A Comparative Analysis of Android Malware

Neeraj Chavan, Fabio Di Troia and Mark Stamp

Department of Computer Science, San Jose State University, San Jose, California, U.S.A.

Keywords: Malware, Android, Machine Learning, Random Forest, Logistic Model Tree, Artificial Neural Network.

Abstract: In this paper, we present a comparative analysis of benign and malicious Android applications, based on static features. In particular, we focus our attention on the permissions requested by an application. We consider both binary classification of malware versus benign, as well as the multiclass problem, where we classify malware samples into their respective families. Our experiments are based on substantial malware datasets and we employ a wide variety of machine learning techniques, including decision trees and random forests, support vector machines, logistic model trees, AdaBoost, and artificial neural networks. We find that permissions are a strong feature and that by careful feature engineering, we can significantly reduce the number of features needed for highly accurate detection and classification.

1 INTRODUCTION

As of 2017, the Android OS accounted for more than 85% of the mobile OS market, and there were more than 82 billion application (app) downloads from the Google Play store during 2017 (Android Statistics, 2017). Predictably, the number of Android malware apps is also large—it is estimated that there were 3.5 million such apps in 2017, representing more than a six-fold increase since 2015 (Malware Forecast, 2017). It follows that effective malware detection on Android devices is of critical importance.

Features that can be collected without executing the code are said to be "static," while features that require execution (or emulation) are considered "dynamic." Dynamic analysis is generally more informative and dynamic features are typically more difficult for malware writers to defeat via obfuscation (Damodaran et al., 2017). However, extracting dynamic features is likely to be far more costly and time consuming, as compared to most static features. Since efficiency is important on a mobile platform, in this paper, we focus on static analysis. Specifically, we consider the related problems of Android malware detection and classification based on requested permissions—features that are easily obtained from the manifest file. We analyze this feature over substantial malware datasets, and we consider the problem of feature reduction in some detail. Perhaps somewhat surprisingly, we find that a small subset of these features suffices. This is significant, since malware de-

tection on a mobile device must be fast, efficient, and effective—the approach consider here meets all three of these criteria.

The remainder of this paper is organized as follows. In Section 2, we briefly discuss relevant background topics, including related work. In Section 3 we discuss our experimental design and give our experimental results. We also provide some discussion of our results. Finally, in Section 4 we conclude the paper and give some suggestions for future work.

2 BACKGROUND

In this section, we first briefly discuss relevant background topics. First, we outline each of the machine learning techniques considered in this paper. Then we discuss some examples of relevant related work.

2.1 Machine Learning Techniques

In this research, we employ a wide variety of machine learning techniques. A detailed discussion of these techniques is well beyond the scope of this paper—here, we simply provide a high-level overview.

Random forest can be viewed as a generalization of the simple concept of a decision tree (Breiman and Cutler, 2001). While decision trees are indeed simple, they tend to grossly overfit the training data, and hence provide little, if any, actual

"learning." A random forest overcomes the weakness of decision trees by the use of bagging (i.e., bootstrap aggregation), that is, multiple decision trees are trained using subsets of the data and features. Then, a voting procedure based on these multiple decision trees is typically used to determine the random forest classification. In each of our random forest experiments, we use 100 trees.

Random trees are a subclass of random forests, where the bagging only involves the classifiers, not the data. We would generally expect better results from a random forest, but random trees will be more efficient. Our random trees results are based on a single tree.

J48 is a specific implementation of the C4.5 algorithm (Quinlan, 2018), which is a popular method for constructing decision trees. We would generally expect random trees to outperform decision trees while, as mentioned above, random forests should typically outperform random trees. However, decision trees are more efficient than random trees, which are more efficient than random forests. Thus, it is worth experimenting with all three of these tree-based algorithms to determine the proper tradeoff between efficiency and accuracy.

Artificial Neural Network (ANN) represents a large class of machine learning techniques that attempt to (loosely) model the behavior of neurons and trained using backpropagation (Stamp, 2018). While ANNs are not a new concept, having first been proposed in the 1940s, they have found renewed interest in recent years as computing power has become sufficient to effectively deal with "deep" neural network, i.e., networks that include many hidden layers. Such deep networks have pushed machine learning to new heights. For our ANN experiments, we use two hidden layers, with 10 neurons per layer, the rectified linear unit (ReLU) for the activation functions on the hidden layers, and a sigmoid function for the output layer. Training consists of 100 epochs, with the learning rate set at $\alpha = 0.001$.

Support Vector Machine (SVM) is a popular and effective machine learning technique. According to (Bennett and Campbell, 2000), "SVMs are a rare example of a methodology where geometric intuition, elegant mathematics, theoretical guarantees, and practical algorithms meet." When training an SVM, we attempt to find a separating hyperplane that maximizes the "margin," i.e., the minimum distance between the classes in the training set. A particularly nice feature of SVMs is that we can map the input data to a higher dimensional feature space, where we are much more likely to be able to separate the data. And, thanks to the so-called "kernel trick," this mapping entails virtually no computational penalty. All of our SVM experiments are based on a linear kernel function with $\varepsilon = 0.001$.

Logistic Model Tree (LMT) can be viewed as a hybrid of decision trees and logistic regression. That is, in an LMT, decision trees are constructed, with logistic regression functions at the leaf nodes (Landwehr et al., 2005). In our LMT experiments, we use a minimum of 15 instances, where each node is considered for splitting.

Boosting is a general—and generally fairly straightforward—approach to building a strong model from a collection of (weak) models. In this paper, we employ the well-known adaptive boosting algorithm, AdaBoost (Stamp, 2017a).

Multinomial naïve Bayes is a form of naïve Bayes where the underlying probability is assumed to satisfy a multinomial distribution. In naïve Bayes, we make a strong independence assumption, which results in an extremely simply "model" that often performs surprisingly well in practice.

The static analysis can be done using the Java Bytecode extracted after disassembling the `apk` file. Also we can extract permissions from the manifest file. In this paper, we take advantage of the static analysis using permissions of applications and use them for detecting malware and also classify different malware families. The effectiveness of these techniques is analyzed using multiple machine learning algorithms.

2.2 Selected Related Work

The paper (Feng et al., 2014) discusses a tool the authors refer to as Appopscopy, which implements a semantic language-based Android signature detection strategy. In their research, general signatures are created for each malware family. Signature matching is achieved using inter-component call graphs based on control flow properties and the results are enhanced using static taint analysis. The authors report an accuracy of 90% on a malware dataset containing 1027 samples, with the accuracy for individual families ranging from a high of 100% to a low of 38%.

In the research (Fuchs et al., 2009), the authors analyze a tool called SCanDroid that they have developed. This tool extracts features based on data flow. The work in (Abah et al., 2015) relies on *k*-nearest neighbor classification based on a variety of features,

include incoming and outgoing SMS and calls, device status, running processes, and more. This work claims that an accuracy of 93.75% is achieved. In the research (Aung and Zaw, 2013), the authors propose a framework and test a variety of machine learning algorithms to analyze features based on Android events and permissions. Experimental results from a dataset of some 500 malware samples yield a maximum accuracy of 91.75% for a random forest model.

In the paper (Afonso et al., 2015), the authors propose a dynamic analysis technique that is focused on the frequency of system and API calls. A large number of machine learning techniques are tested on a dataset of about 4500 malicious Android apps. The authors give accuracy results ranging from 74.53% to 95.96%. Again, a random forest algorithm achieves the best results.

The research (Enck et al., 2010) discusses a dynamic analysis tool, TaintDroid. This sophisticated system analyzes network traffic to search for anomalous behavior—the research is in a similar vein as (Feng et al., 2014), but with the emphasis on efficiency. Another Android system call analysis technique is considered in (Dimjašević et al., 2015).

Our work is perhaps most closely related to the research in (Sugunan et al., 2018) and (Kapratwar et al., 2017) which, in turn, built on the groundbreaking work of (Arp et al., 2014) and (Schmeelk et al., 2015), as well as that in (Zhou et al., 2012). In (Arp et al., 2014), for example, an accuracy of 93.9% is attained over a dataset of 5600 malicious Android apps.

The paper (Kapratwar et al., 2017) considers static and dynamic analysis of Android malware based on permissions and API calls, respectively. A robustness analysis is presented, and it is suggested that malware writers can most likely defeat detectors that rely on permissions. We provide a more careful analysis in this paper and find that such is not the case.

3 EXPERIMENTS AND RESULTS

In this section, we first discuss our datasets and feature extraction process. Then we turn our attention to feature engineering, that is, we determine the most significant features for use in our experiments. We also discuss our experimental design before presenting results from a wide variety experiments.

3.1 Datasets

We use the Android Malware Genome Project (Zhou and Jiang, 2012) dataset. This data consists mainly of apk files obtained from various malware forums

and Android markets—these samples have been widely used in previous research. Labels are included, which specify the family to which each sample belongs. Thus, the data can be used for both binary classification (i.e., malware versus benign) and the multiclass (i.e., family) classification problems.

For our benign dataset, we crawled the PlayDrone project (PlayDrone, 2018), as found on the Internet Archive (Internet Archive, 2018). The resulting apk files might include malicious samples. Therefore, we used Androguard (Androguard, 2018) to filter broken and potentially malicious apk files. Table 1 gives the number of malware and benign samples that we obtained. These samples will be used in our binary classification (malware versus benign) and multiclass (malware family) experiments discussed below.

Table 1: Datasets.

| Experiment | Samples | |
	Type	Number
Detection	Malware	989
	Benign	2657
Classification	Malware	1260

3.2 Feature Extraction

To extract static features, we need to reverse engineer the apk files. We again use Androguard this reverse engineering task. The manifest file, `AndroidManifest.xml`, contains numerous potential static features; here we focus on the permissions requested by an application.

From the superset of malware and benign samples, we find that there are 230 distinct permissions. Thus, for each apk, a feature vector is generated based on these permissions. The feature vector is simply a binary sequence of length 230, which indicates whether each of the corresponding permissions is requested by the application or not. Along with each feature vector, we have a denoting label of $+1$ or -1, indicating whether the sample is malware or benign, respectively. The overall architecture, in the case of binary classification, is given in Figure 1.
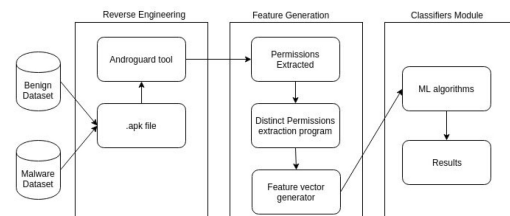

Figure 1: Binary classification architecture.

For the multiclass (family) classification problem, essentially the same process is followed as for the binary classification case. However, we only examine malware samples, and over our malware dataset, we find that only 118 distinct permissions occur. Thus, the feature vectors for the multiclass problem are of length 118.

3.3 Feature Engineering

It is likely that many of the features under consideration (i.e., permissions) provide little or no discriminating information, with respect to the malware versus benign or the malware classification problem. It is useful to remove such features from the analysis, as they essentially act as noise, and can therefore cause us to obtain worse results than we would with a smaller, but more informative, feature set. It is also useful to remove extraneous features so that scoring is as efficient as possible. Consequently, our immediate goal is to determine features that are of no value for our analysis, and remove them from subsequent consideration.

There are several techniques for determining feature significance. Here we consider two distinct approaches to this problem. First, we use information gain to reduce the feature set. Second, we use recursive feature elimination (RFE) based on a linear SVM. Information gain is easily computed and gives us a straightforward means of eliminating features. RFE is somewhat more involved, but accounts for feature interactions in a way that a simple information gain calculation cannot.

The information gain (IG) provided by a feature is defined as the expected reduction in entropy when we branch on that feature. In the context of a decision tree, information gain can be computed as the entropy of the parent node minus the average weighted entropy of its child nodes. We measure the information gain for each feature, and select features in a greedy manner. In a decision tree, this has the desirable effect of putting decisions based on the most informative features closest to the root. This is desirable, since the entropy is reduced as rapidly as possible, and enables the tree to be simplified by trimming features that provide little or no gain.

For our purposes, we simply use the information gain to reduce the number of features, then apply various machine learning techniques to this reduced feature set. Based on the information gain, we selected the 74 highest ranked features—the top 10 of these features are given in Table 2. Features that ranked outside the top 74 provided no improvement in our results.

Table 2: Permissions ranked by IG.

Score	Permission
0.30682	READ_SMS
0.28129	WRITE_SMS
0.17211	READ_PHONE_STATE
0.15197	RECEIVE_BOOT_COMPLETED
0.14087	WRITE_APN_SETTINGS
0.13045	RECEIVE_SMS
0.10695	SEND_SMS
0.10614	CHANGE_WIFI_STATE
0.10042	INSTALL_PACKAGES
0.10019	RESTART_PACKAGES

As mentioned above, we also reduce the feature set using RFE based on a linear SVM. In a linear SVM, a weight is assigned to each feature, with the weight signifying the importance that the SVM attaches to the feature. For our RFE approach, we eliminate the feature with the lowest linear SVM weight, then train a new SVM on this reduced (by one) feature set. Then we again eliminate the feature with the lowest SVM weight, train a new linear SVM on this reduced feature set. This process is continued until a single feature remains, and in this way, we obtain a complete ranking of the features. The potential advantage of this RFE technique is that it accounts for feature interactions among all of the reduced feature sets. The top 10 features obtained using RFE based on a linear SVM are listed in Table 3.

Table 3: Permissions ranked by RFE using a linear SVM.

Rank	Permission
1	WRITE_APN_SETTINGS
2	WRITE_CALENDAR
3	WRITE_CALL_LOG
4	WRITE_CONTACTS
5	WRITE_INTERNAL_STORAGE
6	WRITE_OWNER_DATA
7	WRITE_SECURE_SETTINGS
8	WRITE_SETTINGS
9	WRITE_SMS
10	WRITE_SYNC_SETTINGS

In Figure 2, we give the cross validation score of the linear SVM as a function of the number of features, as obtained by RFE. We see that the top 82 features gives us an optimal result—additional features beyond this number provide no benefit. Consequently, we use the 82 top RFE features in our experiments below.
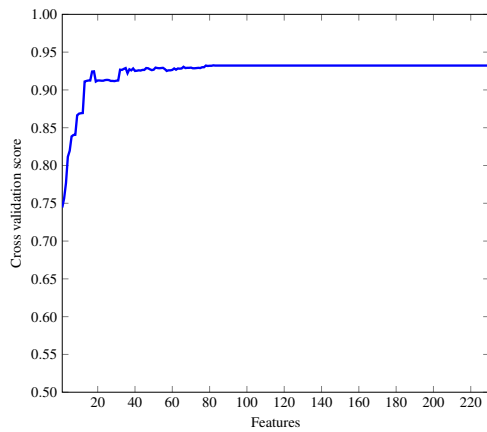
Figure 2: Recursive feature elimination (RFE).

3.4 Binary Classification

In this section, we discuss our binary classification experiments. We consider both the IG features and the RFE features. We test a wide variety of machine learning algorithms and provide various statistics.

First, we consider the IG features, as discussed in Section 3.3, above. Bar graphs of training and testing precision for each of the techniques discussed in Section 2.1 are plotted in Figure 3. For this experiment the number of samples in the malware and benign datasets are listed in Table 1, where we see that there is total of 3647 samples. Here, and in all subsequent experiments, we use 5-fold cross validation. Cross validation serves to maximize the number of test cases, while simultaneously minimizing the effect of any bias that might exist in the training data (Stamp, 2017b).
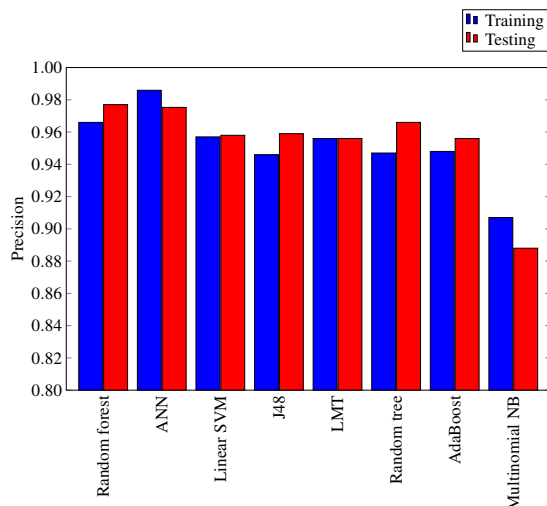
Given a scatterplot of experimental results, an ROC curve is obtained by graphing the true positive rate versus the false positive rate, as the threshold varies through the range of values. The area under the ROC curve (AUC) is between 0 and 1, inclusive, and can be interpreted as the probability that a randomly selected positive instance scores higher than a randomly selected negative instance (Stamp, 2017b). In Figure 4, we give the AUC statistic for the same set of IG feature experiments that we ahve summarized in Figure 3.
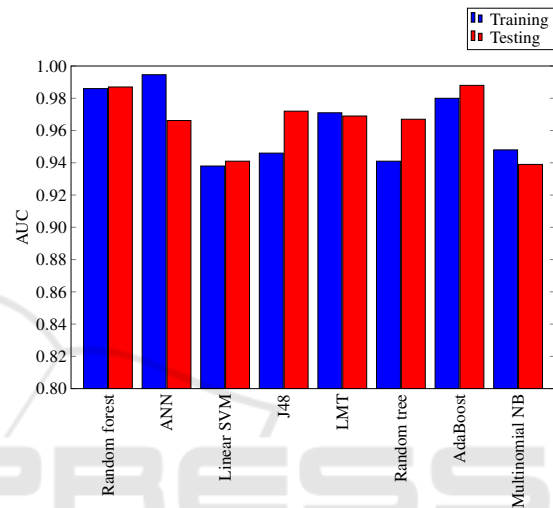

Figure 4: Machine learning comparison based on AUC (IG features).

We repeated the experiments above using the 82 RFE features, rather than the 74 IG features. The precision results for these machine learning experiments are given in Figure 5, while the corresponding AUC results are summarized in Figure 6.

Figure 3: Machine learning comparison based on precision (IG features).
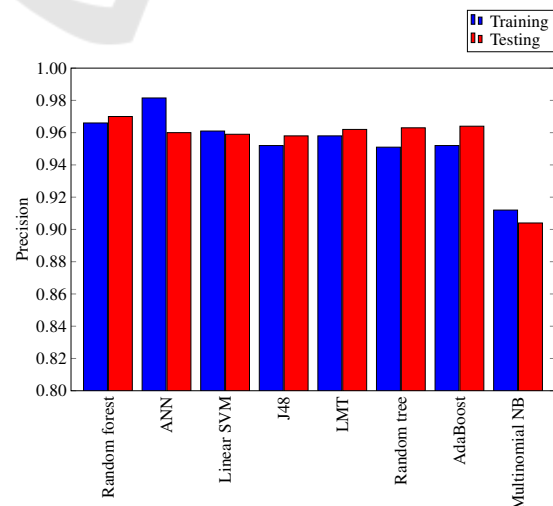

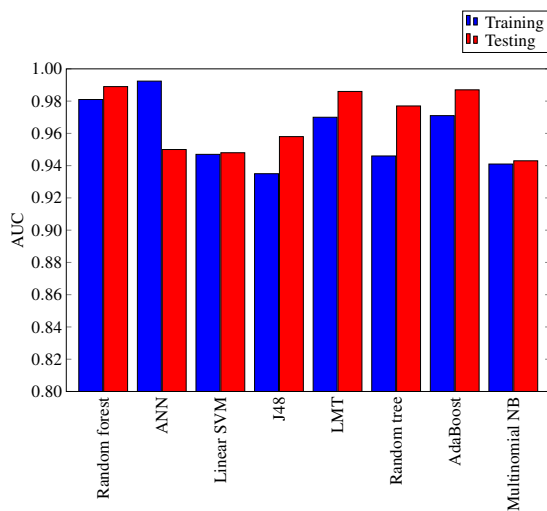Figure 5: Machine learning comparison based on precision (RFE features).

Figure 6: Machine learning comparison based on AUC (RFE features).

The performance between the various machine learning algorithms varies significantly, with multinomial naïve Bayes consistently the worst, while random forests and ANNs perform the best. The IG and RFE cases are fairly similar, although ANNs are better on the RFE features, with random forest are better on the IG features.

3.5 ANN Experiments

Since ANNs performed well in the experiments above, we have conducted additional experiments to determine the effect of an imbalanced dataset and to test the effect of small training sets. For these experiments, we use the IG features and the same binary classification dataset as above, with the skewed training sets selected at random. Again, we use 5-fold cross validation.

We experiment with three different ratios between the sizes of the malware and benign sets, namely, a ratio of 1:3 (i.e., three times as many benign samples as malware samples), as well as ratios of 1:6 and 1:12. For the 1:3 ratio, we have sufficient data to consider following four different cases:

- 100 malware and 300 benign
- 200 malware and 600 benign
- 400 malware and 1200 benign
- 800 malware and 2400 benign

For a 1:6 ratio, we have enough samples so that we can consider the following three cases:

- 100 malware and 600 benign
- 200 malware and 1200 benign
- 400 malware and 2400 benign

Finally, for the 1:12 ratio, we have sufficient data for

the following two cases:

- 100 malware and 1200 benign
- 200 malware and 2400 benign

The testing precision and AUC results for the 1:3, 1:6, and 1:12 training cases are given in Figures 7 (a) through (c), respectively. We see that, for example, with only 100 malware and 300 benign samples, we obtain a testing precision in excess of 0.98 and an AUC of approximately 0.96. Overall, these results show that the ANN performs extremely well, even with a small and highly skewed training set. This is significant, since we would like to train a model as soon as possible (in which case the training set may be small), and the samples are likely to be highly skewed towards the benign case.

3.6 Robustness Experiments

As an attack on permissions-based detection, a malware writer might simply request more of the permissions that are typical of benign samples, while still requesting permissions that are necessary for the malware to function. In this way, the permissions statistics of the malware samples would be somewhat closer to those of the benign samples. Analogous attacks have proven successful against malware detection based on opcode sequences (Lin and Stamp, 2011).

To simulate such an attack, for each value of $N = 0, 1, 2, \ldots, 20$, we include the top N benign permissions in each malware sample. For each of these cases, we have performed an ANN experiment, similar to those in Section 3.5, above. The precision and AUC results are given in the form of line graphs in Figure 8. Note that the $N = 0$ case corresponds to no modification to the malware permissions.

The results in Figure 8 show that there is a decrease in the effectiveness of the ANN when a small number of the most popular benign permissions are requested. However, when more than $N = 5$ permissions are included, the success of the ANN recovers, and actually improves on the unmodified $N = 0$ case. These results show that a straightforward attack on the permissions can have a modest effect, but we also see that permissions are a surprisingly robust feature.

3.7 Multiclass Classification

For the multiclass experiments in this section, we again use permission-based features. The metrics considered here are precision and the AUC. As above, we use five-fold cross validation in each experiment.

For these experiments, we use all malware samples in our dataset that include a family label. The distribution of these malware families is given in

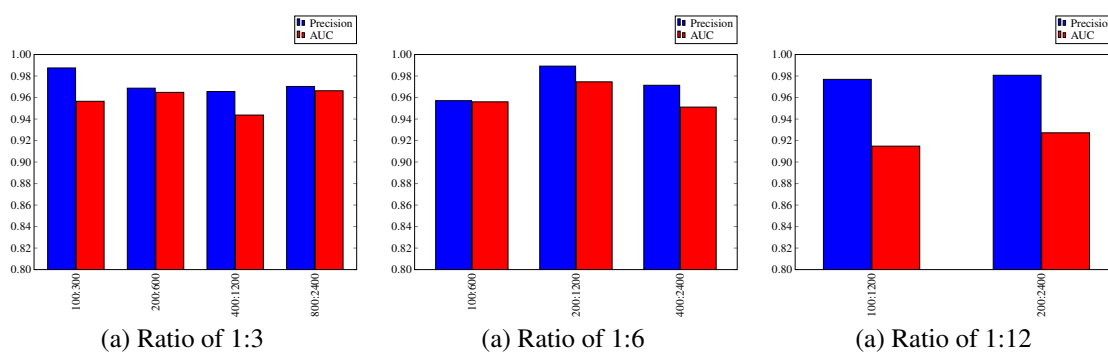(a) Ratio of 1:3   (a) Ratio of 1:6   (a) Ratio of 1:12

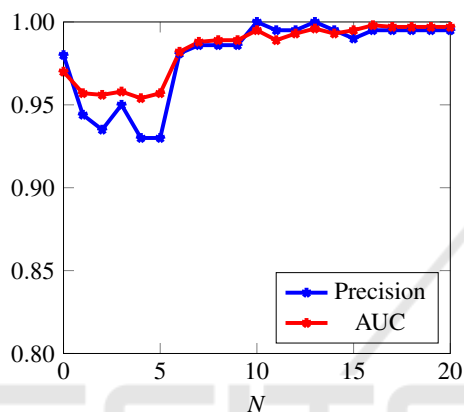Figure 7: ANN results for various malware to benign ratios.

Figure 8: Robustness tests based on ANN.

Figure 9. Note that we have 49 malware families and a total 1260 malware samples.

Bar graphs of precision and AUC results for multiple machine learning techniques are give in Figures 10 (a) and (b), respectively. We observe that a random forest performs best in these multiclass experiments, attaining a testing accuracy in excess of 0.95. This is an impressive number, given that we have such a large number of classes. Other machine learning techniques that perform well on this challenging multiclass problem include LMT and AdaBoost.

While there are 49 malware families in our dataset, from Figure 9 we see that a few large families dominate, while many of the "families" contain only 1 or 2 samples, and most are in the single digits. To determine the effect of this imbalance on our results, we also performed multiclass experiments on balanced datasets. In Figures 11 (a) through (c) we give results for balanced sets with 30, 40, and 50 samples per family, respectively. For example, 11 of the 49 malware families in our dataset have at least 30 samples. From each of these 11 families, we randomly select 30 samples, then perform multiclass experiments and we give the precision and AUC results in Figure 11 (a).

Overall the results in Figure 11 clearly demonstrate that our strong multiclass results are not an artifact of the imbalance in our dataset. In fact, with a sufficient number of samples per family, we obtain better results on the balanced dataset than on the highly imbalanced dataset. Of course, the number of families is much smaller in the balanced case, but we do have a significant number of families in all of these experiments.

Since a random forest performs best in the malware classification problem, we give the confusion matrices for this case in Figure 12. This confusion matrix provides a more fine-grained view of the results reported above. We observe that the misclassification are sporadic, in the sense that all of cases where a significant percentage of samples are misclassified occur in small families, with the number of samples being in the single digits.

4 CONCLUSION

The work in (Kapratwar et al., 2017) considered both permissions (i.e., a static feature) and system calls (i.e., a dynamic feature), and the interplay between the two. The authors concluded that a slight reduction in the number of permissions has a significant effect, and suggested that malware writers may be able to evade detection by relatively minor modifications to their code, such as reducing the number of permissions requested. Here, we provided a more nuanced analysis to show that it is likely to be significantly more difficult to evade permission-based detection than suggested in (Kapratwar et al., 2017). Specifically, in this paper we showed that a relatively small number of permissions can serve as a strong feature vector, even for the more challenging multiclass problem. These results indicate that eliminating the specific permissions that comprise the reduced feature set is likely not an option for most malware. In addition, we showed that taking the opposite tack,
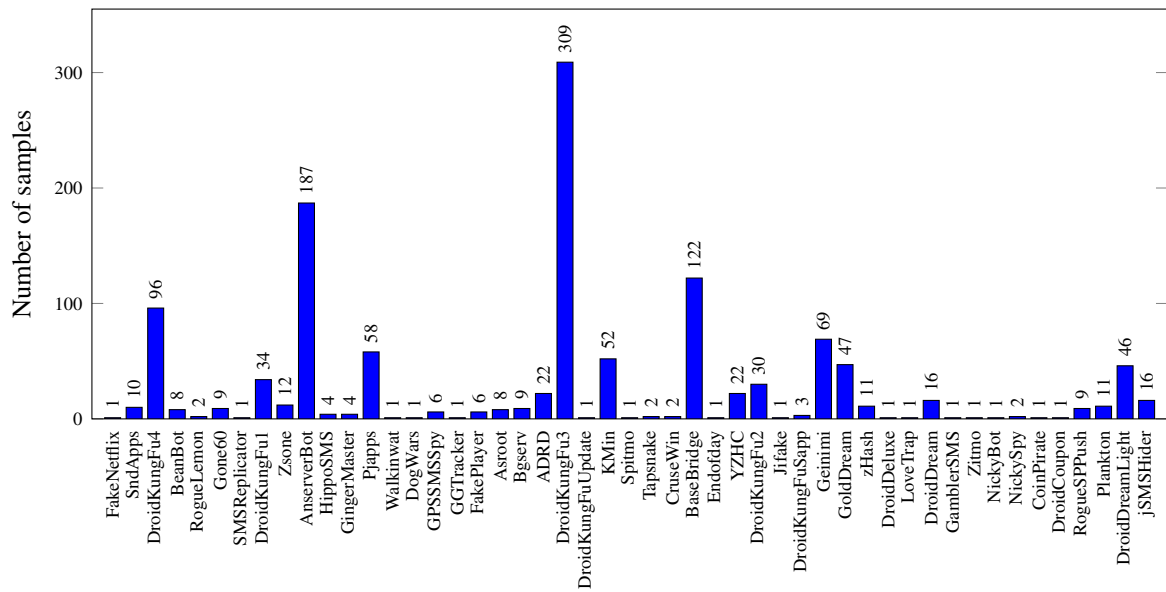
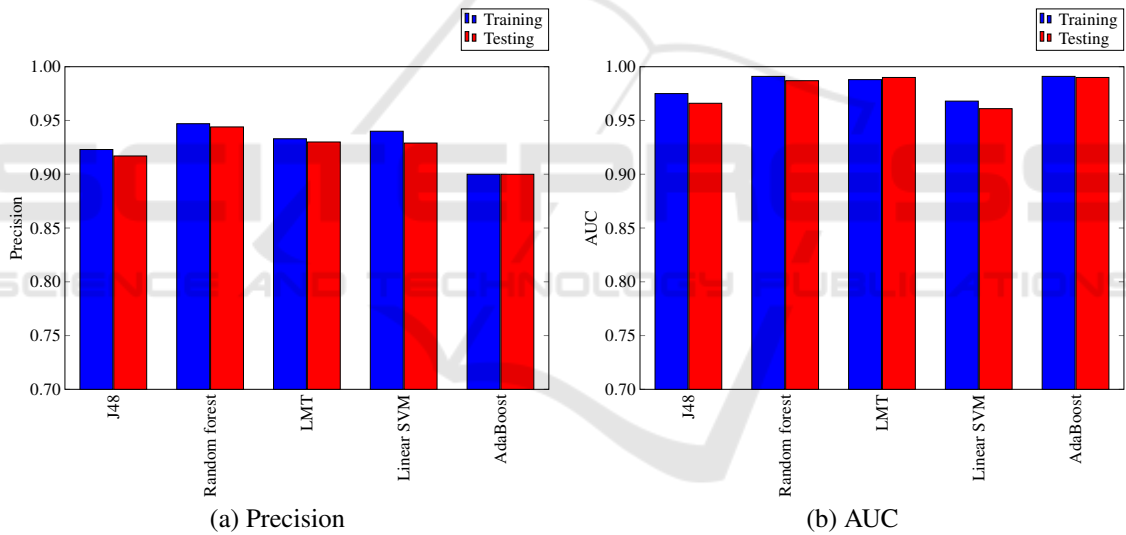Figure 9: Distribution of malware families.

(a) Precision

(b) AUC

Figure 10: Multiclass results.

(a) 30 samples (11 families)

(b) 40 samples (9 families)

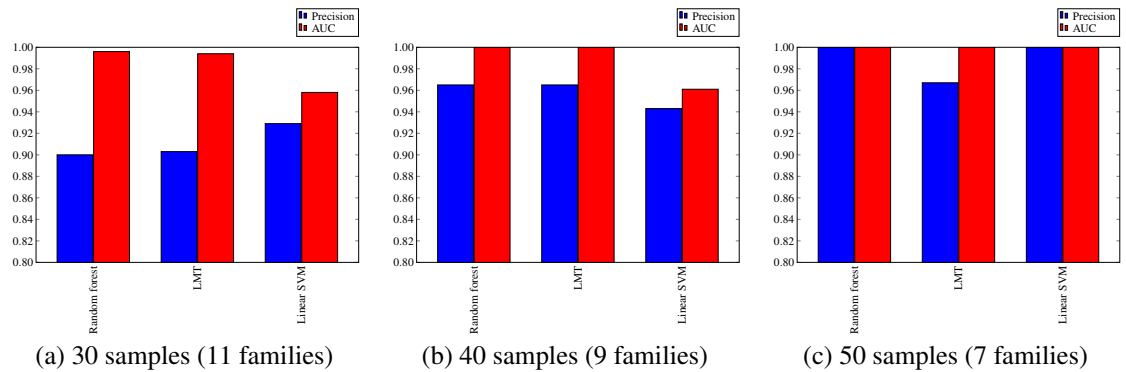(c) 50 samples (7 families)

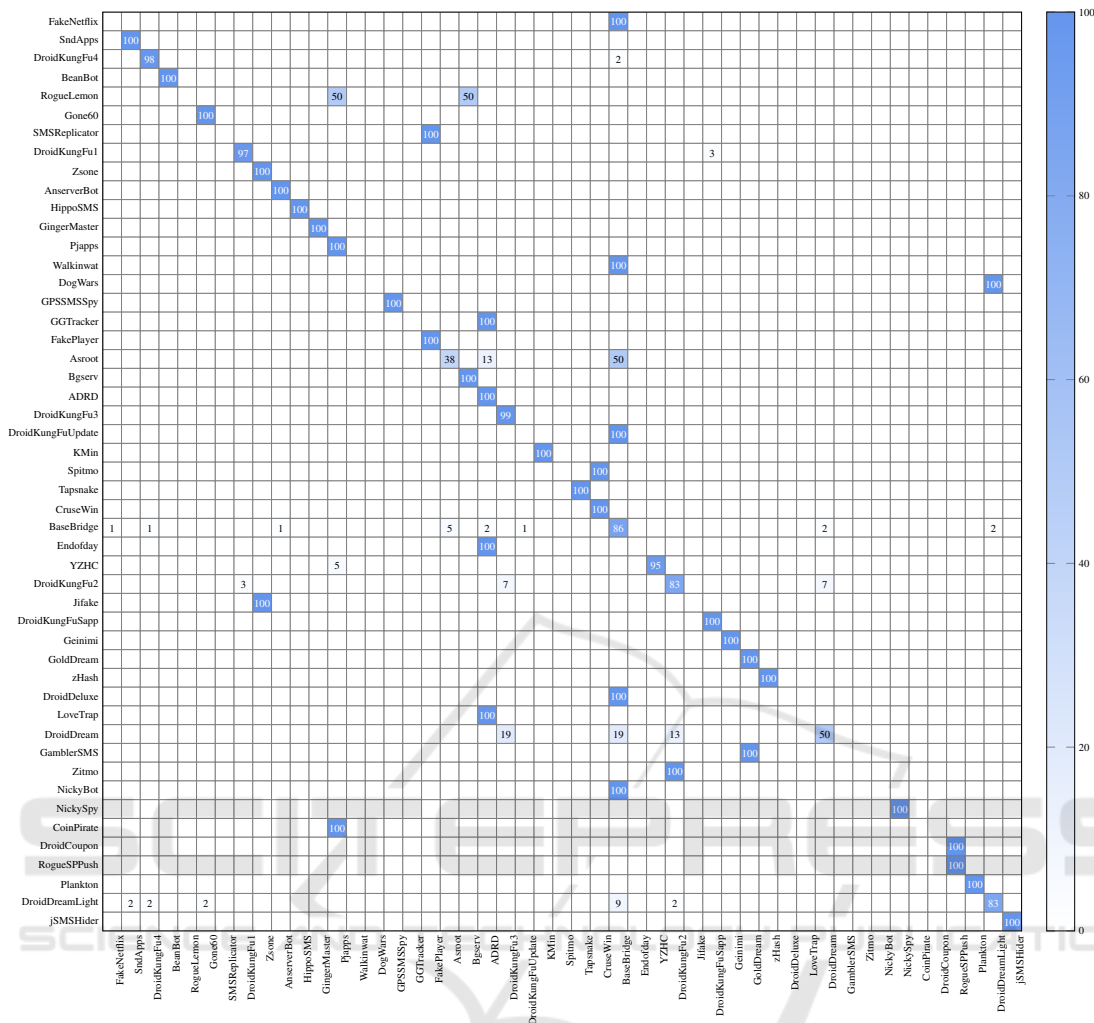Figure 11: Multiclass results with balanced datasets (testing).

Figure 12: Random forest confusion matrix as percentages (training).

that is, adding unnecessary permissions that are common among bening apps, is also of limited value. We conclude that features based on permissions are likely to remain a viable option for detecting Android malware.

Our experimental results also show that malware detection on an Android device is practical, since the necessary features can be extracted and scored efficiently. For example, using an ANN on a reduced feature set, we can obtain an AUC of 0.9920 for the binary classification problem. And even in the case of highly skewed data—as would typically be expected in a realistic scenario—an ANN can attain a testing accuracy in excess of 96%.

The malware classification problem is inherently more challenging than the malware detection problem. But even in this difficult case, we obtained a testing accuracy of almost 95%, based on a random forest. It is worth noting that a random forest also per-

forms well for binary classification, with about 97% testing accuracy. A random forest requires significantly less computing power to train, as compared to an ANN, and this might be a factor in some implementations, although training is often considered one-time work.

For future work, it would be interesting to further explore deep learning for Android malware detection, based on permissions. For ANNs, there are many parameters that can be tested, and it is possible that the ANN results presented in this paper can be significantly improved upon. As another avenue for future work, recent research has shown promising malware detection results by applying image analysis techniques to binary executable files; see, for example (Huang et al., 2018; Yajamanam et al., 2018). As far as the authors are aware, such analysis has not been applied to the mobile malware detection or classification problems.

REFERENCES

Abah, J., V, W. O., B, A. M., M, A. U., and S, A. O. (2015). A machine learning approach to anomaly-based detection on Android platforms. https://arxiv.org/abs/1512.04122.

Afonso, V. M., de Amorim, M. F., Grégio, A. R. A., Junquera, G. B., and de Geus, P. L. (2015). Identifying Android malware using dynamically obtained features. *Journal of Computer Virology and Hacking Techniques*, 11(1):9–17.

Androguard (2018). Androguard: Github repository. https://github.com/androguard/androguard.

Android Statistics (2017). Android statistics marketers need to know. http://mediakix.com/2017/08/android-statistics-facts-mobile-usage/.

Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., and Rieck, K. (2014). DREBIN: effective and explainable detection of android malware in your pocket. In *Proceedings of the 2014 Network and Distributed System Security Symposium*, NDSS 2014. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/11_3_1.pdf.

Aung, Z. and Zaw, W. (2013). Permission-based Android malware detection. *International Journal of Scientific & Technology Research*, 2(3).

Bennett, K. P. and Campbell, C. (2000). Support vector machines: Hype or hallelujah? *SIGKDD Explorations*, 2(2):1–13.

Breiman, L. and Cutler, A. (2001). Random forests^{TM}. https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm.

Damodaran, A., Troia, F. D., Visaggio, C. A., Austin, T. H., and Stamp, M. (2017). A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques*, 13(1):1–12.

Dimjašević, M., Atzeni, S., Ugrina, I., and Rakamarić, Z. (2015). Evaluation of android malware detection based on system calls. Technical Report UUCS-15-003, School of Computing, University of Utah. http://www.cs.utah.edu/docs/techreports/2015/pdf/UUCS-15-003.pdf.

Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. (2010). TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 393–407. USENIX Association.

Feng, Y., Anand, S., Dillig, I., and Aiken, A. (2014). Apposcopy: Semantics-based detection of Android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 576–587.

Fuchs, A. P., Chaudhuri, A., and Foster, J. S. (2009). SCanDroid: Automated security certification of Android applications. https://www.cs.umd.edu/~avik/papers/scandroidascaa.pdf.

Huang, W., Troia, F. D., and Stamp, M. (2018). Robust hashing for image-based malware classification. In *ICETE (1)*, pages 617–625. SciTePress.

Internet Archive (2018). Internet Archive. https://archive.org.

Kapratwar, A., Troia, F. D., and Stamp, M. (2017). Static and dynamic analysis of android malware. In *Proceedings of the 1st International Workshop on Formal Methods for Security Engineering*, ForSE 2017, in conjunction with the 3rd International Conference on Information Systems Security and Privacy (ICISSP 2017), pages 653–662. SciTePress. http://www.scitepress.org/DigitalLibrary/PublicationsDetail.aspx?ID=mI9FBvhgap4=&t=1.

Landwehr, N., Hall, M., and Frank, E. (2005). Logistic model trees. *Machine Learning*, 59(1-2):161–205.

Lin, D. and Stamp, M. (2011). Hunting for undetectable metamorphic viruses. *Journal in Computer Virology*, 7(3):201–214.

Malware Forecast (2017). Malware forecast: The onward march of Android malware. https://nakedsecurity.sophos.com/2017/11/07/2018-malware-forecast-the-onward-march-of-android-malware/.

PlayDrone (2018). PlayDrone: A measurement study of Google Play. https://systems.cs.columbia.edu/projects/playdrone/.

Quinlan, R. (2018). Software available for download. http://www.rulequest.com/Personal/.

Schmeelk, S., Yang, J., and Aho, A. (2015). Android malware static analysis techniques. In *Proceedings of the 10th Annual Cyber and Information Security Research Conference*, CISR '15, pages 5:1–5:8, New York, NY, USA. ACM.

Stamp, M. (2017a). Boost your knowledge of adaboost. https://www.cs.sjsu.edu/~stamp/ML/files/ada.pdf.

Stamp, M. (2017b). *Introduction to Machine Learning with Applications in Information Security*. Chapman and Hall/CRC, Boca Raton.

Stamp, M. (2018). Deep thoughs on deep learning. https://www.cs.sjsu.edu/~stamp/ML/files/ann.pdf.

Sugunan, K., Gireesh Kumar, T., and Dhanya, K. A. (2018). Static and dynamic analysis for android malware detection. In Rajsingh, E. B., Veerasamy, J., Alavi, A. H., and Peter, J. D., editors, *Advances in Big Data and Cloud Computing*, pages 147–155, Singapore. Springer Singapore.

Yajamanam, S., Selvin, V. R. S., Troia, F. D., and Stamp, M. (2018). Deep learning versus gist descriptors for image-based malware classification. In *ICISSP*, pages 553–561. SciTePress.

Zhou, Y. and Jiang, X. (2012). Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, Washington, DC, USA. IEEE Computer Society.

Zhou, Y., Wang, Z., Zhou, W., and Jiang, X. (2012). Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *19th Annual Network and Distributed System Security Symposium*, NDSS 2012.