the decision is considered correct, otherwise the decision is considered to be a false positive. This procedure is only partially automated and time consuming. Therefore, to verify our results, we selected as a benchmark the same set of that was used to verify the Normalized Bernoulli Naive Bayes classifier; specifically we selected the 17,317 unknown Android samples that were collected on 31st January 2015. The final results of the carried out validation are summarized in Table 3.

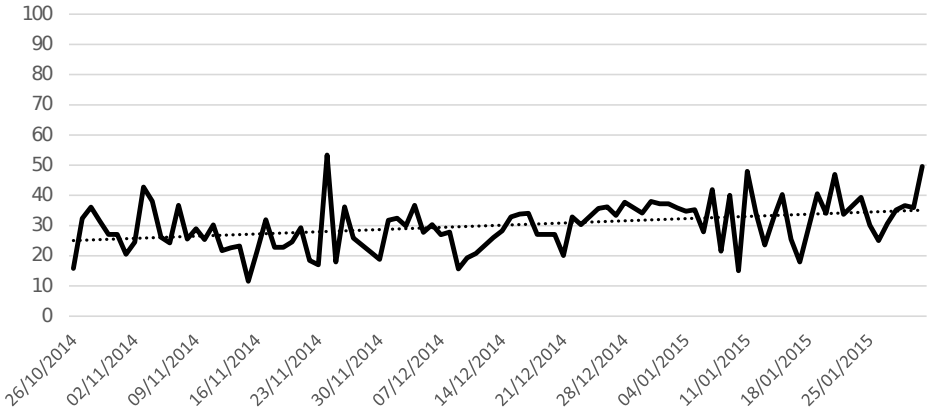

Fig. 3. Percentage of detected samples from the unknown evolving stream.The solid line shows measurements performed for each day of our monitoring period while the dotted one represents the linear trend. We can observe an increase in the detection rate over time.

Table 3 describes the results of the comparison between the selected baseline and proposed approaches. When it comes to the 10,688 knowingly malicious samples obtained on the previously mentioned date, the results show that the baseline approach performs better than the proposed one if we consider TPR rate (column: TPR for knowingly malicious set). However, when we consider the performances against the set of undetected samples, we can see that the performances of the suggested approach are significantly better than those of the baseline. The last column of the table, "Unverified positive decisions", refers to the percentage of samples that were detected by the two approaches but that could not be verified using the previously described validation procedure; this is because, at the time of verification, those samples were not known to VirusTotal. This could be interpreted as a possible sign of maliciousness, but that is not necessarily the case.

Table 3. Comparison of accuracy between the baseline method and the new approach on the evolving stream of data. The knowingly malicious dataset contains only malware, and the precision is thus always 100%. For the "undetected" dataset, the number of false negatives can not be counted, and hence only values for precision and false discovery rate are presented.

Method	**TPR, %, knowingly malicious**	**Precision, %, undetected**	**False Discovery Rate, %, undetected**	**Unverified Positive Decisions, %, undetected**
Baseline	98.64	75.49	16.06	8.45
New approach	90.82	88.08	4.07	7.86

In this experiment, the model was kept constant during the evaluation on the evolving data stream. The trained model could, however, easily be updated when new labels are available. In the Naive Bayes classifier, updating the model can be done by updating existing feature counts without the need to go through old training samples again. The SVM classifier on top of the Naive Bayes classifiers needs to be retrained each time, but due to having only 10 inputs from 10 Naive Bayes probability estimates, it is of low cost. Implementations of online learning with SVM have also been studied, e.g., in [31].

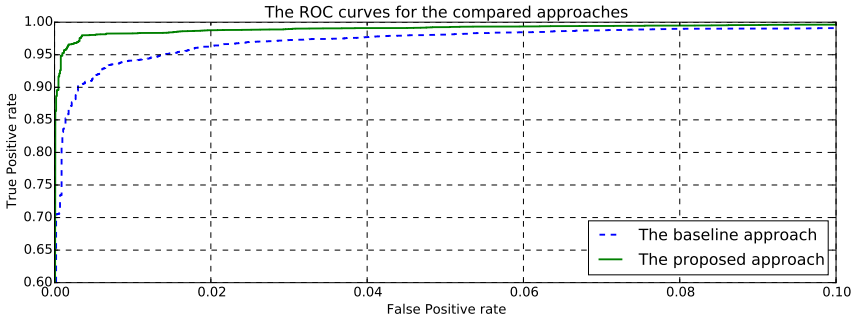

Fig. 4. ROC curves for the proposed approach and baseline approach on the static data set. Note that both axes have been zoomed towards the top left corner.

7.4 Classifier's evaluation against a static set

Additionally, we compared our approach with the selected baseline by applying traditional methodologies on the fixed sample set from 2014. The fixed sample set is overall composed of approximately 120,000 samples; 40,000 samples have then

been randomly selected for the purpose of this experiment. 20,000 of these samples have been added to the training set, while the remainder has been marked as belonging to the test set. Each of these sets contains 10000 malicious samples and 10,000 clean samples. For our experiments we have selected a threshold value for the FP rate of 0.01%; we do consider a low FP rate to be critically important for this kind of systems in an industrial scenario. We do however provide full ROC curves for the interested reader; the curves are presented in Figure 4. From Table 4, we can see that our approach improves over the baseline approach: the new approach provides an increase of about 30% when it comes to the TP rate with a small fixed FP rate of 0.01%. Table 4 provides additional insight into the performances of the two approaches for the selected parameter. Even considering that we compare only against the chosen baseline paper, the produced results, particularly the increase in TP rate, represent a significant improvement over existing work in the field of Android Malware detection.

Table 4. Comparison between the baseline and the new approaches on the fixed training (20,000 samples) and test (20,000 samples) data sets; acceptable false positive rate is 0.01%.

		Real labels		Precision
		Malicious	Benign	
The baseline approach, ROC AUC is 0.99602				
Predicted labels	Malicious	5421	1	0.9998
	Benign	4579	9999	0.6859
Recall		0.5421	0.9999	**Accuracy:** 0.7710
The proposed approach, ROC AUC is 0.99799				
Predicted labels	Malicious	8485	1	0.9999
	Benign	1515	9999	0.8684
Recall		0.8485	0.9999	**Accuracy:** 0.9242

8 Protected samples

Clearly, any purely static classification system will be unable to correctly handle all of those samples that are protected against analysis and inspection. One of the difficulties with these samples comes from the use of obfuscation. Obfuscation is a common technique used to prevent security software from inspecting the contents of malicious files. Obfuscated code is traditionally best handled via dynamic analysis techniques, i.e., taking advantage of the relevant (virtual or real) CPU in order to remove the obfuscated layers; attempting similar operations through static techniques is often time consuming and not effective in the long term.

At the same time, not all the obfuscators are the same. In some cases, the obfuscator may not process the entirety of a target file's structure and content. In such cases, the features not modified by the obfuscator can be leveraged by the proposed ensemble-based approach. Additionally, many obfuscators modify their targets in such way that these modifications are reflected in specific features spaces [32]. As previously mentioned, these cases are well handled by the proposed approach.

Even in those cases where the proposed approach is not helpful in bypassing the protective measure, it provides significant value in the form of operational optimization. In fact, even in such cases the proposed approach can easily identify samples that are protected and provide such information to the rest of the automation for appropriate handling; this could mean raising the priority of the sample so that it undergoes deeper and more expensive dynamic analysis.

We do however consider the problem of dealing with packed and protected APK samples as a significant challenge and leave further investigation to future work.

9 Case study

In this section we investigate a few selected cases that help us illustrate the effectiveness of the proposed ensemble-based approach. By comparing the performances of the baseline NB classifier and the new, ensemble-based classifier we are able to show that the increased precision of the ensemble classifier is at the expense of the recall; this lowered recall, however, is still high enough to allow the use of the classifier in a common data processing pipeline.

Table 5. Confidence values from atomic classifiers for Android malware families. Different families are detected by different atomic classifiers.

Feature subset	**BaseBridge.C**	**SmsSend.OC**	**SmsSend.WB**
F_{DEXP}	**0.996**	0.500	0.500
F_{MFS}	0.500	**0.998**	0.500
F_{HASH}	0.500	0.500	**0.991**
F_{DEXM}	**0.996**	0.500	**0.991**
F_{DEXS}	0.500	**0.998**	**0.992**
F_{MFP}	0.560	0.554-0.559	0.617
F_{RSRC}	0.500	0.500	**0.991**
F_{DEXF}	**0.996**	0.500	**0.991**
F_{DEXT}	**0.996**	0.500	**0.991**
F_{DEXC}	0.500	**0.991**	**0.991**
Second level (SVM) classifier	1.000	1.000	1.000

If we review a number of detection cases for knowingly malicious Android applications in Table 5, we can see that different families are detected by different atomic classifiers. This gives a better understanding of how malware is developed, deployed and of how it evolves.

For instance, test samples belonging to the *Trojan:Android/BaseBridge.C* family of malware were detected by atomic classifiers trained against F_{DEXP}, F_{DEXM}, F_{DEXF} and F_{DEXT} feature subsets. This indicates that this particular family uses similar code constructs but changes auxiliary files and resources.

Malicious APK samples belonging to the *Trojan:Android/SmsSend.OC* family were detected thanks to atomic classifiers associated with F_{DEXS}, F_{DEXC} and F_{MFS}. While examining samples belonging to this family it becomes apparent that they all require different combinations of system permissions, which causes a fluctuation in the confidence levels of F_{MFP}; for example,the mere presence of the permission SEND-SMS causes an increase in confidence of 0.005.

Finally, test malicious samples from *Trojan:Android/SmsSend.WB* family can be detected with the help of atomic classifiers based on F_{DEXT}, F_{DEXF}, F_{DEXS}, F_{DEXC}, F_{RSRC}, F_{HASH} and F_{DEXM}. This also provides the useful observation on how different could be malicious objects that belong to a same family, e.g. *SmsSend.OC* versus *SmsSend.WB*.

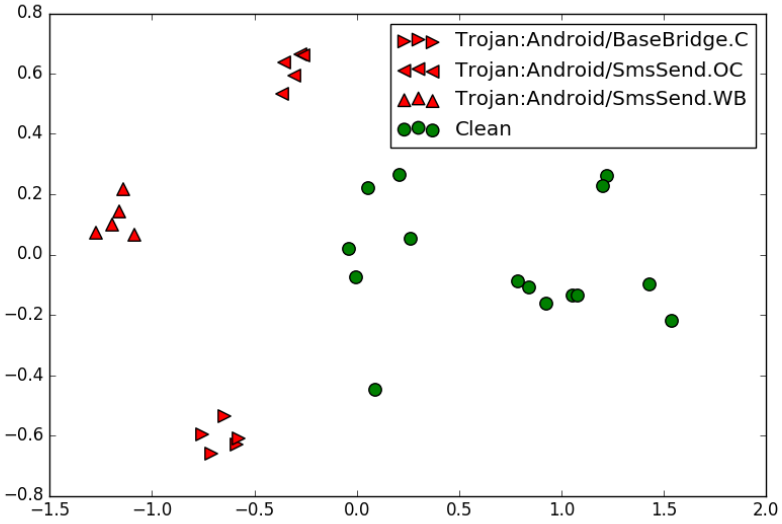

Fig. 5. PCA 2D visualization of a fraction of the test set provides good visibility on separation of some Android malware families (Table 5) and clean APKs.

As it can be seen, due to common values of confidences produced for various malicious families, the ensemble-based detection approach can be easily adopted for distance-based similarity visualisation. Although the thorough review of this particular property of the proposed decision scheme is out of scope of the paper, an example of Principal Component Analysis -based visualization for the data in presented in Figure 5 (data points jittering is used).

10 Conclusions

In this paper we have introduced several important requirements for the real-world deployment of automatic, machine learning-based Android malware detection systems.

First we highlight how the usage of fixed training and test data sets during development and testing of such systems is fraught with the risk of overestimating the systems' accuracy. To address this challenge we extend the traditional development process of such systems with an additional validation step, specifically adopting extra data sets representing the evolution of software applications during a continuous time interval following the completion of the systems' training.

Our paper also presents an improved approach for detecting new, previously unseen malicious Android applications. It consists of an ensemble of Normalized Bernoulli Naive Bayes [8] classifiers producing decisions that are consolidated by a single Support Vector Machine classifier. The proposed scheme is trained with a considerably sized, balanced data set received from a trustworthy source [2]. The validation of the classifier was carried out against a continuous stream of real-life new Android applications collected for a period of three months after the completion of the classifier's training. The results obtained clearly show that our improved approach outperforms previous research approaches [8] in terms of false positive rate. Our evaluation shows a FPR of 4.07% with a TPR of 90.82% on the evolving stream and with a TPR of 85.84% with a fixed FPR of 0.01% against the fixed dataset.

The ensemble-based nature of the proposed approach, where each member of the ensemble works on the basis of specific family of features, makes it suitable to counteract the evolution of malware, which happens almost exclusively in incremental steps. This means that some of the features extracted from malware will inevitably remain the same while only a subset change with any incremental releases.

Despite the noteworthy performances of our ensemble-based classifier, we need to acknowledge the difficult scenario in which these systems operate. For this reason we consider the proposed system suitable for fast and effective prioritization of samples for human analysis and more expensive automatic systems.

Acknowledgements

This work was supported by the Tekes (the Finnish Funding Agency for Innovation) project "Cloud-assisted Security Services" (3886/31/2016).

References

1. Net Applications: Operating system market share (2015) http://www.netmarketshare.com/operating-system-market-share.aspx.
2. F-Secure Corp.: Protect your life on every device (2015)
3. Google Inc.: Android core technologies – Android open source project. http://source.android.com.
4. ZD Net: Over 400 instances of dresscode malware found on Google play store, say researchers. http://www.zdnet.com/article/over-400-instances-of-dresscode-malware-found-on-google-play-store-say-researchers/.
5. Ars Technica: "godless" apps, some found in Google play, can root 90% of Android phones http://arstechnica.com/security/2016/06/godless-apps-some-found-in-google-play-root-90-of-android-phones/.
6. Bleeping computer: Adware found in Android app with over one million installs on Google play store https://www.bleepingcomputer.com/news/security/adware-found-in-android-app-with-over-one-million-installs-on-google-play-store/.
7. The Hacker News: Nasty Android malware that infected millions returns to Google play store http://thehackernews.com/2017/01/hummingbad-android-malware.html.
8. Sayfullina, L., Eirola, E., Komashinsky, D., Palumbo, P., Miche, Y., Lendasse, A., Karhunen, J.: Efficient detection of zero-day Android malware using normalized bernoulli naive bayes. In: IEEE International Conference on Trust, Security and Privacy in Computing and Communications. (2015)
9. Aafer, Y., Du, W., Yin, H.: DroidAPIMiner: Mining API-level features for robust malware detection in Android. In: Security and Privacy in Communication Networks. Volume 127. (2013) 86–103
10. Peiravian, N.: Data mining heuristic-based malware detection for Android applications. Master's thesis, Florida Atlantic University (2003)
11. Sahs, J., Khan, L.: A machine learning approach to Android malware detection. In: Intelligence and Security Informatics Conference (EISIC), 2012 European. (Aug 2012) 141–147
12. Yang, C., Xu, Z., Gu, G., Yegneswaran, V., Porras, P.: DroidMiner: Automated mining and characterization of fine-grained malicious behaviors in Android applications. In: Computer Security – ESORICS 2014. (2014) 163–182
13. Aung, Z., Zaw, W.: Permission-based Android malware detection. International journal of scientific and technology research (2013) 228–234
14. Burguera, I., Zurutuza, U., Nadjm-Tehrani, S.: Crowdroid: Behavior-based malware detection system for Android. In: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices. (2011) 15–26
15. Chen, K., Wang, P., Lee, Y., Wang, X., Zhang, N., Huang, H., Zou, W., Liu, P.: Finding unknown malice in 10 seconds: Mass vetting for new threats at the Google-play scale. In: Proceedings of the 24th USENIX Conference on Security Symposium. SEC'15 (2015) 659–674

16. Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y.: Machine learning-based malware detection for Android applications: History matters! (2014)
17. Gascon, H., Yamaguchi, F., Arp, D., Rieck, K.: Structural detection of Android malware using embedded call graphs. In: Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security. (2013) 45–54
18. Huang, C.Y., Tsai, Y.T., Hsu, C.H.: Performance evaluation on permission-based detection for Android malware. In: Advances in Intelligent Systems and Applications – Volume 2. (2013) 111–120
19. Yerima, S.Y., Sezer, S., McWilliams, G., Muttik, I.: A new Android malware detection approach using bayesian classification. In: Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on. (March 2013) 121–128
20. Suarez-Tangil, G., Tapiador, J.E., Peris-Lopez, P., Blasco, J.: Dendroid: A text mining approach to analyzing and classifying code structures in Android malware families. Expert Systems with Applications **41** (2014) 1104 – 1117
21. Desnos, A., Gueguen, G.: Androguard (2015) https://github.com/androguard/androguard.
22. Tumbleson, C., Winiewski, R.: Apktool – a tool for reverse engineering Android apk files. (2015) http://ibotpeaches.github.io/Apktool/.
23. Smali team: smali – an assembler/disassembler for Android's dex format. (2015)
24. Seneviratne, S., Seneviratne, A., Kaafar, M.A., Mahanti, A., Mohapatra, P.: Early detection of spam mobile apps. In: Proceedings of the 24th International Conference on World Wide Web. WWW '15 (2015) 949–959
25. Virus Total team: Virustotal – free online virus, malware and url scanner. (2015) https://www.virustotal.com/.
26. Murphy, K.P.: Machine Learning: A Probabilistic Perspective. The MIT Press (2012)
27. Zhou, Z.H.: Ensemble Methods: Foundations and Algorithms. Chapman & Hall/CRC (2012)
28. Cortes, C., Vapnik, V.: Support-vector networks. Machine Learning **20** (1995) 273–297
29. Chang, C.C., Lin, C.J.: LIBSVM: A library for support vector machines. ACM Trans. Intell. Syst. Technol. (2011) 27:1–27:27
30. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12** (2011) 2825–2830
31. Laskov, P., Gehl, C., Krüger, S., Müller, K.R.: Incremental support vector learning: Analysis, implementation and applications. Journal of machine learning research **7**(Sep) (2006) 1909–1936
32. Yu, R.: Android packers: Facing the challenges, building solutions. In: Virus Bulletin 2014 Proceedings. (2014) 266–275