Machine learning approach to detect malicious mobile apps

Hassan Kazemian

[1] Intelligent Systems Research Centre
School of Computing and Digital Media
London Metropolitan University
London, UK
h.kazemian@londonmet.ac.uk

Abstract. Malicious developers are developing unsafe mobile apps which puts users at risk of exposing their personal data in unsafe hands. They are using techniques that change over time and their intention is to bypass the detector systems which are mostly rule- based. This paper avoids the limitations of rule-based systems by building a novel malware detector that can detect malicious apps by making use of machine learning techniques primarily focusing on deep neural networks i.e. deep multi-layer perceptron. These techniques have various properties that can adapt and identify various types of malicious applications. Simulation results on various datasets demonstrate clear superiority of this detector over other approaches, as this approach achieves 99% accuracy. Also, the detector is efficient enough to detect within 100 milliseconds or less due to the intelligent use of autoencoder which reduces the dimensions in the feature.

Keywords: Deep neural network, malicious mobile apps, android.

1 Introduction

The influx of range of applications in mobile environments have outpaced the desktop in the last few years. Users now seamlessly install apps with a click of a button and start using them. On average a user utilizes diverse range of apps regularly on daily basis to carry out various activities. For example, users check the timetable of their transport to plan their routes to various places. They keep track of their finances by checking their bank balances and financial transactions from their app. The most used apps are messaging based which allow sending text and multimedia messages to their friends, colleagues and families. Social apps allow users to connect to their nearest ones. Shopping apps allow users to place orders to have their items delivered to their homes. Looking at these various types of apps, they have access to messages, phone calls, data usages. Some of this information are sensitive & private and some of these provide premium services which cost the users money. Before installing an app, the apps ask for various permissions at the beginning of the installation. This allows the user to make a decision whether to allow the app to access permissions. Most users ignore them before the installation. This essentially creates a loophole. A user can install malicious application

without proper checks. Some of the example apps that have access to user per app are listed in Table 1.

Table 1. Android apps with various uses and permissions

Application	Use	Permissions
WhatsApp	Messaging	Photos, Contacts
Skype	Video Conference	Phone Calls, Contacts
Messenger	Messaging	Photos, Phone Calls, Contacts
HSBC	Check Bank Balance	N/A
Google Maps	Plan routes	Location
Gmail	Read Emails	Contacts, Photos

Rule-based systems can detect known malware and look for traits and characteristics that exist in malicious applications. The performances of these detectors are very high and prevent applications being installed on a user's phone. Malicious developer can create a changed version with new characteristics to stop being detected. The old rules in the detector will not highlight this new app as malicious. Thus, apps which are updated regularly will avoid detection.

Traditional machine learning techniques such as naïve Bayes, decision trees, support vector machine & boosting have been used in [1] to detect malicious executables. They either look at the source code or the behavior of apps. These involve looking at consecutive bytes in executables stored on disk or in memory and creating features from these bytes which are fed into the classifiers. The work has achieved reasonable performance but faced a large feature space and had to maintain many of these features. In [2] these two issues are resolve by compressing the feature space using various hashing techniques. Surprisingly, the same level of performance was retained. Rather than reading the binaries directly, in [3, 4, 5], static flow analysis is used instead, whereas static function analysis is used in [6, 7]. Another work performs static code analysis on source code. It finds the relations between function and permissions. The work focuses on deep learning techniques which cater for more combinations and sophisticated data. Some techniques look at the behavior of apps and users. The systems look at the traces of application uses by users and monitor the usage in real time. These papers [8, 9] look at the patterns in the users.

Most of the executions take place within the mobile device where there are limits on computation, storage and battery life. Some experiments take place on remote servers, which mimic the mobile environment within hostile environments [10, 11, 12]. Although this allows for more exploits to be tested, it overlooks the limitations imposed on mobile devices.

Generative and discriminative classification techniques impact on how to solve a classification problem. Some find discriminative techniques such as support vector machine is useful in text and image categorization [13, 14, 15]. These essentially finds the boundary between the classes. On the other hand, generative techniques such as naïve Bayes are even more useful. Generative classifiers have been used in spam classification and find the distribution of each class.

Many applications request permissions from users that apps do not use at all. In [18, 19] their analysis reveals find many apps request permission that can be deemed harmful and risky to normal users. Permissions provide access to various actions and in many cases one permission give access to multiple actions [20, 21, 22]. In this scenario, the user is giving away more control to the app than necessary. For example, a particular permission gives the app access multiple API calls. Some of these calls are necessary and others can be mishandled and expose the user's private information such as messages, photos, call history, etc. which the app may not necessarily need but can exploit to carry out promotional activities.

Area Under ROC Curve (AUC) is a popular metric to verify the performance of a classifier. An area close to 1 indicates a good classifier but with a highly imbalanced dataset, this metric can be biased and may be biased to a particular class. To overcome this problem, some suggest using F1 as the metric as a more reliable metric to determine the performance of imbalanced dataset. In the paper, to avoid the problem of overfitting to a particular class, k-fold cross validation is selected where value of k is 5. This sets the training dataset to 80% and test dataset to 20% for each of the 5 iterations. This ensures that the created model is generalized and is adaptable in all conditions. The research also looks at the model error and loss as the model across various epochs is trained. AUC is a performance metric for binary classification problems. The AUC represents a model's ability to discriminate between positive and negative classes. An area of 1.0 represents a model that made all predictions perfectly. An area of 0.5 represents a model that is as good as random. ROC can be broken down into sensitivity and specificity [23, 24]. A binary classification problem is really a trade-off between sensitivity and specificity. Sensitivity is the true positive rate also called the recall. It is the number of instances from the positive (first) class that actually predicted correctly. Specificity is also called the true negative rate. It is the number of instances from the negative (second) class that were actually predicted correctly. The confusion matrix is a handy presentation of the accuracy of a model with two or more classes. The table presents predictions on the x-axis and accuracy outcomes on the y-axis. The cells of the table are the number of predictions made by a machine learning algorithm. For example, a machine learning algorithm can predict 0 or 1 and each prediction may actually have been a 0 or 1. Predictions for 0 that were actually 0 appear in the cell for prediction = 0 and actual = 0, whereas predictions for 0 that were actually 1 appear in the cell for prediction = 0 and actual = 1.

Looking at the last few years, it has been seen that with larger datasets deep learning techniques outperform traditional machine learning techniques. The research work focuses on deep learning techniques and will improve the performance while gathering more data. A Perceptron is a single neuron model that was a precursor to larger neural networks. It is a field of study that investigates how simple models of biological brains can be used to solve difficult computational tasks like the predictive modeling tasks in machine learning. The goal is not to create realistic models of the brain, but instead to develop robust algorithms and data structures that can be used to model difficult problems. The power of neural networks come from their ability to learn the representation in your training data and how to best relate it to the output variable that you want to predict. In this sense neural networks learn a mapping. Mathematically, they are

capable of learning any mapping function and have been proven to be a universal approximation algorithm. The predictive capability of neural networks comes from the hierarchical or multilayered structure of the networks. The data structure can pick out (learn to represent) features at different scales or resolutions and combine them into higher-order features. For example from lines, to collections of lines to shapes.

2 Features used in simulation

Table 2. Top features for the top 5 families of malicious categories

Malware family	**Top 5 features**		
	Feature s	Feature set	Weight w_s
FakeInstaller	`sendSMS`	S_7 Suspicious API Call	1.12
	`SEND_SMS`	S_2 Requested permissions	0.84
	`android.hardware.telephony`	S_1 Hardware components	0.57
	`sendTextMessage`	S_5 Restricted API calls	0.52
	`READ_PHONE_STATE`	S_2 Requested permissions	0.50
DroidKungFu	`SIG_STR`	S_4 Filtered intents	2.02
	`system/bin/su`	S_7 Suspicious API calls	1.30
	`BATTERY_CHANGED_ACTION`	S_4 Filtered intents	1.26
	`READ_PHONE_STATE`	S_2 Requested permissions	0.54
	`getSubscriberId`	S_7 Suspicious API calls	0.49
GoldDream	`sendSMS`	S_7 Suspicious API calls	1.07
	`lebar.gicp.net`	S_8 Network addresses	0.93
	`DELETE_PACKAGES`	S_2 Requested permission	0.58
	`android.provider.Telephony.SMS_RECEIVED`	S_4 Filtered intents	0.56
	`getSubscriberId`	S_7 Suspicious API calls	0.53
GingerMaster	`USER_PRESENT`	S_4 Filtered intents	0.67
	`getSubscriberId`	S_7 Suspicious API calls	0.64
	`READ_PHONE_STATE`	S_2 Requested permissions	0.55
	`system/bin/su`	S_7 Suspicious API calls	0.44
	`HttpPost`	S_7 Suspicious API calls	0.38

The simulation uses the dataset prepared in [25] which contains 5,560 malware apps. Table 2 shows the top features in the dataset and Figure 1 shows the categories of malware. The figure shows Detection Rates against Malware Families. As the first step, a lightweight static analysis of a given Android application is performed. Although apparently straightforward, the static extraction of features needs to run in a constrained environment and complete in a timely manner. If the analysis takes too long, the user might skip the ongoing process and refuse the overall method. Accordingly, it becomes essential to select features which can be extracted efficiently. The paper thus focuses on the manifest and the disassembled dex code of the application, which both can be obtained by a linear sweep over the application's content. To allow for a generic and extensible analysis, all extracted features are represented as sets of strings, such as permissions, intents and API calls. In particular, the following 8 sets of strings are extracted.

Every application developed for Android must include a manifest file called AndroidManifest.xml which provides data supporting the installation and later execution of the application. The information stored in this file can be efficiently retrieved on the device using the Android Asset Packaging Tool that enables us to extract the sets of data. Also, android applications are developed in Java and compiled into optimized bytecode for the dalvik virtual machine. This bytecode can be efficiently disassembled and provides DREBIN with information about API calls and data used in an application.

To achieve a low run-time, a lightweight disassembler is implemented based on the dex libraries of the Android platform that can output all API calls and strings contained in an application. This information is used to construct the following feature sets.
S1 Hardware components: This first feature set contains requested hardware components. If an application re- quests access to the camera, touchscreen or the GPS module of the smartphone, these features need to be declared in the manifest file. Requesting access to specific hardware has clearly security implications, as the use of certain combinations of hardware often reflects harmful behavior. An application which has access to GPS and network modules is, for instance, able to collect location data and send it to an attacker over the network.

S2 Requested permissions: One of the most important security mechanisms introduced in Android is the per- mission system. Permissions are actively granted by the user at installation time and allow an application to access security-relevant resources. As shown by previous work [13], malicious software tends to request certain permissions more often than innocuous applications. For example, a great percentage of current malware sends premium SMS messages and thus re- quests the SEND SMS permission. Thus, all permissions listed in the manifest are gathered in a feature set.
S3 App components: There exist four different types of components in an application, each defining different interfaces to the system: activities, services, content providers and broadcast receivers. Every application can declare several components of each type in the manifest. The names of these components are also collected in a feature set, as the names may help to identify well known components of malware. For example, several variants of a particular family share the name of particular services in Table 3.

Table 3.

Id	Family	#	*Id*	Family	#
A	FakeInstaller	925	*K*	Adrd	91
B	DroidKungFu	667	*L*	DroidDream	81
C	Plankton	625	*M*	LinuxLotoor	70
D	Opfake	613	*N*	GoldDream	69
E	GingerMaster	339	*O*	MobileTx	69
F	BaseBridge	330	*P*	FakeRun	61
G	Iconosys	152	*Q*	SendPay	59
H	Kmin	147	*R*	Gappusin	58
I	FakeDoc	132	*S*	Imlog	43
J	Geinimi	92	*T*	SMSreg	41

S4 Filtered intents: Inter-process and intra-process communication on Android is mainly performed through intents: passive data structures exchanged as asynchronous messages and allowing information about events to be shared between different components and applications. All intents listed in the manifest are collected as another feature set, as malware often listens to specific intents. A typical example of an intent message involved in malware is BOOT COMPLETED, which is used to trigger malicious activity directly after re- booting the smartphone.

S5 Restricted API calls: The Android permission system restricts access to a series of critical API calls. The method searches for the occurrence of these calls in the disassembled code in order to gain a deeper under- standing of the functionality of an application. A particular case, revealing malicious behavior, is the use of restricted API calls for which the required permissions have not been requested. This may indicate that the malware is using

root exploits in order to surpass the limitations imposed by the Android platform.

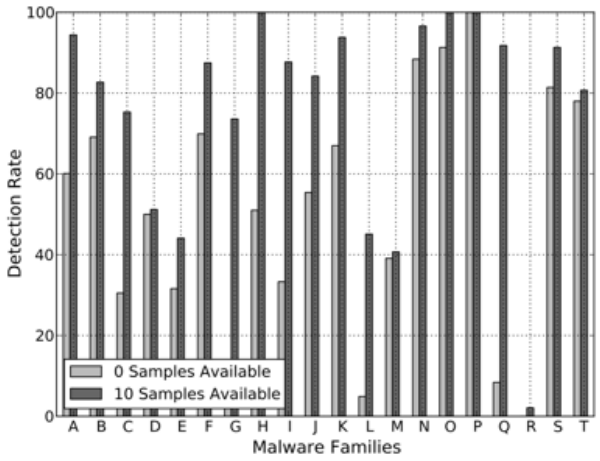

Fig. 1. Top malware families in the dataset.

S6 Used permissions: The complete set of calls extracted in S5 is used as the ground for determining the subset of permissions that are both requested and actually used. For this purpose, the method introduced by Felt et al. is implemented to match API calls and permissions. In contrast to S5, this feature set provides a more general view on the behavior of an application as multiple API calls can be protected by a single per- mission (e.g., sendMultipartTextMessage() and sendTextMessage() both require that the SEND SMS permission is granted to an application).

Suspicious API calls: Certain API calls allow access to sensitive data or resources of the smartphone and are frequently found in malware samples. As these calls can specially lead to malicious behavior, they are extracted and gathered in a separated feature set. In particular, the following types of API calls are collected:

- API calls for accessing sensitive data
- API calls for communicating over the net- work
- API calls for sending and receiving SMS messages
- API calls for execution of external commands
- API calls frequently used for obfuscation

Network addresses: Malware regularly establishes net- work connections to retrieve commands or data collected from the device. Therefore, all IP ad- dresses, hostnames and URLs found in the disassembled code are included in the last set of features. Some of these addresses might be involved in botnets and thus present in several malware samples, which can help to improve the learning of detection patterns.

Malicious activity is usually reflected in specific patterns and combinations of the extracted features. For example, a malware sending premiums SMS messages might contain the permission SEND SMS in set S2, and the hardware component android.hardware.telephony in set S1. Ideally, one would like to formulate Boolean expressions that capture these dependencies between features and return true if a malware is detected. However, inferring Boolean expressions from real-world data is a hard problem and difficult to solve efficiently.

To present a solution, it is aimed at capturing the dependencies between features using concepts from machine learning. As most learning methods operate on numerical vectors, the extracted feature sets to a vector space is mapped to start with. To this end, one defines a joint set S that comprises all observable strings contained in the 8 feature sets. It is ensured that elements of different sets do not collide by adding a unique prefix to all strings in each feature set. In the evaluation the set S contains roughly 545,000 different features.

Using the set S, a multi-dimensional vector space is defined, where each dimension is either 0 or 1. An application x is mapped to this space by constructing a vector, such that for each feature s extracted from x the respective dimension is set to 1 and all other dimensions are 0. Applications sharing similar features lie close to each other in this representation, whereas applications with mainly different features are separated by large distances. Moreover, directions in this space can be used to describe combinations of features and ultimately enable us to learn explain- able detection models.

Let us, as an example, consider a malicious application that sends premium SMS messages and thus needs to request certain permissions and hardware components. At a first glance, the map seems inappropriate for the lightweight analysis of applications, as it embeds data into a high-dimensional vector space. Fortunately, the number of features extracted from an application is linear in its size. That is, an application x containing m bytes of code and data contains at most m feature strings. As a consequence, only m dimensions are non-zero in the vector irrespective of the dimension of the vector space. It thus suffices to only store the features extracted from an application for sparsely representing the vector.

3 Results and Evaluations

A machine with 3.1 GHz Intel Core i7and 16GB RAM which was powerful enough to train the deep neural nets with many hidden layers was utilized. The simplest method to evaluate the performance of a machine learning algorithm is to use different training and testing datasets. The original dataset is split into two parts. Train the algorithm on the first part, make predictions on the second part and evaluate the predictions against the expected results. The size of the split can depend on the size and specifics of the dataset, although it is common to use 67% of the data for training and the remaining 33% for testing. Ideally, you want to select a model at the sweet spot between under fitting and overfitting. This is the goal but is very difficult to do in practice. To understand this goal, one can look at the performance of a machine learning algorithm over time as it is learning a training data. We can plot both the skill on the training data and the skill on a test dataset we have held back from the training process. Over time, as the algorithm learns, the error for the model on the training data goes down and so does the error on the test dataset. If it is trained for too long, the error on the training dataset may continue to decrease because the model is overfitting and learning the irrelevant detail and noise in the training dataset. At the same time the error for the test set starts to rise again as the model's ability to generalize decreases. The sweet spot is the point just before the error on the test dataset starts to increase where the model has good skill on both the training dataset and the unseen test dataset. You can perform this experiment with one's favorite machine learning algorithms. This is often not useful technique in practice, because by choosing the stopping point for training using the skill on the test dataset it means that the test set is no longer unseen or a standalone objective measure.

Some knowledge (a lot of useful knowledge) about that data has leaked into the training procedure. There are two additional techniques you can use to help find the sweet spot in practice, resampling methods and a validation dataset.

Both overfitting and under fitting can lead to poor model performance. But by far the most common problem in applied machine learning is overfitting. Overfitting is such a problem because the evaluation of machine learning algorithms on training data is different from the evaluation the research work actually cares about the most, namely how well the algorithm performs on unseen data. There are two important techniques that you can use when evaluating machine learning algorithms to limit overfitting: either use a resampling technique to estimate model accuracy or hold back a validation dataset. The most popular resampling technique is k-fold cross validation. It allows you to train and test the model k-times on different subsets of training data and build up an estimate of the performance of a machine learning model on unseen data. A validation dataset is simply a subset of training data that one holds back from your machine learning algorithms until the very end of your project. After you have selected and tuned your machine learning algorithms on your training dataset you can evaluate the learned models on the validation dataset to get a final objective idea of how the models might perform on unseen data. Using cross validation is a gold standard in applied machine learning for estimating model accuracy on unseen data. If you have the data, using a validation dataset is also an excellent practice. This algorithm evaluation technique is very fast. It is ideal for large datasets (millions of records) where there is strong evidence that both splits of the data are representative of the underlying problem. Because of the speed, it is useful to use this approach when the algorithm you are investigating is slow to train. A downside of this technique is that it can have a high variance. This means that differences in the training and test dataset can result in meaningful differences in the estimate of accuracy. In the example below the dataset is divided into 67%/33% splits for training and testing and evaluating the accuracy of a Logistic Regression model.

Cross validation is an approach that you can use to estimate the performance of a machine learning algorithm with less variance than a single train-test set split. It works by splitting the dataset into k-parts (e.g., k = 5 or k = 10). Each split of the data is called a fold. The algorithm is trained on k 1 folds with one held back and tested on the held back fold. This is repeated so that each fold of the dataset is given a chance to be the held back test set. After running cross validation, you end up with k different performance scores that you can summarize using a mean and a standard deviation. The result is a more reliable estimate of the performance of the algorithm on new data. It is more accurate because the algorithm is trained and evaluated multiple times on different data. The choice of k must allow the size of each test partition to be large enough to be a reasonable sample of the problem, whilst allowing enough repetitions of the train-test evaluation of the algorithm to provide a fair estimate of the algorithm's performance on unseen data. For modest sized datasets in the thousands or tens of thousands of records, k values of 3, 5 and 10 are common. In the example below 10-fold cross validation is used. It is a good practice to prepare the data before modeling. Neural network models are especially suitable to having consistent input values, both in scale and distribution. An effective data preparation scheme for tabular data when building neural

network models is standardization. This is where the data is rescaled such that the mean value for each attribute is 0 and the standard deviation is 1. This preserves Gaussian and Gaussian-like distributions whilst normalizing the central tendencies for each attribute. When the data is comprised of attributes with varying scales, many machine learning algorithms can benefit from rescaling the attributes to all have the same scale. Often this is referred to as normalization and attributes are often rescaled into the range between 0 and 1. This is useful for optimization algorithms in used in the core of machine learning algorithms like gradient descent. It is also useful for algorithms that weight inputs like regression and neural networks. An important concern with the dataset is that the input attributes all vary in their scales because they measure different quantities. It is almost always good practice to prepare the data before modeling it using a neural network model. Continuing on from the above baseline model, one can re-evaluate the same model using a standardized version of the input dataset. A pipeline framework is used to perform the standardization during the model evaluation process, within each fold of the cross validation. This ensures that there is no data leakage from each test set cross validation fold into the training data.

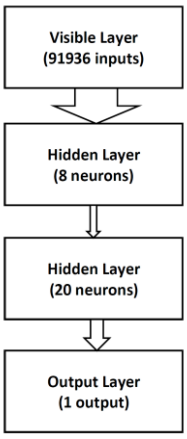

Fig. 2. Architecture of the deep neural network

One way to improve the performance of a neural network is to add more layers. This might allow the model to extract and recombine higher order features embedded in the data. In this section the effect of adding one more hidden layer to the model is evaluated. This is as easy as defining a new function that will create this deeper model, copied from the baseline model above. A new line after the first hidden layer is then inserted. In this case with about half the number of neurons. Another approach to increasing the representational capacity of the model is to create a wider network. In this section, we evaluate the effect of keeping a shallow network architecture and nearly doubling the number of neurons in the hidden layer. Again, all is needed to do is define a new function that creates the neural network model. Here, the number of neurons in the hidden layer compared to the baseline model is increased from 13 to 20.

Figure 2 shows the final architecture of the deep neural network, and it achieved an

accuracy of up to 99% on a large dataset. The confusion matrix and the receiver optimistic curve is shown in Figure 3 and Figure 4 respectively.

		Predicted	
		Safe	Malicious
Actual	Safe	99%	1%
	Malicious	1%	99%

Fig. 3. Mean confusion matrix

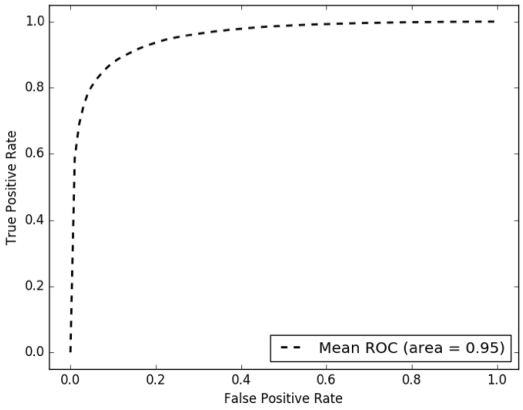

Fig. 4. Mean receiver optimistic characteristic curve

4 Conclusion

With the increase of large-scale datasets such as user behavior in minute detail, results from static code analysis of mobile applications high dimensional datasets that traditional machine learning is unable to cope. Cheap computers, storage & memory alongside new statistical modelling & data mining techniques have given us the opportunity to solve problem in range of domains, not just in security of mobile apps. Traditional machine learning techniques such as support vector machine or random forest can build accurate models for many applications but fails to achieve the same level of performance of deep neural network with larger datasets.

This research proposes to use the novel approach of deep neural network to solve the high dimensionality problem and to find more patterns at various levels. The features from various levels are gathered to represent permissions of apps in a high dimensional space. Simulation results show that the detection accuracy is over 99%, with detection time under 100 milliseconds. In future the detector should look at using other modelling techniques such as recurrent neural network and convolutional network to take into account the time dimension and use graphical processing units to train the models even faster.

References

[1] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita- Rotaru, and I. Molloy, "Using probabilistic generative models for ranking risks of android apps," in Proceedings of the 2012 ACM Conference on Computer and Communications Security, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 241–252.
[2] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in Proceedings of the 18th ACM Conference on Computer and Communications Security, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 627–638.
[3] A. Genkin, D. D. Lewis, and D. Madigan, "Large-scale bayesian logistic regression for text categorization," Technometrics, p. 2007.
[4] Q. Wang, L. Si, and D. Zhang, "A discriminative data-dependent mixture-model approach for multiple instance learning in image classification." in ECCV (4), ser. Lecture Notes in Computer Science, A. W. Fitzgibbon, S. Lazebnik, P. Perona, Y. Sato, and C. Schmid, Eds., vol. 7575. Springer, 2012, pp. 660–673.
[5] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: Scalable and accurate zero-day android malware detection," in Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services, ser. MobiSys '12. New York, NY, USA: ACM, 2012, pp. 281–294.
[6] A.-D. Schmidt, J. H. Clausen, S. A. Camtepe, and S. Albayrak, "Detecting symbian os malware through static function call analysis," in Proceedings of the 4th IEEE International Conference on Malicious and Unwanted Soft- ware (Malware 2009). IEEE, 2009, pp. 15–22.
[7] H. He and E. A. Garcia, "Learning from imbalanced data," IEEE Trans. on Knowl. and Data Eng., vol. 21, no. 9, pp. 1263–1284, Sep. 2009.
[8] J. Z. Kolter and M. A. Maloof, "Learning to detect and classify malicious executables in the wild," J. Mach. Learn. Res., vol. 7, pp. 2721–2744, Dec. 2006.
[9] J. Jang, D. Brumley, and S. Venkataraman, "Bitshred: Feature hashing malware for scalable triage and semantic analysis," in Proceedings of the 18th ACM Conference on Computer and Communications Security, ser. 1 CCS '11. New York, NY, USA: ACM, 2011, pp. 309–320.
[10] A. Desnos, "Android: Static analysis using similarity distance," 2012 45th Hawaii International Conference on System Sciences (HICSS), vol. 00, no. undefined, pp. 5394–5403, 2012.
[11] A.-D. Schmidt, R. Bye, H.-G. Schmidt, J. H. Clausen, O. Kiraz, K. A. Yu ̈ksel, S. A. C ̧amtepe, and S. Albayrak, "Static analysis of executables for collaborative malware detection on android." in ICC. IEEE, 2009, pp. 1–5.
[12] Rosmalissa Jusoh, Ahmad Firdaus, Shahid Anwar, Mohd Zamri Osman, Mohd Faaizie Darmawan, and Mohd Faizal Ab Razak, "Malware detection using static analysis in Android: a review of FeCO (features, classification, and obfuscation)", PeerJ Comput Sci. 2021; 7: e522, doi: 10.7717/peerj-cs.522,11th June 2021.
[13] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining API-level features for robust malware detection in Android," in Proceedings of the 9th Inter- national Conference on Security and Privacy in Communication Networks, Sep. 2013.
[14] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 5–14.
[15] Vegi Shanmukh, **"**Image Classification Using Machine Learning-Support Vector Machine(SVM)", accessed on 25th February 2022, https://medium.com/analytics-vidhya/image-classification-using-machine-learning-support-vector-machine-svm-dc7a0ec92e01, 3rd March 2021**.**
[16] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario, "Automated classification and analysis of internet malware," in Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection, ser. RAID'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 178–197.
[17] A. Shabtai and Y. Elovici, "Applying behavioral detection on android- based devices," 10 2012.
[18] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid android: Versatile protection for smartphones," in Proceedings of the 26th Annual Computer Security Applications Conference, ser. ACSAC 10. New York, NY, USA: ACM, 2010, pp. 347–356. [Online]. Available: http://doi.acm.org/10.1145/1920261.1920313
[19] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior- based malware detection system for android," in Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, ser. SPSM '11. New York, NY, USA: ACM, 2011, pp. 15–26.
[20] 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, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 217–228.
[21] A. P. Felt, K. Greenwood, and D. Wagner, "The effectiveness of application permissions," in Proceedings of the 2Nd USENIX Conference on Web Application Development, ser. WebApps'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 7–7.
[22] J. Davis and M. Goadrich, "The relationship between precision-recall and roc curves," in Proceedings of the 23rd International Conference on Machine Learning, ser. ICML '06. New York, NY, USA: ACM, 2006, pp. 233–240.
[23] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue, "Droid-sec: Deep learning in an- droid malware detection," in ACM SIGCOMM Computer Communication Review, vol. 44, no. 4. ACM, 2014, pp. 371–372.
[24] Z. Yuan, Y. Lu, and Y. Xue, "Droiddetector: android malware characteri- zation and detection using deep learning," Tsinghua Science and Technol- ogy, vol. 21, no. 1, pp. 114–123, 2016.
[25] K. Worldpanel, "Smartphone os market share," 2015.