Android Malware Detection: Building Useful Representations

Luiza Sayfullina*, Emil Eirola†, Dmitry Komashinsky‡, Paolo Palumbo‡ and Juha Karhunen*
* *Aalto University, Espoo, Finland, Email: name.lastname@aalto.fi*
†*Arcada Univerity of Applied Sciences, Helsinki, Finland, Email: emil.eirola@arcada.fi*
‡*F-Secure Corporation, Helsinki, Finland, Email: name.lastname@fsecure.com*

***Abstract*—The problem of proactively detecting Android Malware has proven to be a challenging one. The challenges stem from a variety of issues, but recent literature has shown that this task is hard to solve with high accuracy when only a restricted set of features, like permissions or similar fixed sets of features, are used. The opposite approach of including all available features is also problematic, as it causes the features space to grow beyond reasonable size.**

In this paper we focus on finding an efficient way to select a representative feature space, preserving its discriminative power on unseen data. We go beyond traditional approaches like Principal Component Analysis, which is too heavy for large-scale problems with millions of features.

In particular we show that many feature groups that can be extracted from Android application packages, like features extracted from the manifest file or strings extracted from the Dalvik Executable (DEX), should be filtered and used in classification separately. Our proposed dimensionality reduction scheme is applied to each group separately and consists of raw string preprocessing, feature selection via log-odds and finally applying random projections.

With the size of the feature space growing exponentially as a function of the training set's size, our approach drastically decreases the size of the feature space of several orders of magnitude; this in turn allows accurate classification to become possible in a real world scenario. After reducing the dimensionality we use the feature groups in a light-weight ensemble of logistic classifiers.

We evaluated the proposed classification scheme on real malware data provided by the antivirus vendor and achieved state-of-the-art 88.24% true positive and reasonably low 0.04% false positive rates with a significantly compressed feature space on a balanced test set of 10,000 samples.

***Keywords*-malware classification; logistic regression; random projection; Android; dimensionality reduction; feature selection**

I. Introduction

Nowadays mobile devices are used for managing our everyday needs beyond calls and sms-messages, as evident in the use of mobile banking, shopping, and social network applications. Anti-virus companies aim to stop hackers from making a profit on the Android platform by detecting malware early.

The analysis of file contents without its execution is known as static analysis. Due to the structure of the content of Android applications, most of the features differ from file to file, making it hard to come up with a fixed set of features. Permissions requested by installed application are a well-investigated set of features and have been used previously by researchers, for example in [1], but are limited in their discriminative power. Bag-of-words models with N-gram string features are used in [2], while byte code feature are used in [3]; these approaches, where the dimensionality of the feature set is very large and difficult to handle due to unrestricted selection of features, go to the other extreme when compared to the permission approach.

In this paper we propose a fast and accurate way of generating useful representations from Android application packages, that achieves the right balance between selecting a fixed set of features and extracting all possible features automatically. While this framework is adjusted for the Android malware problem, it can be adapted for other sparse, high dimensional problems.

Our approach contributes the following:

1) from the point of view of malware detection: presenting a scheme for selecting important feature groups and effectively filtering them by their discriminative power using log-odds; finding the trade-off in reducing dimensionality for different groups of features separately; achieving significantly low false positive rate to reduce the number of clean applications to be classified as malicious.
2) from the point of view of machine learning: showing that sparse random projections can efficiently reduce large feature spaces from millions to thousands of features to build a robust and reliable ensemble of logistic classifiers for Android malware detection;

The rest of the paper is structured as follows. Section II outlines publications that are relevant to our paper. Next we describe in detail the features being extracted from Android packages for classification. Section IV describes our dimensionality reduction approach and justifies the use of log-odds and random projections. Section V describes how we integrate our approach with feature preprocessing and an ensemble of logistic classifiers. We evaluate our approach in Section VI on the real data provided by the antivirus vendor.

II. Related Literature Review

The approach described in this paper uses static techniques to distinguish between malware and benign objects, meaning that the object under analysis is never run; this is in

contrast with dynamic approaches, where the object under investigation is run and its behavior analysed.

Machine Learning approaches related to Android malware vary greatly in the features selected: manually collected features [4], N-grams on the code strings [2], or specific groups of Android features, like permissions or manifest strings. The authors of [2] also used unreferenced strings extracted from DEX files with Support Vector Machines (SVM) [5] on 3-grams. Their accuracy on their dataset is 99.2% with slightly high false positive (FP) rate of 2%.

We consider [6], where random projections were used to reduce the feature space (sparse binary features, API trigrams and API calls) to classify Windows malware on a dataset of several million samples, to be the highest-impact contribution to the dimensionality reduction problem in malware classification. Although their work is not directly dealing with Android malware, we consider this publication to be very relevant due to its tackling a similar large-scale classification problem.

The dimensionality reduction problem for Android malware was tackled as well in [3] by exploiting the sparse nature of n-gram frequency matrices. This allows to efficiently compute partial singular value decomposition and thus make PCA possible. However, their dataset consisted of only 3869 Android applications, and with more files even optimized PCA could be hard to fit into system resources. With 20% of the files belonging to the test set, the proposed algorithm achieved a FP rate of 2% and TP rate of 91%.

While reviewing malware detection approaches relying on static analysis, the lowest reported FP rate was shown in [2], where the authors achieved FP = 0.06% and TP = 79.0% on byte-code features with a SVM classifier. In our experiments we achieve much higher TP rate with a fixed FP = 0.04%.

III. Used features

When designing malware detection systems for Android applications one is confronted with a significant freedom when it comes to the choices of feature; this is both a curse and a blessing. An Android Application package (APK) is an archive that in most cases contains 3 important files: *AndroidManifest.xml*, *classes.dex* and *resources.arsc*. This package is used for installing the application.

The first file *AndroidManifest.xml* [7] contains general information about the application to be installed. It provides the summary of the resources, the permissions needed to run the application and its components, including services, activities, libraries, entry points, their capabilities, etc. As most of the information is optional, one can expect variety in the strings observed across different manifest files. Researchers have actively used the permissions for Malware Classification as highly interpretable features.

When it comes to the executable part of the application, the DEX file is responsible for storing the compiled Java byte-code of the application. DEX strings include identifiers from the Java code, specifically: classes, strings, methods, types, prototypes, etc. In fact this group of features contributes the most to the explosion in the number of features. At the same time this group can not be entirely omitted due to the valuable information about the functionality of the application. In order to use the identifiers from DEX we had to implement our own extraction tool for the required strings.

The third file *resources.arsc* consists of the precompiled resources in a binary format. It may include images, icons, strings, or other data used by the application. Our initial study showed that features extracted from these files were usually less effective in distinguishing between malware and non malware; therefore, we don't consider this features any further.

Additionally, one can use information about other files present in the APK; for example, for each of them one could record the file names of embedded files, together with the cryptographic hashes of their content. These features will be considered in the remainder of this document as they add value to classifying repackaged files, for example.

From Table I we can see how large the raw feature spaces are, especially for DEX strings. Additionally, when training a system for Android malware classification, one would be encouraged to use all the available samples for training the model, which could contain millions of malicious files. Classification on this scale would be quite difficult to handle.

For our classification approach we use the most significant feature groups: hashes (HASH), manifest strings (MSTR), DEX strings (DEXS) and permissions from manifest files (PERM).

IV. Proposed Dimensionality Reduction

In this section we describe two subsequent dimensionality reduction (DR) techniques that are then applied to the selected string features in sequence. As classical techniques such as PCA [8], non-negative matrix factorization [9] or autoencoders [10] are rather of high-complexity for datasets with millions of features, we proposed alternative ways, that besides their speed show high performance.

A. Log ratio

The chance of a feature to contribute to the maliciousness of a file is estimated as follows:

$$\theta_i = \frac{p(mal)}{p(ben)}\,, \tag{1}$$

where $p(mal)$ and $p(ben)$ are the probabilities of feature i to occur in malicious and respectively benign files.

The probability of a feature appearing in the malicious class is defined as:

$$p(mal) = \frac{mal + k}{M + 2k}\,, \tag{2}$$

Table I: The size of the sets of raw extracted features from different groups. With the growth of the dataset, the feature space reaches millions of features for DEXS and HASH feature types. **Nmal** and **Nclean** denote the number of malicious and clean files respectively.

Feature type	**Source**	**Nmal=1000**	**Nclean=1000**	**Nmal=10000**	**Nclean=10000**
Manifest strings (MSTR)	AndroidManifest.xml	73,292	113,106	742,335	1,093,489
Permissions (PERM)	AndroidManifest.xml	13,590	11,457	137,776	111,548
Dex strings (DEXS)	classes.dex	4,362,573	9,960,902	44,155,410	99,268,349
Hashes (HASH)	hashes of all seen files	133,683	431,243	1,370,128	4,490,554

where k is a smoothing parameter used to avoid zero probabilities, mal is the number of malicious files having the feature, and M is the total number of malicious files observed. The same logic applies to the benign class.

Thus:

$$\theta_i = \frac{mal + k}{ben + k} \cdot \frac{B + 2k}{M + 2k}, \tag{3}$$

where B and M are the number of benign and malicious files respectively.

If we want to select the features being present only in the malicious class (i.e., $ben = 0$), we can achieve the following approximation when k is small and the data is balanced for simplicity, meaning that $B = M$:

$$\log(\theta_i) = \log(mal+k) - \log(ben+k) \approx \log(mal) - \log(k) \tag{4}$$

Hence the feature occurs one or more times in a malicious file when $\log(\theta_i) \geq \log(1) - \log(k) \geq -\log(k)$. Accordingly to have the feature only being present in benign files the following inequality should hold: $\log(\theta_i) \leq \log(k)$. Both inequalities hold when a feature occurs in either of the classes and $|\log(\theta_i)| \geq -\log(k)$.

Therefore, the features with $|\log(\theta_i)| < -\log(k)$ contribute both to malicious and benign files. To select some portion of such features in addition to purely discriminative features, we take features with $|\log(\theta)| > -\log(ck)$, where $c < \frac{1}{k}$ and c should be adjusted depending on how we want to squeeze the feature space.

In Figure 1 we plot the $\log\theta$ histogram for the DEXS features extracted from 1000 samples from malicious and benign classes. Only the most discriminative features with $|\log\theta| > -\log(ck)$ were selected.

B. Random Projection

For a given matrix $A \in \mathbb{R}^{N \times D}$ a random projection [11] defines a transformation to a lower dimensional space by multiplying by a randomly generated matrix $R \in \mathbb{R}^{D \times K}$, namely:

$$B = A \cdot R. \tag{5}$$

The resulting matrix B preserves all pairwise distances of A provided that R consists of i.i.d. entries with mean $\mu = 0$ and constant variance σ [12]. Non-zero entries of the matrix

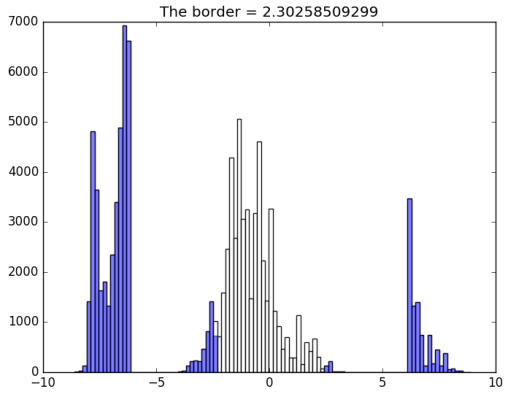

Figure 1: The histogram of DEXS feature log-odds for 1000 malicious and 1000 benign examples. The border is 2.302, c = 10 and k = 0.01 and only blue features are selected. Through the c parameter, we also use features from the middle part of the histogram, where the features contribute to both classes. Two high peaks on both sides are $-\log(k)$ and $\log(k)$, where the feature occurs once in malicious (benign) classes.

in a very sparse setting as introduced in [13] are generated with probability $\frac{1}{s}$. Initialization of the random matrix R can be expressed as follows:

$$R = \sqrt{s} \begin{cases} 1 & \text{with probability } \frac{1}{2s} \\ 0 & \text{with probability } 1 - \frac{1}{s} \\ -1 & \text{with probability } \frac{1}{2s} \end{cases}. \tag{6}$$

The density of the generated matrix is controlled by parameter s, that is shown to preserve the distances well when $s << \sqrt{D}$.

As this type of projection is capable of preserving similarity between the objects very efficiently, we used it as the main approach for feature compression. In comparison to Random Projections, Principal Component Analysis (PCA) has much higher complexity $O(D^2N + D^3)$.

Taking into account that matrix A consists of binary and sparse features, our transformation can be calculated even

faster, depending on the implementation used.

In fact the projected matrices B can differ significantly, taking into account that we need to select only a small fraction $K \approx \sqrt{D}$ and some of the features will be taken into account less than the others. However the variation of the Area Under the ROC Curve (AUC) [14] results becomes smaller with different RP initializations as K increases, which can be seen in Table II. We checked the performance of the logistic regression separately on DEXS features with 10 repetitions.

Table II: The effect of random projection initialization on DEX strings on the AUC. With the increase of K the deviation becomes smaller. $\mathbf{Ntrain} = \mathbf{Ntest} = 10,000$ and 10 replications are done. In other words, one does not need to try different RP initializations and find the best one; with sufficiently enough chosen features, the classification performance varies much less.

K	**AUC**
2,000	0.9913 ± 0.00052
4,000	0.9935 ± 0.00032
10,000	0.9944 ± 0.00027

V. Proposed approach

Below we describe all the details necessary to implement the proposed approach.

A. File parsing

At this stage for each file we collect the most significant feature groups: HASH, MSTR, DEXS and PERM. Only DEXS go through a preprocessing function that splits long strings according to predetermined delimiters and gets rid of irrelevant symbols. As other features are more standardized in form, like manifest strings, we do not preprocess them not to loose important information. DEXS contain strings from the code that can vary significantly.

As all strings from different files are preprocessed in the same way, this approach keeps the same frequencies for initially the same features. At the same time, features having a small difference, might end up producing the same features after the preprocessing. For instance, "SIM_CARD" and "sim card" form the same set of features after preprocessing: "sim","card".

Below you can find three steps we use for the feature preprocessing:

1) First remove the following symbols from the string: <> () ; ! : = ? * +
2) Recursively split the string according to the following delimiters: /n /t _$,&.
3) Split each string from step 2 if the string is in camel case: camelCase = (camel,Case)
4) Lowercase all the resulting strings

We have used the most common symbols in steps 1 and 2, the list of delimiter-symbols or symbols for removal could be extended. The features in each group should then be sorted by binary search during the training phase to speed up the intersecting of selected features and features from each file.

B. Feature Selection

After the preprocessing, we select features based on what was extracted from the training set.

From each file only the features with minimum number of feature occurrences α in the training dataset and minimum number of readable symbols β are extracted.

Secondly, the features are filtered based on their discriminative power calculated by $\log \theta_i$ of each feature in the training set. Specifically, we take HASH and DEXS features where $|\log \theta_i| > -\log ck$ for chosen c and k. Filtering $MSTR$ can be done optionally in case the extracted feature space is too large to work with. As PERM is a much smaller group, one does not need to filter them by log-odds. We will report our choices of c and k in Experimental Results Section VI.

C. Dimensionality Reduction

When the set of features is fixed, we build a bag-of-words model for the rest of the training set, where each file is represented by the binary occurrence of the features. Using only binary features optimizes storage costs and does not have a strong effect on the output, as most of the features will have 0 or 1 frequency in a file. For each i^{th} group of features the training matrix $X_i \in \mathbb{R}^{N \times D_i}$ is built.

Now with the help of random projection matrix $R_i \in \mathbb{R}^{D_i \times K_i}$ the projected matrix X_i^{rp} will be calculated. The size of the projected dimension K_i for each group of features should be decided based on the computational limits. In practice, we did not apply RP both to permissions and manifest strings, as the dimension D_i for them was rather reasonable (<10,000) in comparison to the millions of strings from DEX strings.

We have to stress that our DR is so efficient because cross-validation of the random projection based on its performance with the logistic classifier is not needed. We have previously conducted (Table II) a series of experiments on separate logistic regression classifiers trained on different initializations and showed negligible variation in the outcome when it comes to AUC, FP and TP rate.

D. Building an ensemble

After dimensionality reduction with RP, compact data matrices X_i^{RP} are used for building our proposed ensemble classifier. It comprises four logistic regression [15] classifiers, from which individual votes are combined by an SVM [5] classifier in order to provide a final verdict about the class of the object.

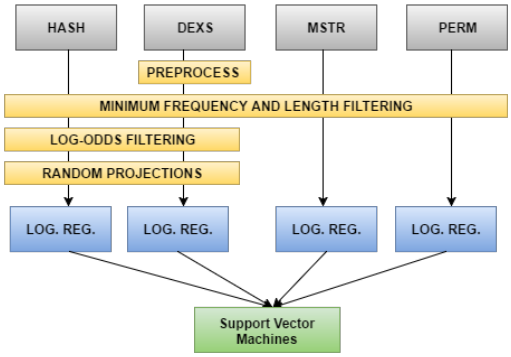

Figure 2: The schema of the proposed ensemble method. We don't apply log-odds and RP for PERM and MSTR, however with a larger training sample it could be helpful. Only DEXS features require preprocessing, as the others have relatively standard form.

Logistic regression is a generalized version of linear regression for binary outputs. However in logistic regression classifier linear combination of the explanatory variables X is found to approximate logistic function of the probability of the malicious class $p(mal)$ with vector of coefficients $\boldsymbol{\beta}$:

$$\mathbf{logit}(p(mal)) = \log \frac{p(mal)}{1 - p(mal)} = \boldsymbol{\beta} \cdot \boldsymbol{X} \qquad (7)$$

In our experiments the logistic regression classifier outperformed Naive Bayes, SVM and Decision Trees implementations from [16], verifying a high performance previously observed in [6]. The schema for our approach is shown in Figure 2.

VI. Experimental results

A. Used datasets

Benign Android samples were collected according to their observation date up to the end of 2014. Malicious samples are dated from the 1st of June, 2014 up to the 25th of October, 2014. We thank the antivirus vendor for providing us with the datasets.

The dataset used for training and validating the model consists of $\mathbf{Ntrain} = 20{,}000$ files equally from malicious and benign classes. Half of the files from $\mathbf{Ntrain}$ were used for feature selection with log-odds, and half for training the model. Test dataset has $\mathbf{Ntest} = 10{,}000$ samples, taken equally from both classes.

B. Chosen parameters

The preprocessing step was done with a minimum feature occurrence $\alpha = 3$ and a minimum number of readable symbols $\beta = 5$, based on the experiments in [17]. The attempts of trying to increase α decreased the performance, thus being fixed to rather low number. For large group of features, like DEXS and HASH $c = 10$ for log-odds; for MSTR $c = 100$ and k was set to 0.01 for all features. For MSTR $\log(ck) = 0$, meaning that log-odds was not used, as the size of its feature space was reasonable.

Logistic regression was cross-validated with $C \in \{0.5, 1.5\}$ with a step 0.05, and SVM with $C \in \{0.5, 1\}$ with a step 0.1. Other settings were set to default parameters as in scikit-learn python toolbox [16]. The features HASH and DEXS were reduced to dimensionality $K = \mathbf{Ntrain}$ by default, unless stated otherwise.

Table III: The comparison between the performances of the proposed approach against the Naive Bayes implementation, split by different sets of features. Using hashes in both cases increases recall and AUC. Our proposed approach significantly outperforms bag-of-words NB model, which is with much larger feature space. Alternatively, the best proposed approach with HASH achieves FP = 0.1% and TP = 96.16% with another border.

Method	**Feat.**	**AUC**	**TP**	**FP**
Proposed	DEXS, MSTR PERM, HASH	**0.9991**	**88.24%**	**0.04%**
NB	DEXS, MSTR PERM, HASH	0.9726	67.72%	0.04%
Proposed	DEXS, MSTR PERM	0.9986	86.92%	0.04%
NB	DEXS, MSTR PERM	0.9726	67.28%	0.04%

C. Evaluation

One important aspect when evaluating malware detection systems is the FP rate; a low FP rate means that benign applications will rarely be blocked. At the same time overall performance can be assessed based on Area Under the ROC Curve (AUC) [14], that takes into account the area under the curve with all FP and TP borders. We will report both AUC and FP, TP values. As very low FP rate is very important in practice, we fix FP to 0.04% and report corresponding TP rate. We have chosen FP = 0.04% as it was the lowest non-zero FP that compared NB classifier produced and to make a fair comparison with the ensemble of classifiers, we have chosen the same FP rate.

In Table III we show those measures after running our approach on the $\mathbf{Ntrain} = 20{,}000$ and $\mathbf{Ntest} = 10{,}000$ datasets. Adding hashes as a feature group slightly enhanced the performances. In addition to the proposed approach, we have implemented a Naive Bayes (NB) approach for comparison, where the features are only filtered by $\alpha = 3$

and $\beta = 5$ without log-odds filtering and RP. As Naive Bayes classifier has a low-complexity, we can use larger feature space with it and check, if using more features helps to improve the results. The estimates for the NB method used both Laplace smoothing parameter $k = 0.0001$ and normalization as proposed in [17]. From Table III we can see that our approach with reduced feature space significantly outperforms Naive Bayes, which uses the extensive feature space. We have to note that the NB classifier needs many more samples and features to increase its performance as its probability estimates become more accurate with the growth of number of samples [8].

VII. Conclusions

In this paper we suggest an efficient approach for dimensionality reduction for the proactive detection of Android malware. Specifically, we proposed an efficient way to extract meaningful representations from four highly significant Android feature groups and used those representations with an ensemble of logistic classifiers to achieve significant performances.

The dimensionality within large groups (HASH and DEXS) was reduced significantly: from 9.9 million to 10,000 by an initial preprocessing step followed by two low-computation dimensionality reduction methods: log-odds and random projections. Other groups of features, strings and permissions from manifest file form much smaller feature space in comparison and filtering by the minimum feature occurrence and number of readable symbols was the only technique we used. If necessary, log-odds and random projections can be applied to any group of features with their own parameters.

The evaluation of the proposed dimensionality reduction approach on the dataset of 20,000 training and 10,000 test samples with trustworthy labels showed an improved TP rate of 88.24% with an acceptably low FP rate of 0.04%. To the best of our knowledge we are in line with state-of-art performance due to preserving quite a small FP rate.

We believe that due to the simplicity and relative efficiency of this approach, it can be adopted by practitioners in the security field.

References

[1] D. Arp, M. Spreitzenbarth, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of Android malware in your pocket," 2014.

[2] R. Killam, P. Cook, and N. Stakhanova, "Android malware classification through analysis of string literals," 2016, text Analytics for Cybersecurity and Online Safety (TA-COS).

[3] B. Wolfe, K. Elish, and D. Yao, "High precision screening for Android malware with dimensionality reduction," in *Machine Learning and Applications (ICMLA), 2014 13th International Conference on*, 2014, pp. 21–28.

[4] L. Apvrille and A. Apvrille, "Identifying unknown Android malware with feature extractions and classification techniques," in *Proceedings of the 2015 IEEE Trustcom/BigDataSE/ISPA - Volume 01*, ser. TRUSTCOM '15, 2015, pp. 182–189.

[5] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, pp. 273–297, 1995.

[6] G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu, "Large-scale malware classification using random projections and neural networks," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013, pp. 3422–3426.

[7] "App manifest." [Online]. Available: http://developer.android.com/guide/topics/manifest/manifest-intro.html

[8] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*, 2006.

[9] D. D. Lee and H. S. Seung, "Algorithms for non-negative matrix factorization," in *Advances in Neural Information Processing Systems 13*, T. K. Leen, T. G. Dietterich, and V. Tresp, Eds. MIT Press, 2001, pp. 556–562.

[10] I. Goodfellow, Y. Bengio, and A. Courville, "Deep learning," 2016, book in preparation for MIT Press. [Online]. Available: http://www.deeplearningbook.org

[11] D. Achlioptas, F. McSherry, and B. Schlkopf, "Sampling techniques for kernel methods," in *IN ANNUAL ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 14: PROCEEDINGS OF THE 2001 CONFERENCE*. MIT Press, 2001, pp. 335–342.

[12] D. Achlioptas, "Database-friendly random projections: Johnson-lindenstrauss with binary coins," *J. Comput. Syst. Sci.*, vol. 66, no. 4, pp. 671–687, Jun. 2003.

[13] P. Li, T. J. Hastie, and K. W. Church, "Very sparse random projections," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '06, 2006, pp. 287–296.

[14] T. Fawcett, "An introduction to ROC analysis," *Pattern Recogn. Lett.*, vol. 27, no. 8, pp. 861–874, Jun. 2006.

[15] P. McCullagh and J. A. Nelder, *Pattern Recognition and Machine Learning (Information Science and Statistics)*, 1989, vol. 37.

[16] F. Pedregosa, G. Varoquaux, A. Gramfort, Michel *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[17] L. Sayfullina, E. Eirola, D. Komashinsky, P. Palumbo, Y. Miche, A. Lendasse, and J. Karhunen, "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.