2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing

Random Forest Classification for Detecting Android Malware

Mohammed S. Alam
and Son T. Vuong
Department of Computer Science
University of British Columbia
Vancouver, BC V6T1Z4
Email: malam@cs.ubc.ca vuong@cs.ubc.ca

***Abstract*—Internet connected smartphone devices play a crucial role in the application domain of Internet of Things. These devices are being widely used for day-to-day activities such as remotely controlling lighting and heating at homes, paying for parking, and recently for paying for goods using saved credit card information using Near Field Communication (NFC). Android is the most popular smartphone platform today. It is also the choice of malware authors to obtain secure and private data. In this paper we exclusively apply the machine learning ensemble learning algorithm *Random Forest* supervised classifier on an Android feature dataset of 48919 points of 42 features each. Our goal was to measure the accuracy of Random Forest in classifying Android application behavior to classify applications as malicious or benign. Moreover, we wanted to focus on detection accuracy as the free parameters of the Random Forest algorithm such as the number of trees , depth of each tree and number of random features selected are varied. Our experimental results based on 5-fold cross validation of our dataset shows that Random Forest performs very well with an accuracy of over 99 percent in general, an optimal Out-Of-Bag (OOB) error rate [3] of 0.0002 for forests with 40 trees or more, and a root mean squared error of 0.0171 for 160 trees.**

I. Introduction

Internet connected smartphones play an important role in our day to day communication needs. They are not only used for traditional cost incurring activities such as long distance phone calls and SMS, but for a variety of other tasks that contribute to the domain of Internet-Of-Things. Today, smartphones are used to browse personal and corporate accounts on social networks such as Facebook and Twitter; conducting monetary activities such as paying for goods and services via credit card information saved on mobile devices using NFC [19], paying for car parking using pay by phone applications [20]; and are used by security conscious users as 2-factor authentication devices for banking, email and cloud repositories such as Dropbox and Gmail. As such, malware authors have begun focussing their interest on coding malicious application for smartphones similar to what has been done for personal computing devices over the last three decades. Smartphones can now be infected by virus, SMS trojans [13], and be infected to be made part of a botnet to perform Distributed Denial of Service (DDoS) attacks [21] on websites.

The most popular smartphone platform today is the Android platform developed by Google. It is also the current choice of malware authors. In Android, each application has an associated .apk file which is synonymous to a .exe file on the Windows platform. Due to the open software installation nature of Android, users are allowed to install any executable file from any application store. This could be from the official Google Play store [17], or a third party site. This ease of installing applications compared to other smartphone platforms such as Apple's iOS [22] platform makes Android users vulnerable to malicious applications. Moreover, unlike the iOS platform, Google does not verify if applications in their official Play store have malicious intent prior to making it publicly available. As such, malware has also been found in the official Google Play store applications by security vendors [18].

Classification techniques such as Support Vector Machines, K-Nearest Neighbours, Decision Trees, Logistic Regression and Naive Bayes have widely been used in the area of intrusion detection research in the security community. They are predominantly used for behavior based detection methods, also called anomaly detection methods. The purpose of this paper was to identify the feasibility of using Random Forest classification to detect if an Android device has been compromised by malware by inspecting application behavior data. In this paper we focus on training a classifier based on an offline analysis of Android application behavior. Once trained, this technique could then be used to analyze new Android applications prior to deployment on a real Android device. This could be done by testing computed feature vectors collected during run-time of the application on an Android Virtual Device.

For experimentation, we used a modified dataset made available by Amos [2] under the Creative Commons Attribution 3.0 Unported License downloaded in Feb 28, 2013. This dataset was obtained from emulating user action using adb-monkey [8]. More information regarding the dataset is provided in section III-A. We focus on the detection accuracy of Random Forest as the number of trees, depth of each tree and number of random features selected are varied for the algorithm. We perform a 5-fold cross validation on our dataset for the error estimate.

According to our knowledge, we have not come across any other paper that has exclusively worked with the free parameters of Random Forest algorithm on an Android feature dataset. The main contributions of this paper can be summarized as follows:

- Apply Random Forest classification algorithm for behavior detection on features [2] obtainable from an unrooted Android devices by predominantly monitoring

978-0-7695-5046-6/13 $26.00 © 2013 IEEE
DOI 10.1109/GreenCom-iThings-CPSCom.2013.122

features based on the Android Binder API, Memory and CPU measurements to detect Android malware.

- Conduct detailed experimentation to measure the accuracy of Random Forest classifier as the number of trees and the depth of each tree in the forest is varied for different size of random features selected.

The rest of the sections are organized as follows. Section II provides relevant background to understand the problem domain. In section III we describe our experimental approach. Section IV discusses our results, the advantages and disadvantages. Section V describes future direction related to this paper.

II. Background

In order to understand the application domain of this paper, we now discuss three related areas: feature collection for Android operating system, key features of the Random Forest algorithm, and application of machine learning in the domain of mobile security.

A. Android Feature Collection

In order to apply any machine learning classifier, it is important to first be able to collect relevant features that can be observed from the system. The features that the Android system allows access permissions to depends on if the device has been rooted. Rooting is the process whereby one has privileged access control to the system. As mentioned in Android Security Overview [1], Android uses the Linux Kernel as the bottom layer. All layers on top of the kernel layer run without privileged mode. i.e. All applications and system libraries are inside a virtual application sandbox. Thus, applications are prohibited from accessing other application data (unless explicitly granted permission by other applications). As per [1], each Android application is assigned a unique user ID (UID) which runs as that user in a separate process. Thus, if a feature vector is created from features of Android API in unrooted mode, then only system information made available by Android can be used. On the other hand, having a rooted device allows one to install system tools that could gather features from underlying host and network behavior. Example of features that can be obtained from rooted devices include: data being sent by applications, IP addresses being communicated with in the network, number of active connections, the system calls being invoked, etc. The related work in the area of machine learning approaches to Android data either deal with feature collection from unrooted devices [2], [7], or rooted devices [4], [5], [6].

B. Random Forest

Random Forest is an ensemble learning algorithm developed by Breiman [3]. An ensemble learner method generates many individual learners and aggregates the results. Random Forest uses an extension to the bagging approach. In Bagging, each classifier is built individually by working with a bootstrap sample of the input data. In a regular decision tree classifier, a decision at a node split is made based on all the feature attributes. But in Random Forest, the best parameter at each node in a decision tree is made from a randomly selected number of features. This random selection of features helps Random Forest to not only scale well when there exists many features per feature vector, but also helps it in reducing the interdependence (correlation) between the feature attributes [15] and is thus less vulnerable to inherent noise in the data.

As mentioned by the author [3], the number of random features m selected per decision node in a tree decides the error rate of the forest classification. The error rate of the Random Forest classifier depends on the *correlation* between any two trees, and the classification *strength* of each individual tree. Reducing the random features selected m causes reduction in both the correlation between classification trees and the strength of classification of each individual tree. Increasing m increases both the correlation between the trees and the strength of each tree. Breiman explains that the Out-of-Bag (OOB) error rate is an indication of how well a forest classifier performs on the dataset. The out-of-bag model leaves out one-third of the input dataset for building the kth tree from the bootstrap sample for each tree. This one-third sample is used to test the kth tree and the results of misclassification averaged over all trees. The author claims that for most cases OOB error estimate is a good estimate of the error and hence cross validation or separate test set is usually unnecessary when using the Random Forest algorithm [23].

C. Machine Learning Classifier for Android Mobile Systems

Application of machine learning methodologies for classification of Android malware is currently an emerging area of interest. As per literature, malware detection can be done either via static detection or dynamic detection. Static detection techniques (also called signature matching) have high detection rates and require less computational resources. This is the traditional approach taken by anti-virus software. Dynamic detection techniques (also called anomaly/behavior detection) on the other hand, suffer from low detection rates and require more resources to reduce the amount of false positives reported. They usually are prone to high false positives without adequate training. We now look at related approaches to applying machine learning classifiers on Android-based data.

Kim et al. [4] developed an automatic feature extraction tool implemented in JavaScript for static detection. The input to their system is a .apk Android executable file. The authors use JavaScript code to extract the following as features: Number of permissions requested by the application (gathered from the manifest.xml file in Android .apk package file); API count of each method related to phone management invoked from the code; API count of each method related to phone control and privacy information called from the code. Based on these features collected, the authors use *J48 Decision Tree* classifier from the Weka library and perform 10 fold cross validation on 1003 Android applications. They received a true positive rate of 82.7 percent and false negative rate of 17.3 percent.

Burguera et al. [5] used a rooted Android device. The Linux based tool *strace* was used to capture system calls. The authors captured the number of each system call invoked by a given Android application to form the feature vector. Each feature vector was composed of 250 linux 2.6.23 system calls. Based on experimental results, the most relevant system calls

for malware detection were *read(), open(), access(), chmod() and chown()*. The authors used a simple 2-means clustering algorithm to distinguish between benign applications and their corresponding malware version. The distance between clusters is just a Euclidean distance between the feature vectors. Their solution would cluster similarly named applications into two different clusters, one for the malware and the other for the benign sample. The solution works under the assumption that a benign sample is re-fitted with malicious code before being uploaded. Thus, a malware without a benign version cannot be detected. One weakness of the system is that their tool cannot detect malware that uses very few system calls, as was exhibited in Monkey Jump 2 malware.

Dini et al. in [6] used a rooted device to design a host based real time anomaly detector based on 1-Nearest Neighbour classifier. They monitor 13 features. At user-level 2 features are monitored: if the phone is active/inactive; and if SMS is being sent when the phone is inactive. At the kernel level, they created a kernel module to monitor 11 system calls: *open, ioctl, brk, read, write, exit, close, sendto, sendmsg, recvfrom, recvmsg*. They used 2 models, one which captures the features every 1 second interval, and another which captures every 1 minute interval.They tested their system on 10 genuine malware samples. The authors claim a malware detection rate of 93 percent in general with a false positive rate of 0.0001. Unlike other systems that monitor features per application, their model monitors system wide behavior.

Shabtai et al. in [7] monitored 88 features on an unrooted device. They monitor the features on 2 real devices used by 2 different users. They tested their approach on 16 benign applications and 4 self generated malware. Their model monitors features at 2 second intervals. The authors compared the use of the following classifiers: k-Means, Logistic Regression, Histograms, Decision Tree, Bayesian Networks and Naive Bayes. The authors uses a *filter* approach for feature selection by comparing Chi-Square, Fisher Score and Information Gain. They used top 10, 20, 50 features for computing the scores for each feature out of the 88 features available. Based on their evaluation, they have a detection rate of 80 percent. Their false positive rate is 0.12. They claim that Naive Bayes and Logistic Regression were superior for most configurations. Fisher score with top 10 features scored the best among their experimentation. The authors identified the following features as being of importance in distinguishing malware and benign applications: *Anonymous_pages, Garbage_Collections, Battery_Temp, Total_Entries, Active_Pages, and Running_Processes*.

Amos in [2] created an automated system to analyze malware samples. The tool allows for the automated analysis of benign and malicious Android applications by running adb scripts in Linux systems. Since we used data gathered by [2] for our experiments, section III-A provides more details about the dataset. The author compared results from using the following classifiers: Bayes Net, J48 decision tree, Logistic Regression, Multilayer Perceptron, Naive Bayes and Random Forest. We would like to point out that Random Forest was only tested with the default setting of 10 trees and 6 parameters. The author did not test the system by varying the number of trees, depth of each tree and the number of randomly chosen parameters. Based on the cross validation results provided, Random Forest already performed better than other classifiers even with the default settings. Since no description of the test results are available, our analysis is based on inspecting the cv-full.txt results file posted by the author.

Z. Aung and W. Zaw [24] performed a static analysis of 500 Android .apk files by inspecting the permissions requested by a given application. They used 160 permissions as their feature vector. They compared Random Forest, J48 Decision Tree, and Classification and Regression Tree (CART) algorithms. Unfortunately they do not discuss the parameter setting used for Random Forest, nor the number of benign and malicious applications used for experimentation.

III. Experiment

A. Dataset Description

For this experiment, we used a slightly modified dataset provided by [2]. The author developed a shell script to automatically analyze .apk Android application files by running them in available Android emulators. For each .apk file, the emulator simulates user interaction by randomly interacting with the application interface. This is done using the Android adb-monkey tool [8]. Based on inspection of the source code, we can conclude that each feature vector of the dataset is collected at 5 second intervals. The memory features were collected by observing the"proc" directory in the underlying Linux system of Android. The CPU information was collected by running the Linux "top" command. The Battery and Binder information was collected by using an "intent" object. The "permissions" feature is the summation of the total permissions requested by each running application (package) obtained from the *PackageManager* class in Android.

The author provides dataset [1] based on observing 1330 Malicious apk samples and 407 benign samples. This is based on observation of feature vectors collected for each application. The original dataset had a total of 32342 data (feature vector) samples with 7535 benign samples (classified as positive class) and 24807 malicious samples (classified as negative class). A classifier becomes more biased towards the oversampled class if there is a class imbalance. In order to balance the two classes, we used the SMOTE package from Weka to create more data points of the under-sampled class (benign class). We then applied the random filter to randomize the distribution of the two classes in the data file used. This is because the SMOTE package generates data points of the under sampled class. Randomization was required as SMOTE adds data entries to the end of the data file. This would be a problem during cross validation when the folds are computed as we will have points primarily from the under sampled class in the last few folds. Performing this manipulation to the original dataset provided us 48919 data samples with 24,112 samples of the positive class and 24807 samples of the negative class. For our experimentation we used this new sample. Since the exact versions of the .apk files used for generating the initial dataset was not available, we modified the feature vector dataset generated by [2].

Based on observation of the original dataset, and the collection strategy used, we see a few limitations. The most notable areas are in the battery data features and the network

[1] As of Feb 28, 2013.

TABLE I. ANDROID FEATURES OBSERVED

Category	Feature
Battery	IsCharging
	Voltage
	Temperature
	BatteryLevel
	BatteryLevelDiff
Binder	Transaction
	Reply
	Acquire
	Release
	ActiveNodes
	TotalNodes
	ActiveRef
	TotalRef
	ActiveDeath
	TotalDeath
	ActiveTransaction
	TotalTransaction
	ActiveTransactionComplete
	TotalTransactionComplete
	TotalNodesDiff
	TotalRefDiff
	TotalDeahDiff
	TotalTransactionDiff
	TotalTransactionCompleteDiff
CPU	CPU Usage
Memory	memActive
	memInactive
	memMapped
	memFreePages
	memAnonPages
	memFilePages
	memDirtyPages
	memWritebackPages
Network	TotalTXPackets
	TotalTXBytes
	TotalRXPackets
	TotalRXBytes
	TXPacketsDiff
	TXBytesDiff
	RXPacketsDiff
	RXBytesDiff
Permission	Permission

features. Given that these data samples were collected by running in an emulator, there were no observed change in battery data among any of the feature vectors for both classes. Similarly, the Network data was fixed for all the data points. This would mean that 13 of the observed 42 features do not contribute to classification as the 13 features will not contribute to information gain (reduction in entropy) score used by Random Forest algorithm at each decision node. The features are however correctly computed on a real device. Testing over 1700 applications on a real device in a timely manner would be prohibitively expensive, hence the provided dataset was used. The list of 42 features monitored by [2] are listed in Table I.

B. Software

We used the Random Forest classifier in the experimental Weka 3.7.9 software for an empirical analysis of the Random Forest algorithm. We chose the experimental release of Weka over the stable release 3.6.9 due to the multi-threading support in the experimental release for the Random Forest algorithm. This reduces the time complexity of each experimental run from hours to minutes for cross-fold experiments with large number of trees. The Random Forest classifier implementation for Weka does not implement feature importance computation. Availability of this feature would have helped in observing the features that are weighted higher by Random Forest algorithm. Hence, we have not computed feature importance.

C. Hardware

The experiments were run on a laptop with the Intel Core-i7 3630QM 2.4 GHz processor with 8GB of RAM. The amount of RAM dedicated for Weka was increased to 4GB for computing large number of trees. We used 8 threads for each computation.

D. Results

For experimentation, we run tests for each of the Number of Trees , Depth of Tree and Attribute combination. We tested with 10, 20, 40, 80 and 160 trees for each Random Forest. For each setting of trees we tested with 4, 6, 8, 16 and 32 randomly selected attributes for each decision node. For each Tree and attribute combination, we tested with 1, 2, 4, 8, 16 and 32 depth trees. Table II shows the results when each tree is allowed to grow to maximum depth. We performed 5-fold cross validation for each experiment.

Following are the parameters that we measured as shown in Table II:

- *OOB Error*
  This is the Out-Of-Bag error explained in section II-B.
- *Root MSE*
  The square root of the mean squared error based on 5-fold cross validation.
- *% True Class*
  This value shows the percentage of the samples that were classified correctly.
- *# Incorrect*
  This number shows the total number of the 48,919 samples that were misclassified. We evaluate this number to show the minor variations that happen as the number of trees and random features selected change.

As shown in Table II, the best Root Mean Squared Error value of 0.0171 is achieved with 160 trees and 16 features. The best setting based on the lowest number of misclassifications (7 out of 48,919) was achieved with 160 trees and 8 features selected. These have been highlighted in Table II.

Fig. 1 provides comparison of how the depth of tree impacts the number of inaccurate classifications. In this figure we show results of a Random Forest with 20 trees. As can be seen, regardless of the number of features selected, the number of incorrect classifications stabilizes at a tree depth of 16. Similar behavior is observed for other size of forests, i.e. forests with 10, 40, 80 and 160 trees. We omit them here for brevity.

TABLE II. EXPERIMENTAL RESULTS FOR VARYING NUMBER OF TREES AND NUMBER OF RANDOM FEATURES WITH TREES ALLOWED TO GROW TO MAXIMUM DEPTH

Trees	Features	OOB Err	Root MSE	% True Class	#Incorrect
10	4	0.0067	0.0291	99.9469	26
10	6	0.0064	0.0245	99.9407	29
10	8	0.0058	0.0221	99.9600	19
10	16	0.0056	0.0206	99.9600	19
10	32	0.0056	0.0200	99.9600	19
20	4	0.0008	0.0259	99.9670	16
20	6	0.0006	0.0223	99.9632	18
20	8	0.0005	0.0208	99.9693	15
20	16	0.0004	0.0191	99.9652	17
20	32	0.0006	0.0184	99.9693	15
40	4	0.0003	0.0242	99.9693	15
40	6	0.0004	0.0210	99.9734	13
40	8	0.0002	0.0197	99.9755	12
40	16	0.0003	0.0178	99.9612	19
40	32	0.0003	0.0178	99.9714	14
80	4	0.0004	0.0239	99.9734	13
80	6	0.0002	0.0203	99.9775	11
80	8	0.0002	0.0187	99.9836	8
80	16	0.0002	0.0175	99.9734	13
80	32	0.0003	0.0178	99.9673	16
160	4	0.0003	0.0233	99.9775	11
160	6	0.0002	0.0201	99.9755	12
160	8	0.0002	0.0183	99.9857	**7**
160	16	0.0002	**0.0171**	99.9734	13
160	32	0.0002	0.0175	99.9693	14

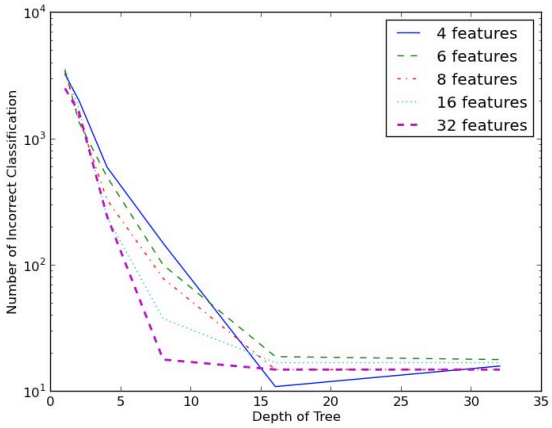

Fig. 1. Misclassification comparison with 20 trees as depth of tree is varied

Fig. 2 provides comparison of how the depth of the tree impacts the out-of-bag error rate for Random Forest with 40 trees. As can be seen from the figure, low error rates are achieved quickly if 16 and 32 features are selected at tree depth of 8. If lower number of features are selected, then a tree depth of 16 is required before the error rate stabilizes. Similar behavior is observed for other forest sizes for the given tree depth and number of random features combination.

Fig. 3 provides observed root mean squared error rates as the depth of the tree is observed at 1, 2, 4, 8, 16 and 32 for a Random Forest with 80 trees. Stable value of the error rate is observed at a tree depth of 8 when 32 features are observed. A depth of 16 is required for all other observed feature counts. We observe similar pattern for other tree sizes.

In order to compare the results obtained by applying the Random Forest algorithm to other classification techniques,

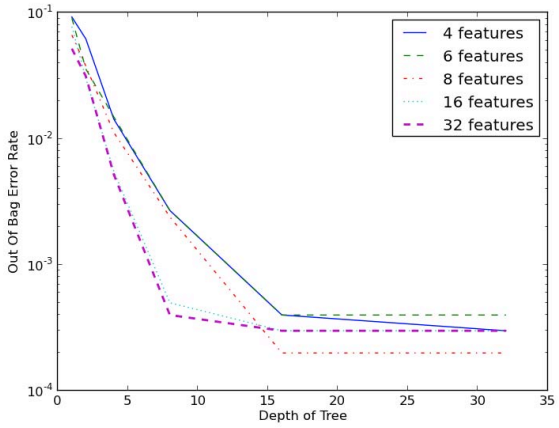

Fig. 2. Out Of Bag error rate comparison with 40 trees as depth of tree is varied

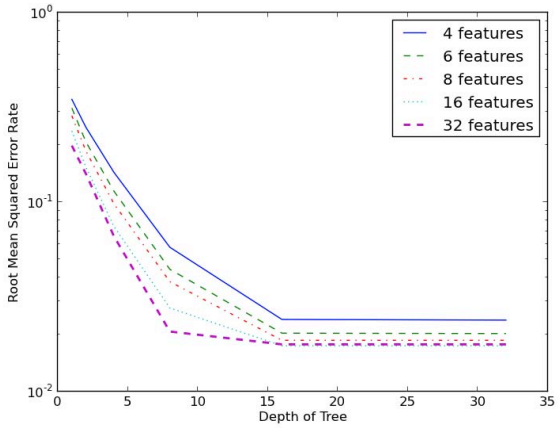

Fig. 3. Root Mean Squared error rate comparison with 80 trees as depth of tree is varied

TABLE III. MISCLASSIFICATION COMPARISON OF DIFFERENT CLASSIFIERS

Algorithm	Misclassification
BayesNet	342
NaiveBayes	2361
MultilayerPerceptron	38
J48	47
Decision Stump	2532
Logistic Regression	323
Random Forest	7

we perform preliminary classification tests on our dataset with default settings for classifiers in Weka. The number of misclassified instances of the 48919 points are as shown in Table III. It should be pointed out that all the algorithms listed have associated free parameters which can possibly be tuned to obtain better results. We consider it outside the scope of this paper as our focus is on testing the parameters of Random Forest for our dataset.

IV. Discussion

Based on our experimental results on 5-fold cross validation, we can make the following observations for the Android features evaluated:

- **High Accuracy of Random Forest:** Random Forest provides an exceptionally high accuracy with over 99.9 percent of the samples correctly classified when trees are allowed to grow to maximum depth. The square-root of the mean-squared-error is 0.0291 or less. The out-of-bag error estimate of Random Forest is acceptably low with 40 trees or higher. It varies between 0.0002 and 0.0004. The best setting based on root MSE value was using 160 trees and 16 random features selected with a Root MSE value of 0.0171. The best setting based on number of incorrect classifications was 160 trees with 8 random features selected with a score of 7 incorrect classifications.
- **More trees are better:** The overall trend shows that the out-of-bag error reduces on average as the number of trees increase. For our data sample, we observe significant better out-of-bag error rates when we have at least 20 trees for the varying random features selected from 4 to 32.
- **Depth of tree required:** Based on experimentation, we observe that for our dataset, we need to construct trees of depth 16. Lower depth than this causes higher number of inaccurate classifications. Measuring trees of depth greater than 16 does not cause a statistically significant change for a given number of trees in the Random Forest algorithm for our dataset. Random Forest provides an accuracy of over 91 percent with a tree depth of 1. With 4 features selected, Random Forest provides greater than 99 percent classification with a depth of 8. With 6 or more features selected, a 99 percent correct classification can be achieved with a depth of 4 for each tree.
- **Lower features per tree better:** For a given forest size, after a certain point, we see that as the number of features are increased for a given tree, the number of incorrect classifications increase. The author in [3] had mentioned that ideally choosing log M features (where M is the size of the total attributes) would yield a good result. In our case, choosing 6 or 8 attributes for a forest gives ideal results for a particular forest size.
- **Misclassified malicious samples:** Even though we do not show it in our results table, for all experiments, most of the misclassified cases were due to the malicious class samples misclassified as being of the benign class i.e. there were more false negatives than false positives. For example the experimental setting yielding the best result with a forest with 160 trees, 8 random features, and depth of 16 for each tree, we observed 5 false negatives and 2 false positives. This is based on observing the generated confusion matrix for each experiment case.

Even though the current dataset yields very promising results, its applicability for real-time monitoring of malware infections on Android devices would be difficult as is. The current detection technique would be promising in the scenario where the Android .apk executable application file is run in a secluded environment prior to deployment on a real device, or prior to being made publicly available. We have to understand that the features observed for the dataset are global in nature i.e. All concurrently running applications together impact the features observed. This would cause the data features measured to be very noisy at best without being able to individually break down the impact of each application on the measured features. Similar issues can be identified with [7] as feature vectors were collected by running a single Android application at a time on a real device. One approach we envision would be to normalize the measured feature changes by fine grained inspection of currently running applications. This information is currently allowed to be queried by the Android API. The dataset that we used was collected by running adb-Monkey [8] by Amos[2] on benign and malicious apk sample on the Android emulator. All the features monitored by [2] are a subset of those collected by [7]. An appropriate extension would be to compare our results for Random Forest with the dataset from [7] which was collected on a real device.

V. Conclusion and Future Work

As discussed in this paper, many classification approaches from machine learning have been applied to deduce the right strategy to detect infections by malware on mobile devices. Similar to detecting malware infections on PCs, both signature-based and behavior (Anomaly) based approaches are being tried on mobile devices.

In this paper we have tried to deduce if the machine learning classifier, *Random Forest* can be used to detect malware infections. The dataset used was gathered from [2]. The dataset was generated automatically by monitoring a set of 42 attributes as listed in section III-A by running the Android adb-monkey tool [8]. In our experiments, based on a 5-fold cross validation, Random Forest provided an exceptionally high accuracy of over 99.9 percent of the samples correctly classified (see table II). The optimal square-root of the mean-squared-error achieved was 0.0171. The optimal out-of-bag error rate obtained was 0.0002 with a minimum forest size of 40 trees.

As malware in the Android mobile setting is constantly adapting to detection techniques, many of the detection models described here and in the referenced literature in section II have to be tuned to observe new behavior patterns. We now list some future directions that can be taken to refine this work further.

A. Bayesian Optimization for Parameter selection

Bayesian optimization [14], [16] is a promising approach for automatically adjusting free parameters in any given algorithm. In the case of Random Forest, the free parameters include: the number of trees in the forest, the number of random features selected at each decision node of a tree, and the maximum depth of each tree. We could use an objective function that uses the out-of-bag error estimate to guide the Bayesian Optimization algorithm into selecting the right combination of parameters that provide us acceptable settings based on our required threshold.

B. Observing More features

The dataset we used primarily focused on the binder API of Android, CPU usage pattern, and memory usage pattern on an Android device emulator. Many papers listed in the related work section used system call tracking as features. Random Forest is relatively immune to increase in the number of features of a feature vector as it requires to observe only log *m* features of the available *m* features. Thus, monitoring system call features and more fine grained network level behavior would allow us to create better detection models.

C. Ensemble Learning

One approach that we intend on trying in the future is to use ensemble learning [9][10]. Ensemble learning creates a learning model by integrating the results of multiple models. Currently we are monitoring features on Android devices that are not rooted. Given that mobile devices can be rooted, this would allow better feature collection such as features bases on network data, permissions and system calls. We could create an ensemble of weak learners some of which monitor device at the user level (in case of unrooted devices) and others that monitor features at the kernel level (in case of rooted devices). For example, we could create a learning model for *tcpdump* data [11] to monitor network traffic and another to monitor system call data using *strace*[12].

A foreseeable challenge in this case would be to see how good a classification technique would be in the absence of observed data during testing time. This would be the case if some of the features cannot be observed if a device is unrooted; or a tool for capturing the behavior is missing in a rooted device. Similar issues will occur if Google decides to remove access to API used for observing behavior at user level.

ACKNOWLEDGMENT

Mohammed Alam would like to thank Dr. Nando De Freitas for providing valuable insight on machine learning techniques taught by Dr. Freitas during the graduate course at UBC. We would also like to thank the reviewers who provided valuable feedback on the initial version of this paper.

REFERENCES

[1] Google. Android Security Overview. Internet: http://source.android.com/tech/security, [May 15, 2013].

[2] B. Amos. "Antimalware". Internet: https://github.com/VT-Magnum-Research/antimalware, [May 15, 2013].

[3] L. Breiman. "Random Forests". *Machine Learning*, 45 (1): 532. doi:10.1023/A:1010933404324, 2001.

[4] D. Kim, J. Kim, and S. Kim. "A Malicious Application Detection Framework using Automatic Feature Extraction Tool on Android Market" in *3rd International Conference on Computer Science and Information Technology (ICCSIT'2013)* January 4-5, 2013.

[5] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. "Crowdroid: Behavior-Based Malware Detection System for Android". in *Proc. SPSM*, 2011

[6] [6] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra, "MADAM: a Multi-Level Anomaly Detector for Android Malware". in *Proc. MMM-ACNS*, 2012

[7] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weisee. "Andromaly: a behavioral malware detection framework for android devices". *Journal Intelligent Systems*, 38: pp.161-190, 2012.

[8] Google. "UI/Application Exerciser Monkey". Internet: http://developer.android.com/tools/help/monkey.html, [May 15, 2013].

[9] L. Kuncheva, and C. Whitaker, "Measures of diversity in classifier ensembles", *Machine Learning*, 51, pp. 181-207, 2003.

[10] L. Rokach, "Ensemble-based classifiers". *Artificial Intelligence Review*, 33 (1-2): 139. doi:10.1007/s10462-009-9124-7, 2010.

[11] M. Cerdano, "Analyzing Android Network Traffic". Internet: http://mobile.tutsplus.com/tutorials/android/analyzing-android-network-traffic/, [May 15, 2013].

[12] Android Tools. Internet: http://elinux.org/Android_Tools, [May 15, 2013].

[13] M. Parkour, Internet: http://contagiominidump.blogspot.ca/, [May 15, 2013].

[14] Ziyu Wang, Masrour Zoghi, David Matheson, Frank Hutter, and Nando de Freitas. "Bayesian Optimization in a Billion Dimensions via Random Embeddings", *International Joint Conference on Artificial Intelligence (IJCAI)*, Technical Report arXiv:1301.1942, 2013.

[15] Antonio Criminisi, Jamie Shotton, and Ender Konukoglu. "Decision Forests: A Unified Framework for Classification, Regression, Density Estimation, Manifold Learning and Semi-Supervised Learning". *Foundations and Trends in Computer Graphics and Vision*. Vol. 7: No 2-3, pp 81-227, 2012.

[16] J. Snoek, H. Larochelle, and R. Adams. "Practical Bayesian Optimization of Machine Learning Algorithms". *Advances in Neural Information Processing Systems*, arXiv:1206.2944v2, 2012.

[17] Google. Internet: http://play.google.com, [May 15, 2013].

[18] Trend Micro. Internet: http://countermeasures.trendmicro.eu/android-malware-believe-the-hype, [May 10,2013]

[19] Dewan, Sunil G., and L-D. Chen. "Mobile payment adoption in the USA: a cross-industry, crossplatform solution." *Journal of Information Privacy & Security*, 1.2, pp. 4-28, 2005.

[20] PayPoint Inc. Internet: http://www.paybyphone.com/, [May 15, 2013].

[21] "China mobile users warned about large botnet threat", BBC, (Jan 15, 2013). Internet: http://www.bbc.co.uk/news/technology-21026667

[22] Apple Store. Internet: http://store.apple.com, [May 10, 2013].

[23] L. Breiman, and A. Cutler. "Random Forests". Internet: http://www.stat.berkeley.edu/ breiman/RandomForests/cc_home.htm, [May 10, 2013].

[24] Z. Aung, and W. Zaw. "Permission-Based Android Malware Detection." *International Journal of Scientific and Technology Research*, Vol. 2, Issue 3, March 2013.