Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 173 (2020) 345–353

Procedia Computer Science

www.elsevier.com/locate/procedia

International Conference on *Smart Sustainable Intelligent Computing and Applications* under ICITETM2020

Android Malware Detection based on Vulnerable Feature Aggregation

Arindaam Roy[a,*], Divjeet Singh Jas[a], Gitanjali Jaggi[a], Kapil Sharma[a]

[a]*Department of Information Technology- Delhi Technological University, New Delhi, India*

Abstract

Android has paved the way for the smartphone revolution. With the ever-growing advancements in technology, there is an inherent increase in the user reliance upon mobile technologies and third-party applications for communication, banking, and commerce. Needless to say, this is accompanied by steady growth in the number of attack surfaces, giving rise to new and highly advanced malicious software. Traditional malware detection approaches have revolved around pattern-based detection, which can easily be deterred using zero-day attacks. In this paper, we present a novel feature-engineering technique for android malware detection using Machine Learning. We perform static analysis to map each Application Programming Interface call to certain features, which is later aggregated to find the frequency of occurrence per feature. We empirically evaluate our approach and its robustness on 972 obfuscated android applications and 1100 benign applications and achieve an ROC-AUC score of 98.87%. We also demonstrate the scalability of our model by reducing the feature set by 75.9% and achieving a comparable ROC-AUC score of 95.67%.

Keywords: Android; Malware; Machine Learning; Lightweight; Static Analysis; Obfuscation.

1. Introduction

Simplified access to portable devices like mobile phones have made them an indispensable tool in daily lives and have successfully replaced conventional methods of carrying out mundane routines like ordering food, transferring money, or booking transportation. This increase in services and utilities offered at the click of a button should be accompanied by a growth in the strategies adopted to defend the system and the data present on it against the proliferation of malware.

* Corresponding author.
E-mail address: arindaam_bt2k16@dtu.ac.in

1877-0509
10.1016/j.procs.2020.06.040

As per Nokia Threat Intelligence Report - 2019 [1], Android devices were most affected by malware infections with a percentage of 47.15% of the total cases; Windows and personal computers came next at 35.82%, followed by IoT devices with 16.17%.iPhones attributed for less than 1% of the total attacks. In fact, a 31% increase was seen in the number of attacks on Android devices from 2017 to 2018.

Advanced android malware like Monokle [2], that has the ability to self-sign trusted certificates to intercept encrypted SSL traffic, and record a phone's lock screen activity in order to obtain passcodes help conduct espionage. Another example is MobSTSPY [3], that masquerades as a legitimate application purporting to be things like flashlights, games, and work productivity tools. It steals user information by lifting data like user location, text messages, contact lists, call logs, and clipboard items; and is also capable of uploading files found on the device.

Wei, Xuetao, et al. [4] bring to light that android malware can even affect enterprises by causing losses in data integrity, enterprise data-and hence, competitional advantage and consequently, monetary loss. All the above data is supportive of the necessity of research in this area.

Traditional Static Analysis methods tend to fail in the detection of malicious software since android devices are usually resource-poor, which limits the analysis at instruction level. Applications may circumvent signature-based detection if they are new or undefined in the pre-existing database, or alter the structure of the code by mechanisms like obfuscation. Lastly, due to a very small difference in the permissions required by benign and malicious files, it is not a completely reliable solution. Machine learning solutions have been widely explored to solve the problem of Android Malware detection, however as Pendlebury, Feargus, et al. [5] mention, most of these approaches suffer from training on dataset having spatial and temporal bias.

In our research, we explore the Android API (Application programming interface) calls to extract features and further aggregate them to find the total frequency of each feature and represent them in a single tuple per Android Package Kit (APK) file. We use Non-negative Matrix Factorization(NMF), a powerful machine-learning technique for reducing the total number of features to make our model lightweight and scalable. We attempt to address the issue of temporal bias by validating our model on a dataset belonging to a future time-set, the details of which can be found in Section 5.1.

Our contribution can thus be summarised as follows-

- We propose a light-weight malware detection framework, that operates upon only 50 features.
- We address the issue of temporal bias that exists in most of the prevailing malware detection approaches using machine learning. We do this by training our model on the standard DREBIN [6] data set collected in the period of August 2010 to October 2012, and validating on a data-set that was collected in 2014.
- We empirically show the robustness of our approach by evaluating it on a dataset with obfuscated malware [7].
- We obtain the highest accuracy of 93.77% with Random Forest Classifier in case of non-reduced features, and 88.72% with SVM in case of reduced features on the android applications with obfuscation.

The rest of this paper is organized as follows: Section 2 surveys the research carried out in the field of malware detection. Section 3 overviews the background knowledge utilised in our experimentation. The detailed detection approach is provided in Section 4, while statistical results on the chosen dataset are provided in Section 5. We close our research with concluding remarks in Section 6.

2. Related Work

Previous research carried out in the field of Android Malware detection, corroborates our concern towards breach of important security pillars like data confidentiality, user privacy, and information availability. Arshad, Saba, et al. [8] divide current antimalware strategies into two categories, Static and Dynamic Approaches. As discussed, conventional static detection methods are limited by signature databases, complex reflective calls, and high reliance upon users and resources. A detailed classification of various real-life android malware detection systems based on analysis methods can be found in the research conducted by Muttoo et al. [9].

Over the years, the focus has shifted towards machine-learning-based detection systems. For example, Zarni Aung, Win Zaw [10] discuss the validation of these methods on traditional permission-based detection systems and encourages the development of similar solutions. Similarly, Droidminer [11] is a lightweight classifier that mines complex malware data patterns based on API Calls and the associated package-level information. It performs frequency analysis to capture the most relevant API calls that malware invokes, and evaluates them using different ML classifiers. Albeit, it does not include features of API calls such as the permissions needed to execute it. Additionally, a number of research studies have also incorporated topic modeling techniques to study the behaviour of an Android Application. Garg M et al. [12] implements the method by using Application permissions to identify outliers, that are potentially harmful to the system.

S. Hou et al. [13] proposed a deep belief network based Android malware detection system DroidDelver. They performs feature engineering on API calls by categorizing it into blocks and employs deep learning techniques to classify it.

Researchers have also performed investigation upon APK files by extracting permissions and API calls as a feature set. Chan et al. [14] obtain an accuracy rate of more than 90% using this method. This research has been extended by Peiravian et al. [15] to implement machine learning models with improved accuracy of 94.9%.

Over the last decade, an interesting shift has been seen towards behavioral semantics for identification of malicious software. While Zhang, Hanqing, et al. experiment with Association Rule Analysis [16], Zhu, Rui, et al. come up with a technique for investigation by generating call-graphs of the given application [17]. Another example is DaDiDroid [18], which constructs the weighted directed graphs of API Calls and utilises these features to obtain a 91% accuracy on an obfuscated dataset. These methods are however, limited by a high need for resources like space, time and computational power and can thus be transformed to a lighter and more robust model which is where our research bridges the gap.

3. Background

In this section, we present the pith of the Android Application architecture, which is significant for understanding the proposed feature engineering method. Then, we introduce the concept of Non-Negative Matrix Factorization and how it is utilised to make our approach lightweight.

3.1. Android Application

Primarily Java is used to write Android applications, which is then packaged and distributed as a compressed file called APK file. Each APK file is assisted by its resources, a manifest file, and the associated byte-code. The byte-code of an application compiled into Dalvik EXecutable(DEX) format, is executed on a Java VirtualMachine (JVM). classes.dex is the main DEX file of the app and can be disassembled into an intermediate representation format - smali code using reverse engineering tools. Each APK file also consists of various components of different types. These components lay the foundation of an APK. Every component is an entry point; that is, it gives a basis of interaction to a user or a system. Hence, it is essential to analyze the component API for security reasons. There are four different types of components in Android application files (APK files):

1. Activity: An activity is the entry point that allows the user to interact with the system. It represents a single screen through which actions are performed on the screen.
2. Service: A service is a component that enables the app to remain running in the background without an interface. It communicates with other apps from the background.
3. Broadcast receiver: It enables the delivery of events to the application by the system outside of a regular user flow. This feature lets the app capture messages which are meant for the entire system. The system can also deliver broadcasts to applications that are not currently running.
4. Content Provider: It manages a shared set of application data that can be stored on any storage location from were the application can access it. Storage location includes the file system, the web etc. Other applications can search or modify the data if it is allowed by the content provider.

All the components should be declared in the manifest file of the application before it can be used. intent and intent filters are used to establish communication between the components. An intent is a messaging object used to communicate with other application and request certain action from it. An ”intent-filter” is the expression declared in the manifest file of the app that specifies the type of intent that the component receives.

3.2. Non-Negative Matrix Factorization(NMF)

NMF or NNMF[19, 20] is a family of algorithms in linear algebra and multivariate analysis, that factorizes a matrix A into two matrices W and H, such that all three matrices have non-negative elements. That is,

$$A_{mXn} \approx W_{mXk} H_{kXn}, \tag{1}$$

where $k \ll min(m,n)$

is a non-negative integer, giving the rank of approximation of A.

Here, **W** is the application static feature matrix, and **H** is the weight matrix based upon the number of calls.

Thus, it is a dimensionality reduction technique in cases where the matrix values are non-negative and hence, easy to inspect. It is an important step in our approach, since it deals with a high and potentially sparse dimensional space and modifies it down to more significant features, hence saving computational time and power. It also deals with noise during processing, thus circumventing incorrect classification.

4. Proposed Approach

An overview of the proposed approach is shown in Fig. 1. Our proposal includes four main modules- Preprocessing, Feature Extraction, Feature Vector Generation and Reduction, and finally, Classification using machine learning.

4.1. Preprocessing: Unpacking and Decompilation

To analyze the APKs, it has to be first unzipped and then decompiled. As discussed before, each APK file contains the application byte-code, the required resources, and a Manifest.xml file, which contains information pertaining to security features like permissions, components, and so on.

The byte-code contains features like associations, instructions, and methods which can be extracted from the source codes that are present within the APK files as Dalvin Executable files, also called dex codes. Reverse engineering is performed by converting dex files to human-readable smali code by using Apktool [21], thus completing the decompilation process.

4.2. Feature Extraction

The Manifest file givers us security-sensitive information, but using these features alone gives us a result with high false positives. Therefore, we investigate the smali code generated in the prior step to get all the API Calls, and further reverse map the API Calls to their permissions and intents using PScout [22] and official Android Documentation [23]. The following features are extracted from the API calls:

- Function type: Android functions, also known as methods, may belong to one of the four categories:
  1. Android System API: As listed in Android Official References or Java libraries.
  2. Third-Party API: Widely-used, third-party platforms like Facebook, Twitter, Google, etc.
  3. Component: As declared in the Manifest.xml file
  4. Risky API: APIs which can be used for obfuscation, such as cryptographic modules, are labeled as risky
  5. Others
- Permissions:
  1. Required: Any special permissions that are required by the function to execute. These may include access to file storage, contacts, etc.
  2. Hardware: Access requests for hardware like Camera, Microphone, Location, etc. are considered as a feature.

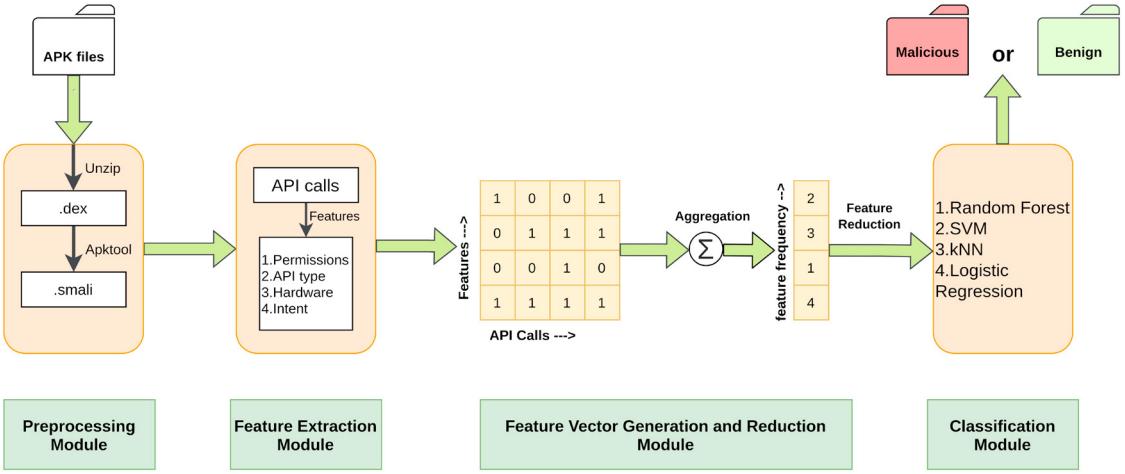

Fig. 1: Overview of proposed approach

3. Component: When the class containing the function requests for permission, all functions of the Component class have access to that resource. These are stored as component permission features.

- Intent filter: Intent filters can fetch activity information on the OS or other applications. These may also start an activity with specified intent.

4.3. Feature Vector Generation and Reduction

- Once the features of each API call is generated, we obtain a matrix with dimensions *(n* x *m)* where *'n'* represents the number of API Calls, and *'m'* represents the number of features extracted from it, as illustrated in (2). We represent the features in one-hot encoded fashion, which forms the tuple corresponding to each API. Thereafter, summation (4) is performed over columns for all these features to consolidate them into a single vector, wherein each value in the vector represents the frequency of that static feature with respect to the application under study. Thus, this module enables us to represent each application by a fixed size vector (3) which essentially becomes the feature set.

$$\chi = \begin{bmatrix} \vartheta_{1,1} & \vartheta_{1,2} & \cdots & \vartheta_{1,n} \\ \vartheta_{2,1} & \vartheta_{2,2} & \cdots & \vartheta_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ \vartheta_{m,1} & \vartheta_{m,2} & \cdots & \vartheta_{m,n} \end{bmatrix} \tag{2}$$

is aggregated to

$$\gamma = \begin{bmatrix} \ell_{1,1} & \ell_{1,2} & .. & .. & \ell_{1,m} \end{bmatrix} \tag{3}$$

where,

$$\ell_{1,m} = \sum_{i=1}^{i=n} \vartheta_{i,m} \tag{4}$$

- Feature reduction using NMF: Once the fixed-sized feature set is achieved, we utilize the NMF technique for achieving an alternate representation of the feature set, which is constructed in a lower-dimensional space, thus

Models	Accuracy(%)	Precision(%)	Recall(%)	F-Score(%)	ROC-AUC(%)
Logistic Regression	91.86	94.77	84.72	89.47	98.78
SVM	93.35	90.88	93.06	**91.95**	96.27
Random Forest	**93.77**	**99.80**	84.72	91.73	**98.87**
KNN	93.15	89.37	**94.44**	91.84	97.51

Table 1: Performance metrics of machine-learning classifiers on non-reduced features of obfuscated APKs

Models	Accuracy(%)	Precision(%)	Recall(%)	F-Score(%)	ROC-AUC(%)
Logistic Regression	76.15	68.22	77.78	72.68	77.67
SVM	**88.72**	89.52	**81.94**	**85.56**	95.34
Random Forest	80.32	**90.27**	59.72	71.88	95.62
KNN	87.84	89.83	79.17	84.16	**95.67**

Table 2: Performance metrics of machine-learning classifiers on reduced features of obfuscated APKs

permitting us to generate a reduced feature set while preserving the information that the original feature set encompassed. This makes our approach lightweight and enables us to run the module on low-end systems.

4.4. Classification

- Machine Learning for Classification: The feature set constructed from the previous step are fed into popular machine learning classifiers like Logistic Regression [24], K-Nearest Neighbours [25], Support Vector Machine [26], Random Forest [27] for malware detection.

5. Experimental Results

5.1. Dataset

In this paper, to train our model we use 1100 malware APKs collected between the year 2010 to 2012 as a part of the DREBIN dataset [6] and 1100 benign APKs given in "CICInvesAndMal2019" [28] dataset provided by the Canadian Institute of Cybersecurity. For validating our model we use 792 obfuscated malicious applications collected in 2014, as provided under "Android validation dataset" by the Canadian Institute of Cybersecurity [7] and 1100 benign APKs. Since the validation data-set was collected at a later point in time as compared to the training data set, we circumvent temporal bias that tends to prevail in machine learning based android detection approaches.

5.2. Performance Evaluation

We decompile our APK data with Apktool and parse the .smali files to extract the API calls. We reverse-map the API calls with its features by using the publicly available Pscout [22] data. We then find the frequency of each feature by summing over the features and preparing our data for the machine-learning classifiers. Furthermore, we also reduce the feature list using NMF and evaluate our models on the reduced data. The evaluation of the generated feature set is carried out with the help of multiple machine learning classifiers. This assessment is carried out on 2 different feature sets. In the first case, the classifiers are employed on only the 209-dimension frequency-based feature set.

Table 1 summarizes the performance metrics of all the employed models on obfuscated data. In this case, the best performance was achieved by Random Forest, which attains an accuracy of 93.77% along with the ROC-AUC value of 98.87%. In the second case, NMF was applied to the feature set to obtain a lower 50-dimension representation of the original feature set, enabling us to reduce the size of the feature set by 75.9%. The dimension was determined by iteratively testing performance of the Random Forest classifier with the reduced feature size. The accuracy achieved for the Random Forest classifier with respect to reduced dimension is shown in Figure 2. The effectiveness of this

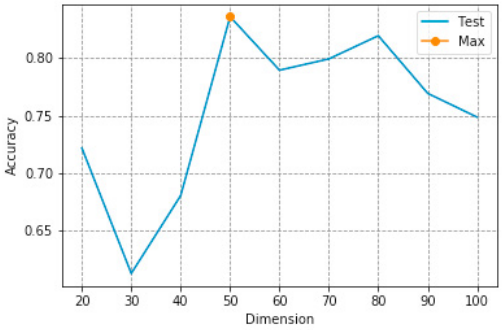

Fig. 2: Accuracy with respect to feature dimension of the validation data

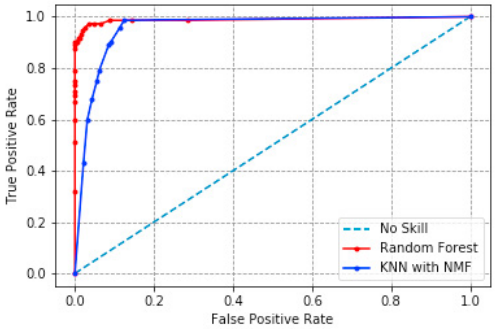

Fig. 3: ROC Curve of best Models of reduced and un-reduced features.

reduced feature set of 50 items was assessed with respect to the same classifiers and is summarized in Table 2. A comparative ROC curve has been plotted in Figure 3, which visually shows the difference in performance by Random Forest Classifier on un-reduced dataset and SVM on reduced dataset.

Our classification models and its tuned hyper-parameters are as follows:

1. ***Support Vector Classifier***: The kernel utilized for classification is a Radial Basis Function (RBF). The inverse of regularization strength parameter (C) is tested for multiple values, among which the best result is achieved for C as 500 for both non-reduced and reduced feature vector model.
2. ***Logistic Regression***: The hyper-parameter, l2, is used as the penalization norm for both the non-reduced feature model, and reduced feature model. The inverse regularization strength parameter C is set to 1 in both cases.
3. ***Random Forest***: We achieve the best results with the number of trees chosen as 40 for non-reduced model and 90 for reduced feature vector model. The splitting criteria utilized in both the cases is Gini impurity.
4. ***K Nearest Neighbour***: We use the Euclidean metric for the distance measure. Most importantly, the value of K for which optimal performance is observed is 29 for un-reduced data and 9 for reduced data.

5.3. Comparison with existing approaches

The comparison of the robustness of the various state of the art approach is summarized in Table 3. MaMaDroid [29] and DaDiDroid [18] used multiple obfuscation techniques such as identifier renaming, call indirection, packing on their APK dataset.

Approach	No. of apps	Accuracy	Precision	Recall	F-Score
MaMaDroid	44,000	73.5%	79.10%	27.1%	40.3%
DaDiDroid	63,693	91.2%	90.10%	82.7%	86.2%
Our best model (RF)	2,200	93.77%	99.80%	84.72%	91.73%

Table 3: Comparison with the state of the art.

6. Conclusion

This work firstly analysed API calls from the extracted smali codes for the construction of a frequency-based feature vector for each app. The effectiveness of this feature set is evaluated with the help of multiple machine learning classifiers, among which Random Forest showed the best performance achieving an ROC-AUC score of 98.87% on obfuscated malware. Additionally, NMF was utilized as a technique for obtaining a lower, 50-dimensional representation of the original feature set of 209 features. While the reduction introduced is of a substantial 75.9%, the performance observed did not experience a significant decline, with an ROC-AUC score of 95.67% with K Nearest Neighbour classifier. Such a scenario portrays the potential scalability of the proposed detection system. The above scores are achieved for obfuscated malware, hence proving its robustness.

References

[1] Nokia, "Nokia threat intelligence report," [online], 2019, https://pages.nokia.com/T003B6-Threat-Intelligence-Report-2019.html(visited on 01/24/2020).
[2] Lookout, "Monokle- the mobile surveillance tooling of the special technology center," [online], July 2019, https://www.lookout.com/documents/threat-reports/lookout-discovers-monokle-threat-report.pdf.
[3] Bala Sethunathan – Director Security Practice CISO, "Softwareone cyber threat bulletin," [online], September 2019, https://www.softwareone.com/-/media/global/insights/premium-downloads/softwareone-cyberthreatbulletin-2019-09.pdf.
[4] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Malicious android applications in the enterprise: What do they do and how do we fix it?" in *2012 IEEE 28th International Conference on Data Engineering Workshops*, April 2012, pp. 251–254.
[5] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "{TESSERACT}: Eliminating experimental bias in malware classification across space and time," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 729–746.
[6] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket." in *Ndss*, vol. 14, 2014, pp. 23–26.
[7] H. Gonzalez, N. Stakhanova, and A. A. Ghorbani, "Droidkin: Lightweight detection of android apps similarity," in *International Conference on Security and Privacy in Communication Networks*. Springer, 2014, pp. 436–453.
[8] S. Arshad, M. A. Shah, A. Khan, and M. Ahmed, "Android malware detection & protection: a survey," *International Journal of Advanced Computer Science and Applications*, vol. 7, no. 2, pp. 463–475, 2016.
[9] S. K. Muttoo and S. Badhani, "Android malware detection: state of the art," *International Journal of Information Technology*, vol. 9, no. 1, pp. 111–117, 2017.
[10] W. Z. Zarni Aung, "Permission-based android malware detection," *International Journal of Scientific & Technology Research*, vol. 2, no. 3, pp. 228–234, 2013.
[11] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *International conference on security and privacy in communication systems*. Springer, 2013, pp. 86–103.
[12] M. Garg, A. Monga, P. Bhatt, and A. Arora, "Android app behaviour classification using topic modeling techniques and outlier detection using app permissions," in *2016 Fourth International Conference on Parallel, Distributed and Grid Computing (PDGC)*, Dec 2016, pp. 500–506.
[13] S. Hou, A. Saas, Y. Ye, and L. Chen, "Droiddelver: An android malware detection system using deep belief network based on api call blocks," in *International Conference on Web-Age Information Management*. Springer, 2016, pp. 54–66.
[14] P. P. Chan and W.-K. Song, "Static detection of android malware by using permissions and api calls," in *2014 International Conference on Machine Learning and Cybernetics*, vol. 1. IEEE, 2014, pp. 82–87.
[15] N. Peiravian and X. Zhu, "Machine learning for android malware detection using permission and api calls," in *2013 IEEE 25th international conference on tools with artificial intelligence*. IEEE, 2013, pp. 300–305.
[16] H. Zhang, S. Luo, Y. Zhang, and L. Pan, "An efficient android malware detection system based on method-level behavioral semantic analysis," *IEEE Access*, vol. 7, pp. 69 246–69 256, 2019.
[17] R. Zhu, C. Li, D. Niu, H. Zhang, and H. Kinawi, "Android malware detection using large-scale network representation learning," *arXiv preprint arXiv:1806.04847*, 2018.
[18] M. Ikram, P. Beaume, and M. A. Kaafar, "Dadidroid: An obfuscation resilient tool for detecting android malware via weighted directed call graph modelling," 05 2019.

[19] D. D. Lee and H. S. Seung, "Algorithms for non-negative matrix factorization," in *Advances in neural information processing systems*, 2001, pp. 556–562.
[20] ——, "Learning the parts of objects by non-negative matrix factorization," *Nature*, vol. 401, no. 6755, pp. 788–791, 1999.
[21] R. Winsniewski, "Apktool: a tool for reverse engineering android apk files," 2012.
[22] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 217–228.
[23] Google, "Developer guides," [online], 2019, https://developer.android.com/guide/index.html (visited on 07/24/2020).
[24] A. Y. Ng and M. I. Jordan, "On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes," in *Advances in neural information processing systems*, 2002, pp. 841–848.
[25] J. M. Keller, M. R. Gray, and J. A. Givens, "A fuzzy k-nearest neighbor algorithm," *IEEE transactions on systems, man, and cybernetics*, no. 4, pp. 580–585, 1985.
[26] C. J. Burges, "A tutorial on support vector machines for pattern recognition," *Data mining and knowledge discovery*, vol. 2, no. 2, pp. 121–167, 1998.
[27] A. Liaw, M. Wiener *et al.*, "Classification and regression by randomforest," *R news*, vol. 2, no. 3, pp. 18–22, 2002.
[28] L. Taheri, A. F. A. Kadir, and A. H. Lashkari, "Extensible android malware detection and family classification using network-flows and api-calls," in *2019 International Carnahan Conference on Security Technology (ICCST)*. IEEE, 2019, pp. 1–8.
[29] L. Onwuzurike, E. Mariconti, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini, "Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version)," *ACM Transactions on Privacy and Security*, vol. 22, 11 2017.