High Precision Screening for Android Malware with Dimensionality Reduction

Britton Wolfe
Information Analytics and Visualization Center,
Indiana Univ.-Purdue Univ. Fort Wayne (IPFW),
2101 E. Coliseum Blvd., Fort Wayne, IN 46825 USA
Email: wolfeb@ipfw.edu

Karim Elish Danfeng Yao
Department of Computer Science,
Virginia Tech,
2202 Kraft Dr., KWII, Blacksburg, VA 24060 USA
Email: kelish@vt.edu and danfeng@vt.edu

***Abstract*—We present a new method of classifying previously unseen Android applications as malware or benign. The algorithm starts with a large set of features: the frequencies of all possible n-byte sequences in the application's byte code. Principal components analysis is applied to that frequency matrix in order to reduce it to a low-dimensional representation, which is then fed into any of several classification algorithms. We utilize the implicitly restarted Lanczos bidiagonalization algorithm and exploit the sparsity of the n-gram frequency matrix in order to efficiently compute the low-dimensional representation. When trained upon that low-dimensional representation, several classification algorithms achieve higher accuracy than previous work.**

I. Background

The Android operating system continues to gain market share among smart phone users across the world. At the end of 2013, it had reached over 50% market share in the United States and Great Britain and over 70% in Germany and China [1]. In all four countries, Android gained more than 4% market share over the previous year. With an increase in market share also comes an increase in the attention of malware developers. There are hundreds of malicious applications in the official and alternative Android marketplaces [2]. This work presents a new way of detecting malicious Android applications, resulting in higher accuracy than previous methods.

Our technique starts with n-gram frequencies, which can be extracted from the raw data without any domain knowledge. For example, the frequency of the 3-gram `3F:52:BA` would be the number of times that that 3-byte sequence occurred in the byte code for the application. In contrast, domain-specific features (e.g., user-trigger dependence relations [3]) require manual effort to develop, as they vary significantly for different malware. Domain-specific features highlight empirically observed behaviors or patterns of specific types of malware, whereas n-grams are raw, low-level features. Our primary contribution is a new way to represent Android applications for classification: applying dimensionality reduction on the n-gram frequencies. The resulting low-dimensional representation is very useful for identifying malware, letting us achieve a higher classification accuracy than Android work with domain-specific features.

A. Malware Classification

Traditional methods for detecting malware rely upon recognizing a specific signature that has been previously identified as belonging to a specific, known malware. A limitation of this approach is that it cannot recognize previously unknown malware. In contrast, heuristic-based or machine learning methods generalize from samples of malware and clean files, allowing them to recognize previously unknown malware.

Each application is represented as a vector of feature values, which can come from dynamic analysis or static analysis. Dynamic analysis involves running the application in a sandbox and recording information about its behavior, such as battery and network usage [4]. Static analysis uses features extracted without running the application, such as the permissions that the application requests upon installation or information about the control flow of the program (e.g., [2]). Both types of analyses are useful and provide complementary insights about applications' behaviors. Our work uses static analysis.

B. Related Work: General Security

Machine learning techniques have been widely adopted in the computer security literature since the work by Lee et al. [5]. Equipped with domain knowledge, the methods extract *domain specific* features based on empirical observations of malicious programs or traffic patterns.

For example, solutions described by Cova et al. [6] use binary classification techniques to identify malicious Javascript code on the web. The features they extracted from malicious code include the presence of redirection and obfuscation. Xie et al. [7] used a Bayesian network to infer abnormal network traffic patterns. Besides classifying programs and network traffic, learning-based security research also includes database intrusion detection [8] and SMS/social network spam detection [9].

Our classification method differs from these domain experts' research in that our classification algorithm is performed on straightforward features (namely n-grams) without any specific domain knowledge. We focus on developing fundamental and general data processing and learning methods, as opposed to the selection of specific domain features. Our high classification accuracy indicates the effectiveness of this *domain-independent feature* approach for Android applications.

Using n-gram frequencies as features has been successful in classifying malware in Windows' portable executable format (PE32) files [10, 11]. However, that research uses a

feature selection algorithm to pick only a few n-grams as the features to pass along to the classification algorithm. Their feature selection algorithm works by independently scoring each feature, using mutual information or Fisher score with respect to the class label. The highest scoring features were then passed along to a classification algorithm.

Instead of individual feature selection, we use dimensionality reduction to process the n-gram frequencies and generate a d-dimensional representation of each application. This improvement lets the algorithm exploit interactions between features that the previously used feature selection algorithms would miss because they score features independently from one another. Furthermore, each of the d dimensions produced by dimensionality reduction can encode information about many original features. Thus, the d features produced by dimensionality reduction will contain more information than simply selecting d individual features and discarding all of the information from the other n-gram frequencies.

C. Related Work: Android Malware

Researchers have applied both static and dynamic approaches to malware detection on Android devices. The approaches differ in the features extracted and the classification algorithms employed, leading to varying degrees of success. The data sets employed by the researchers were also of different qualities, ranging from just a handful of malware that the researchers created themselves up to data sets with hundreds of examples pulled from live marketplaces.

Schmidt et al. [12] used a data set of ELF files. It consisted of approximately 240 malware which targeted Linux systems (i.e., not specifically designed for the mobile ARM architecture), and less than 100 Linux system commands from an Android device. They used static analysis to construct binary features, one for each function called by any file in the data set. That information was extracted using `readelf`. They applied three classifiers (rule inducer, nearest neighbor, and decision tree) to a few subsets of the features. All of their configurations that achieved 80% or higher detection rate (i.e., true positive rate) also suffered a false positive rate over 10%.

Burguera et al. [13] proposed the CrowdDroid system for identifying a specific type of malware: repackaged malware. Repackaged malware is created by taking a benign application and repackaging it with additional malicious code. The CrowdDroid approach uses dynamic features. A central system collects the frequencies of several system calls from several users running the application on different devices. It then uses k-means clustering with $k = 2$ to cluster the results, with the goal of separating the benign instances of the application from the malicious (repackaged) instances. Their experiments used only four author-created malware and two real malware. While CrowdDroid successfully identified all of the author-created malware, it produced a 20% false positive rate on one of the two real malware (the more substantial application of the two).

Shabtai et al. [14] also used a small number of fabricated malware (i.e., four applications) to test their Andromaly system, due to a lack of real malware at that time. They used 88 hand-designed dynamic features, including memory page activity, CPU load, SMS message events, network usage, touch screen pressure, binder information, and battery information, among others. They pared down the features using the information gain and Fisher scores for each individual feature, selecting the features with the best scores. Then they applied several classifiers: decision trees, naïve Bayes, Bayes nets, histograms, k-means, and logistic regression. Even on a synthetic data set, their best configuration—naïve Bayes after using Fisher score to select 10 features—still had over 10% false positive rate, with approximately 88% accuracy.

Later research had the advantage of access to more actual malware. Like Andromaly, Amos et al. [4] used hand-selected dynamic features (e.g., memory, CPU, binder information), but evaluated performance on a larger data set: 1330 malware and 408 benign applications. They compared random forests, naïve Bayes, multilayer perceptrons, Bayes nets, logistic regression, and decision trees. As with the previously mentioned work, their methods suffer from a high false positive rate: over 15% for all of their configurations. Their accuracy was 95% on new traces from applications included in the training set, but no higher than 82% on traces from applications that were not included in the training set.

Sanz et al. [15] used a simple feature set: the permissions and features of the device that the application requests upon installation. They are listed in the downloaded application's manifest, so these features are extracted with static analysis. Their data set consisted of 357 benign and 249 malicious applications. They tried several classifiers: logistic regression, naïve Bayes, Bayes nets, support vector machines with polynomial kernel, k-nearest neighbors, decision trees, random trees, and random forests. As with other work, the false positive rate remains stubbornly high: their false positive rate is never below 11%, and even that classifier only detects 45% of the malware. The best overall accuracy was 86%, using random forests.

Sahs and Khan [2] tried a substantially different approach, training a 1-class support vector machine on benign applications in order to detect malware as anomalies. They used a custom kernel that combines permissions information with control flow graph information, both of which come from static analysis. However, their false positive rate is nearly 50%, making their method untenable.

Wu et al. [16] report much better results—false positive rate below 1% and accuracy of 98%—but they only report on the training set error. Without evaluating on a testing set or using cross-validation, the good results are likely due to overfitting, instead of a model that generalizes well to unseen malware.

Peng et al. [17] explored the use of different probabilistic generative models for scoring the risk of different Android applications. They used the permissions requested by the application as the binary features (i.e., static analysis). Each model estimates the probability that an application would request those permissions. Each model is trained on several thousand applications from the marketplace, which the authors assume to all be benign. When a new application requests permissions that have a low probability according to the model, it is flagged as unusual or high risk. The probabilistic models range in complexity from simple naïve Bayes through a hierarchical mixture of naïve Bayes models. They used 378 malware applications mixed with different subsets of the benign set

to calculate cross-validation error. The hierarchical mixture of naïve Bayes models performs the best, detecting 78% of malware with a false positive rate of 4%. The simpler models also do well, achieving close to the same results.

D. Receiver Operating Characteristics (ROC) Curve

For classifiers that produce probability estimates—e.g., "There is a 72% chance this application is malware"—instead of just a yes/no decision, the aggressiveness of the overall system can be adjusted without modifying the classifier itself. To do this, one simply adjusts the probability threshold at which an application is declared malware. When the threshold is 0.0, everything is declared malware (i.e., the most aggressive classifier). On the other extreme, when the threshold is 1.0, nothing is declared malware. The default threshold is 0.5, picking the most likely category according to the classifier. This ability is important for malware classification because in different situations, different levels of aggressiveness would be appropriate. If one wants very high security, one might pick an aggressive classifier that can detect all of the malware, but also mistakenly flags several benign applications as malware (i.e., high false positives). On the other hand, if the classifier is used as part of a larger security suite, a less aggressive classifier would be preferred, producing fewer false positives.

While there are several ways to measure the quality of a classifier—accuracy, false positive rate, precision, etc.—the receiver operating characteristic curve (ROC curve) illustrates the trade off between false positives and detection rate as one moves from a conservative classifier (i.e., nothing is malware) to an aggressive classifier (i.e., everything is malware). (See Figure 3 for examples of ROC curves.) One can examine the curves in several different ways. The most concise is to calculate the area under the curve (AUC), which summarizes the quality of the classifier at all different levels of aggressiveness. An AUC of 1.0 is optimal, representing a perfect classifier.

One can also examine specific points on the ROC curve to find what fraction of malware can be detected—the true positive rate or TPR—when limiting the false positive rate (FPR) below some threshold. For example, one might want no more than 2% FPR in a particular system, so looking at the TPR value on the ROC curve when FPR=0.02 will estimate the detection rate of such a system.

E. Summary of Related Work

Table I summarizes the results from previous work. The table lists the TPR for different values of the FPR, along with the AUC. When the ROC is not reported in the given work, the closest FPR column is filled in. The best classifiers from each publication that meet the FPR limit are reported, and the best AUC is reported. Thus, the different columns may represent different classifiers. Publications where all of the FPR values were above 0.15 are omitted. As noted in Section I-C, there are many factors that influence the results, such as the makeup of the data set and whether or not applications as a whole are classified (static analysis) or execution traces from applications are classified (dynamic analysis). Thus, this table alone is an oversimplification of the results, but it highlights the difficulty in achieving decent detection rates at FPR of 0.02 or less. Our methods, in contrast, can detect 91% of malware at FPR=0.02 (Section III-D).

II. Methods

The most common features used in previous static analysis work are the permissions that the Android application requests upon installation. Our work explores a very different type of feature: n-gram frequencies. We converted Android application code (.apk) from the dex format to a .jar file using the Dare tool[1] [18]. The n-gram frequencies come from the resulting Java byte code (i.e., .class files) of the applications, excluding native libraries. The frequency of a particular n-gram in an application is the sum of the frequencies for all the .class files in the application.

Because there are too many n-grams to pass them directly as features to a classifier (see Table II), we apply principal components analysis (PCA) to obtain a low-dimensional representation of each application [19]. Each of the d values in that representation is a linear function of the original feature vector for that application.

The n-gram frequency matrix has one row for each training application and one column for each possible n-gram. Even after removing n-grams that do not occur in the training data, the matrix still has millions of columns. Thus, a basic application of PCA to the n-gram frequency matrix would be too inefficient. We employ two techniques to make the PCA problem tractable.

First, we exploit the sparsity of the matrix in order to store it in memory. While the original matrix is sparse (Section III-A), it needs to be centered before applying PCA. That centered matrix is not sparse, so our program dynamically creates a matrix multiplication function that is tailored to the frequency matrix (i.e., a function object). That function computes the product of the centered matrix and a vector argument without needing to expand the uncentered, sparse matrix. Specifically, let X be the (uncentered) frequency matrix. Then the centered matrix Z is $X - \mathbf{1}\mu^\top$, where μ is a column vector of the column means of X and $\mathbf{1}$ is a column vector of ones. Thus, the function object computes Zv for a vector v as $Xv - (\mu^\top v)\mathbf{1}$, where $\mu^\top v$ is a scalar and X is sparse.

Second, we use the implicitly restarted Lanczos bidiagonalization algorithm (IRLBA) [20]. It provides a way to efficiently compute an approximate, partial singular value decomposition of the large n-gram frequency matrix. We only need a partial singular value decomposition because only the d most significant right singular vectors are needed to project the data into d dimensions.

After obtaining the d-dimensional representation of each application, we explored several classifiers to learn the difference between malware and clean files. Sections III-C and III-D detail the results from those classifiers.

III. Experiments

A. Data Set

We used a collection of 3869 Android applications, which consists of 1433 malicious applications and 2436 benign applications. The malicious Android applications were collected from the VirusShare repository[2] and the Android Malware

[1] http://siis.cse.psu.edu/dare/

[2] http://virusshare.com/

TABLE I. SUMMARY OF RELATED WORK REPORTING MODERATE OR LOW FALSE POSITIVE RATES. TPR NUMBERS READ FROM A PLOT ARE APPROXIMATE, INDICATED BY $\approx$.

Citation	AUC	TPR values for FPR $\leq x$					Limitations
		$x = 0.01$	$x = 0.02$	$x = 0.05$	$x = 0.10$	$x = 0.15$	
Schmidt et al. [12]	-	0.77	-	-	0.99	1.00	ELF files only (system utilities, not APK applications)
Shabtai et al. [14]	0.913	-	-	≈ 0.967	-	0.847	author-created malware
Sanz et al. [15]	0.920	-	-	-	-	0.50	
Peng et al. [17]	0.954	< 0.5	≈ 0.59	≈ 0.79	≈ 0.87	≈ 0.90	

Genome Project[3] [21]. The benign Android applications are free, real-world applications collected from the Google Play market, covering various application categories. These free applications include different levels of popularity, as determined by the user rating scale. We used two existing malware detection tools [22, 23] to scan the collected free applications. Applications that did not trigger any alerts in those tools are kept in the benign set.

The applications were partitioned into training and test sets as follows. For the clean applications, a random 20% were selected for the test set, with the remainder going into the training set. The malicious applications were split based on the malware family. For each family with just one application, that application was randomly assigned to training or testing. For all the other families, at least one application was assigned to the test set. A few families of varying sizes were completely held out of the training set, so we could evaluate the algorithm's accuracy on completely unseen malware families. For each of the other malware families, 20% of the applications were selected for the test set, with the remainder going into the training set. There were 1948 benign and 1066 malicious applications in the training set, and there were 488 benign and 367 malicious applications in the test set. Only the training data was used for model selection, reserving the test data until the final evaluation (Section III-D).

Exploiting the sparsity of the n-gram frequency matrix was essential for efficient application of PCA. There are two kinds of sparsity we exploited. First, some n-grams never occur in the training data. Those columns are filled with zeroes, so they are removed from the frequency matrix. Table II shows that the fraction of columns that have any non-zero element decreases drastically as n increases. For example, with $n = 4$ less than half of one percent of the columns are used at all. Even after removing the all-zero columns, the matrix is still sparse. We found that the fraction of non-zero elements decreases slightly as n increases, but remains close to 8% for each n (Table II). This low density of non-zero elements is important for fitting the matrix into memory.

B. IRLBA Results

Table III shows the running time of IRLBA for each combination of n and d that we explored. The results are from a system with a quad-core 2.53 GHz Intel Xeon CPU (model E5630) and 48GB of RAM. Only the larger values of n and d take more than an hour to run. Overall, $d = 128$ and $n = 3$ leads to the highest accuracy classifiers (Section III-C). Thus, a little over two hours is sufficient to run IRLBA and generate informative features from the n-gram frequency matrix. This only needs to be done once during training, then used to classify as many applications as needed.

TABLE II. SPARSITY OF THE n-GRAM FREQUENCY MATRIX. NUM. n-GRAMS IS THE NUMBER OF DISTINCT n-GRAMS PRESENT IN THE TRAINING DATA. NUM. NON-ZERO AND FRAC. NON-ZERO ARE THE NUMBER OF NON-ZERO ELEMENTS IN THE FREQUENCY MATRIX AND THE FRACTION OF ELEMENTS THAT ARE NON-ZERO, RESPECTIVELY.

n	Num. n-grams	Num. Non-Zero	Frac. Non-Zero
2	64,486	27,702,199	0.0832
3	2,352,501	129,153,554	0.0828
4	14,324,759	257,692,827	0.0818
5	42,807,668	374,699,304	0.0803
6	86,571,071	450,483,143	0.0783

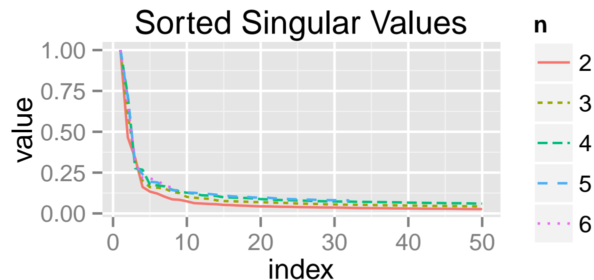

Fig. 1. Scaled singular values of the n-gram frequency matrices.

Even after exploiting the sparsity of the frequency matrix, the bottleneck of our algorithm is the RAM needed to perform PCA using IRLBA. We ran the algorithm with a cap of 48GB of RAM, and some of the values of n and d were halted after exceeding that limit (Table III). There is likely room to improve the memory usage, since we did not optimize the software as part of this work. Instead, we evaluated the IRLBA package for R as it comes off the shelf[4]. While the memory requirement is a limitation of the algorithm, the PCA step does not need to be performed often, so a cloud computing lease would be a good option for performing this computation. Furthermore, new training data can be incorporated without redoing PCA: simply use the existing singular vectors to compute the d-dimensional representation of the new training data points, feeding the updated data set into the classifier for retraining.

The sum of the top d singular values returned from IRLBA tells the amount of variance in the original data that is captured when projecting to the d-dimensional representation of the data. Figure 1 shows the singular values for the different values

[3] http://www.malgenomeproject.org/

[4] http://cran.r-project.org/web/packages/irlba/

TABLE III. Running time (wall clock time) for IRLBA (hh:mm:ss). The column indicates d and the row indicates n. Blank cells indicate combinations of n and d that exceeded the memory limit.

n ↓ d →	2	4	8	16	32	64	128	256	512	1024
2	0:00:43	0:00:45	0:00:45	0:00:51	0:01:08	0:01:54	0:04:29	0:15:21	1:20:21	6:24:32
3	0:03:52	0:03:56	0:04:27	0:07:29	0:15:57	0:36:43	2:10:48	12:25:00	64:34:42	
4	0:10:33	0:10:37	0:13:36	0:30:58	1:09:26	3:20:42				
5	0:21:17	0:22:18	0:31:55	1:14:02	3:00:53					
6	0:34:11	0:36:12	0:51:47							

of n, normalized so that the highest value for each n is 1.0. For all values of n, the first few singular values are much larger than the others. For example, the fifth highest singular value is less than 25% of the largest value, for each n. While a low value of d will capture the majority of the variation in the data set, that does not necessarily mean that low d will lead to good classification. The next section includes an investigation of the relationship between d and classifier accuracy.

C. Classification Results: Cross-Validation Measures

Running IRLBA produces one low-dimensional data matrix for each combination of d (i.e., the number of principal components used) and n. For each of those matrices, we ran several classification algorithms, evaluating their 10-fold cross-validation accuracy to pick the best values of n and d, along with any other parameters specific to the classification algorithm. This section examines that step, while Section III-D evaluates the best classifier of each type on separate test data.

We use the receiver operating characteristic curve (ROC curve) to evaluate the classifiers, comparing them based on their area under the curve (AUC). We compared five classifiers: support vector machines, random forests, naïve Bayes, k-nearest neighbors, and boosted decision trees (J48 with Adaboost). We used the Weka implementation of each classifier [24]; the specific respective classes are weka.classifiers.{functions.SMO, trees.RandomForest, bayes.NaiveBayes, lazy.IBk, and meta.AdaBoostM1 with weka.classifiers.functions.supportVector.RBFKernel as the kernel}. For support vector machines, we explored values of $C \in \{10^{-3}, 10^{-2}, 10^{-1}, ..., 10^{3}, 10^{4}\}$ and $\gamma \in \{2^{-1}, 2^{-2}, 2^{-3}, ..., 2^{-10}\}$, picking the best values for each n and d based on the highest cross-validation AUC. Similarly, for the random forests, we set the number of trees to 16, 32, 64, 128, and 256, picking the best value for each n and d. For k-nearest neighbors, we picked the best value of k from 1 through 7.

Figure 2 illustrates the cross-validation AUC values for each classifier and for each combination of n and d. All of the algorithms except naïve Bayes reach above 0.97. The best overall AUC was 0.9885, attained by random forests with 256 trees, $n = 3$, and $d = 128$. This is higher than achieved in previous work (Section I-C, Table I).

Of the five classifiers, there is a clear distinction between naïve Bayes and the other four classifiers. It is not surprising that naïve Bayes performed poorly, since it assumes independence of the feature values given the classification. This contrasts with the PCA transformation, which entangles many of the n-gram counts by generating linear combinations of those n-gram frequencies.

TABLE IV. Classifiers of each type with highest cross-validation AUC.

Algorithm	n	d	Other	AUC
SVMs	2	512	$C = 10, \gamma = 0.5$	0.9820
Random Forests	3	128	trees=256	0.9885
Naïve Bayes	2	16		0.8649
KNN	3	512	k=1	0.9795
Boosted Dec. Trees	3	128		0.9853

The high AUC for the other classifiers indicates that *using PCA on the n-gram frequencies produces informative features for Android malware classification*. Each of those classifiers performs the worst with low d, generally increasing the AUC with higher d until a point of diminishing returns around $d = 128$. Table IV lists the AUC and the parameters for the best classifier of each type. These were the models used for test set evaluation, except for k-nearest neighbors. For KNN, we evaluate the $n = 3, d = 128$ model on the test set because it is simpler than the best model ($n = 3, d = 512$) but achieves an AUC only 0.0007 less.

D. Classification Results: Separate Test Set

After picking the best classifier of each type (Section III-C), we evaluated their accuracies on our test set. We ran three evaluations of the classifiers, using three different subsets of the test data. They each use all the benign applications from the test set, but they differ in which malware from the test set are used. The "unfamiliar" evaluation only uses the malware from families that were *not* represented in the training data. The "familiar" evaluation uses the other malware (i.e., their families were represented in the training data), and the "all" evaluation uses all of the malware in the test set.

Our "all" data set is like the evaluations performed in previous work, where the authors do not consider the families of the malware. The results are presented at the top of Table V, along with the best results from previous work (copied from Table I). The *random forest classifier performs the best*, both in terms of AUC and in the TPR at the different FPR levels. Furthermore, the random forest *improves upon previous work by a substantial amount, particularly for FPR=0.01 and FPR=0.02, where we can detect an additional 30% of the malware* when compared with previous work. For example, at FPR=0.02, we can detect over 91% of the malware in our test data set, whereas the previous best rate was 59% [17]. Evaluating the classifiers at low FPR levels is quite important for this application: with thousands of applications in the Android marketplace, a screening tool with a false positive rate over 2% is still too high for practical use.

Not only does the random forest perform well, but KNN, SVMs, and boosted decision trees all perform better than

TABLE V. EVALUATION OF CLASSIFIERS ON TEST DATA. RESULTS WITHIN EACH DATA SET ARE SORTED BY THE TPR AT FPR=0.01. THE BEST VALUE IN EACH COLUMN FOR EACH DATA SUBSET IS SHOWN IN BOLD.

Data Set	Algorithm	AUC	TPR values for FPR= x				
			$x = 0.01$	$x = 0.02$	$x = 0.05$	$x = 0.10$	$x = 0.15$
Peng et al. [17]		0.954	< 0.5	≈ 0.59	≈ 0.79	≈ 0.87	≈ 0.90
All	Naïve Bayes	0.8100	0.3733	0.4005	0.5068	0.5804	0.6213
	KNN	0.9730	0.7657	0.8093	0.9019	0.9373	0.9510
	SVMs	0.9670	0.7793	0.8093	0.8638	0.9183	0.9482
	Boosted Dec. Trees	0.9700	0.8229	0.8338	0.8856	0.9264	0.9510
	Random Forests	**0.9850**	**0.8638**	**0.9101**	**0.9428**	**0.9564**	**0.9619**
Familiar	Naïve Bayes	0.8580	0.4599	0.4878	0.6237	0.6969	0.7247
	SVMs	0.9840	0.8711	0.8920	0.9199	0.9721	0.9791
	KNN	0.9920	0.8955	0.9094	0.9686	0.9756	**0.9861**
	Boosted Dec. Trees	0.9890	0.9477	0.9477	0.9547	0.9686	0.9756
	Random Forests	**0.9930**	**0.9582**	**0.9686**	**0.9756**	**0.9791**	0.9826
Unfamiliar	Naïve Bayes	0.6370	0.0625	0.0875	0.0875	0.1625	0.2500
	KNN	0.9040	0.3000	0.4500	0.6750	0.8000	0.8250
	Boosted Dec. Trees	0.9030	0.3750	0.4250	0.6375	0.7750	0.8625
	SVMs	0.9080	0.4500	0.5125	0.6625	0.7250	0.8375
	Random Forests	**0.9540**	**0.5250**	**0.7000**	**0.8250**	**0.8750**	**0.8875**

the best previous work. The fact that all of these classifiers perform well indicates that the features upon which they are trained—n-grams compressed through PCA—contain useful information for classifying Android malware. Thus, our work demonstrates that Android malware exhibit some low-level similarities that can be exploited to recognize other malware.

When we break up the malware into familiar and unfamiliar sets, the classifiers perform better on the familiar subset (Table V). While this is not surprising, the results highlight the importance of evaluating malware classifiers on unfamiliar malware families, an evaluation that has not been done in previous work. This is important because the unfamiliar subset best simulates the situation of real interest: the classifier needs to detect malware that have not yet been created, including those from families that have not yet been created.

The random forest's performance on unfamiliar malware is better than the previous best performance on familiar/unfamiliar combined (like our "all" configuration). At the the FPR=0.02 level, we can detect 10% additional malware, despite the more challenging test set. The detection rate of 70%, although better than previous work, leaves room for improvement. Future work will include augmenting the n-grams with other features such as permissions bits, providing the classifier with complementary information about the applications. Our preliminary results in this area indicate that such combinations can significantly increase the malware detection accuracy even beyond our contributions in this work.

IV. SUMMARY

We presented a new method for classifying Android applications as malicious or benign that is more accurate than previous work. The method extracts Java byte code from the Android application, then computes the n-gram frequencies of that byte code. The frequency matrix is passed through PCA, which can be efficiently performed using a combination of IRLBA and sparse matrix representations. The output of PCA is then fed to a classifier, which learns to distinguish between malicious and benign applications.

We found that our new feature set—features produced by using PCA on the n-gram frequency matrix—is very informative. In particular, several classification algorithms used those features to achieve AUC values of 0.98 or higher on cross-validation data; all of those scores are higher than found in previous work. We also computed the AUC on testing data where the malware is from families that were not used in training. The the 0.954 AUC that the random forest earns on that data provides evidence that it will perform well on new malware families.

REFERENCES

[1] L. Whitney, "iPhone market share shrinks as Android, Windows Phone grow," http://news.cnet.com/8301-13579_3-57616679-37/iphone-market-share-shrinks-as-android-windows-phone-grow/, Jan. 2014.

[2] J. Sahs and L. Khan, "A machine learning approach to Android malware detection," in *2012 Euro. Intelligence and Sec. Informatics Conf. (EISIC)*, 2012, pp. 141–147.

[3] B. Wolfe, K. O. Elish, and D. Yao, "Comprehensive behavior profiling for proactive android malware detection," in *Proc. of Information Sec. - 17th Int. Conf. (ISC 2014)*, ser. Lecture Notes in Comp. Sci., 2014, p. To appear.

[4] B. Amos, H. Turner, and J. White, "Applying machine learning classifiers to dynamic android malware detection at scale," in *2013 9th Int. Wireless Commun. and Mobile Computing Conf. (IWCMC)*, 2013, pp. 1666–1671.

[5] W. Lee, S. J. Stolfo, and K. W. Mok, "A data mining framework for building intrusion detection models," in *Proc. of the 1999 IEEE Symposium on Sec. and Privacy*. IEEE, 1999, pp. 120–132.

[6] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious JavaScript code," in *Proc. of 19th Int. World Wide Web Conf.*, 2010.

[7] P. Xie, J. H. Li, X. Ou, P. Liu, and R. Levy, "Using Bayesian networks for cyber security analysis," in *2010 IEEE/IFIP Int. Conf. on Dependable Syst. and Networks (DSN)*. IEEE, 2010, pp. 211–220.

[8] A. Srivastava, S. Sural, and A. K. Majumdar, "Database intrusion detection using weighted sequence mining," *Journal of Computers*, vol. 1, no. 4, pp. 8–17, 2006.

[9] H. Tan, N. Goharian, and M. Sherr, "$100,000 prize

jackpot. Call now!: Identifying the pertinent features of SMS spam," in *Proc. of the 35th Int. ACM SIGIR Conf. on Research and development in information retrieval*. ACM, 2012, pp. 1175–1176.

[10] J. Z. Kolter and M. A. Maloof, "Learning to detect and classify malicious executables in the wild," *J. of Machine Learning Research*, vol. 7, pp. 2721–2744, 2006.

[11] A. Shabtai, R. Moskovitch, Y. Elovici, and C. Glezer, "Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey," *Inform. Sec. Technical Report*, vol. 14, no. 1, pp. 16 – 29, 2009.

[12] A.-D. Schmidt, R. Bye, H.-G. Schmidt, J. Clausen, O. Kiraz, K. Yuksel, S. Camtepe, and S. Albayrak, "Static analysis of executables for collaborative malware detection on android," in *IEEE Int. Conf. on Commun., 2009*, 2009, pp. 1–5.

[13] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-based malware detection system for Android," in *Proc. of the 1st ACM Workshop on Sec. and Privacy in Smartphones and Mobile Devices*, ser. SPSM '11, 2011, pp. 15–26.

[14] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, "Andromaly: a behavioral malware detection framework for Android devices," *Journal of Intelligent Inform. Syst.*, vol. 38, no. 1, pp. 161–190, 02 2012.

[15] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P. Bringas, and G. Alvarez, "PUMA: Permission usage to detect malware in Android," in *Int. Joint Conf. CISIS'12-ICEUTE'12-SOCO'12*, 2013, vol. 189, pp. 289–298.

[16] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "DroidMat: Android malware detection through manifest and API calls tracing," in *7th Asia Joint Conf. on Inform. Sec. (Asia JCIS)*, 2012, pp. 62–69.

[17] 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 *Proc. of the 2012 ACM Conf. on Computer and Commun. Sec.*, ser. CCS '12, 2012, pp. 241–252.

[18] D. Octeau, S. Jha, and P. McDaniel, "Retargeting Android applications to Java bytecode," in *Proc. of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2012.

[19] H. Hotelling, "Analysis of a complex of statistical variables into principal components," *Journal of Educational Psychology*, vol. 24, pp. 417–441, 1933.

[20] J. Baglama and L. Reichel, "Restarted block Lanczos bidiagonalization methods," *Numerical Algorithms*, vol. 43, no. 3, pp. 251–272, 2006.

[21] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. of the IEEE Symposium on Sec. and Privacy*, 2012, pp. 95–109.

[22] "Virus Total," https://www.virustotal.com/.

[23] K. O. Elish, D. Yao, and B. G. Ryder, "User-centric dependence analysis for identifying malicious mobile apps," in *Proc. of the Workshop on Mobile Sec. Technologies (MoST), in conjunction with the IEEE Symposium on Sec. and Privacy*, 2012.

[24] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: An update," in *SIGKDD Explorations*, vol. 11, no. 1, 2009.

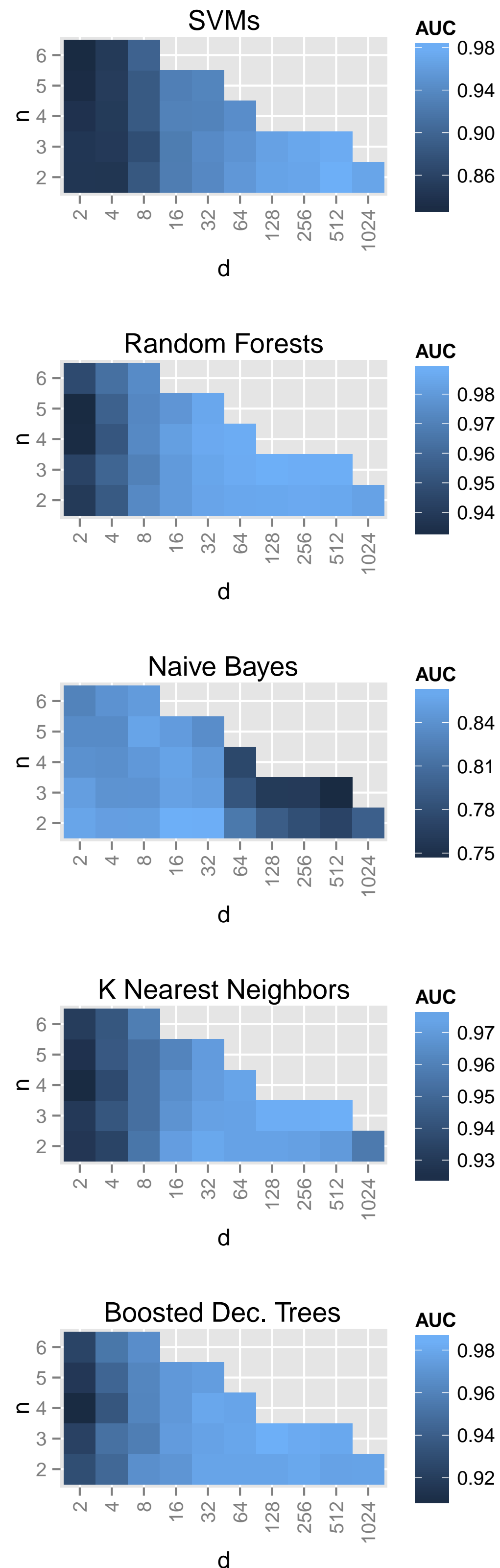

Fig. 2. The cross-validation AUC for each type of classifier.

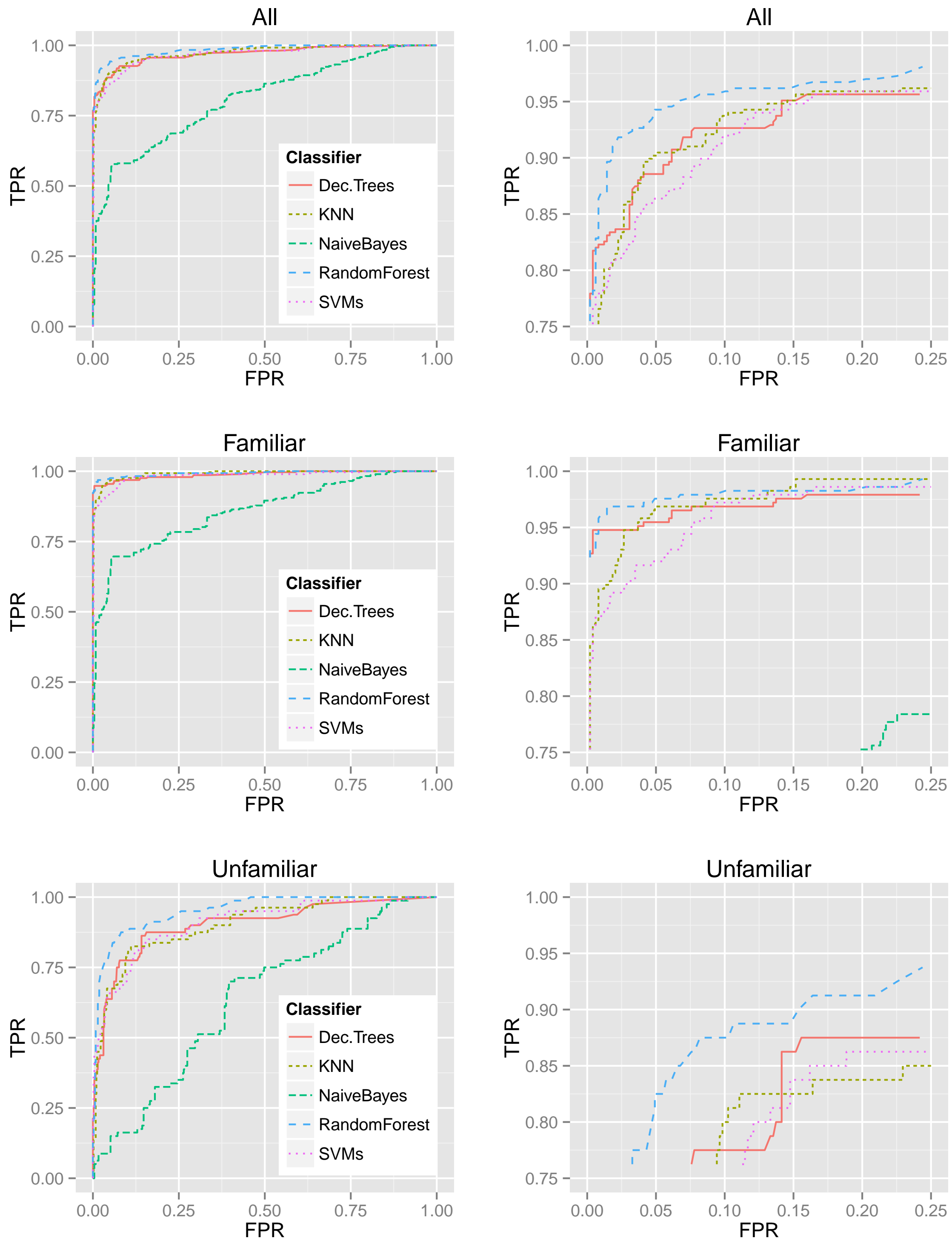

Fig. 3. The ROC curve for each test data subset: true positive rate (TPR) versus false positive rate (FPR). The left column shows the full curves, while the right column magnifies the top left corner of the curves.