An Android Malware Detection Approach Using Bayesian Inference

Che-Hsun Liu, Zhi-Jie Zhang and Sheng-De Wang

Graduate Institute of Electrical Engineering, National Taiwan University, Taipei 10617, Taiwan
{r01921044, r01921025, sdwang}@ntu.edu.tw

Abstract: Android malware detection has been a popular research topic due to non-negligible amount of malware targeting the Android operating system. In particular, the naive Bayes generative classifier is a common technique widely adopted in many papers. However, we found that the naive Bayes classifier performs badly in Contagio Malware Dump dataset, which could result from the assumption that no feature dependency exists.

In this paper, we propose a lightweight method for Android malware detection, which improves the performance of Bayesian classification on the Contagio Malware Dump dataset. It performs static analysis to gather malicious features from an application, and applies principal component analysis to reduce the dependencies among them. With the hidden naive Bayes model, we can infer the identity of the application. In an evaluation with 15,573 normal applications and 3,150 malicious samples, our work detects 94.5% of the malware with a false positive rate of 1.0%. The experiment also shows that our approach is feasible on smartphones.

I. Introduction

Android has been the most popular operating system in mobile devices. There are two well-known mechanisms provided by Google to protect the Android system: the permission review on the application installation and the Google Bouncer check on application publishing. The permission review is a simple mechanism that tells device owners before installation what kinds of resources the application will access. Users must explicitly approve the agreement to continue the installation. However, most users tend to approve without meticulous examination of the permissions. In addition, some normal applications request permissions that are not required to execute [1] the applications. Google Bouncer is a security service announced at February 2012 [2], which scans its online store automatically for potential malware by a dynamic analysis. However, it is possible to circumvent Google Bouncer as part of the analytic process has been revealed [3]. It is important to study Android malware detection to make up for the deficiency.

Due to the inefficiency of detecting new Android malware, anomaly detection studies are more prevalent in recent years than conventional signature-based approaches. Generally, the anomaly detection techniques are coarsely categorized into two groups: dynamic analysis and static analysis. Dynamic analysis approaches like DroidScope [4], Andromaly [5] and Crowdroid [6] track runtime behaviors of applications. Some execution paths may be omitted though the runtime behaviors are reflected. Static analysis approaches like DREBIN [7] and Peng et al. [8] conquer the weakness by inspecting file contents of applications, which are considered more robust.

Bayesian classification model, especially naive Bayes, is the most popular machine learning generative model among the static analysis methods including Yerima et al. [9], Peng et al. [8] and MAMA [10]. However, we found that applying the naive Bayes model in Contagios Malware dataset [11] has extremely poor performance. It is caused possibly by the nature of naive Bayes lacking in considering complicated feature dependencies and by the great diversity of the dataset. Hence, in this paper, we propose using principal component analysis and the hidden naive Bayes model to mitigate the influence of feature dependency. We conduct experiments on a dataset comprising 15,573 normal applications collected from Google Play and 3,150 malware samples from Contagio Malware Dump. The proposed method outperforms related works [8]–[10] with 98.2% accuracy, 94.5% true-positive rate and 1.0% false-positive rate.

In summary, the contributions of this paper to Android malware detection are listed as follows:

- The experiment shows that our approach mitigates the influence of complex feature dependencies on Android malware detection, which results in a performance improvement.
- By selecting those features emphasizing malware by mutual information on malicious API calls and permissions, we can avoid benign attacks, i.e. malware utilizing large number of features that emphasize normal applications.
- With reasonable computational complexity, our approach is feasible to perform analysis on smartphones.

II. Related Work

The studies in the field of Android security are diverse in recent years. In response to the remarkable growth of Android malware, several approaches and concepts from the conventional computer security research have been applied. Enck et al. [12] seeks all the possible vulnerabilities in Android application that can be exploited by malware. Felt et al. [13] studies the incentives of malware on different mobile

978-1-5090-4314-9/16 $25.00 © 2016 IEEE
DOI 10.1109/CIT.2016.76

systems and provides practical advice for defense. Zhou & Jiang [14] systematically characterize the behavior of existing Android malware families and provide the overview in a clear manner.

A. Static Analysis Approaches

As implied by the name, this kind of approaches uses static program analysis without actually running programs. With mature DEX decompilers like Baksmali [15] and Androguard [16], the original Java source code of Android applications can be almost fully recovered.

One kind of studies focuses on the Android permissions. To detect overpriviledged applications, i.e. applications with unnecessary permissions, Stowaway [17] constructs a map that identifies the permission required for each method in the Android API by semi-automatic testing. More generally, PScout [18] can perform version-independent static analysis that analyzes the Android source code to build the permission map. Wei et al. [19] present a longterm study concerning the evolution of permission model in the Android ecosystem.

Kirin [20] is a malware detection service providing a lightweight certification on Android devices. It checks the specific rules on permissions to mark malware. Instead of signature-based approaches, Peng et al. [8] use probabilistic generative models on permissions to compute the risk scores of applications. MAMA [10] fully capitalizes on the information within the AndroidManifest.xml file and uses random forests to detect malware. Yerima et al. [9] selects features from those defined by the domain knowledge, and then applies the Bayesian classification. DroidAPIMiner [21] selects features from API calls by a frequency analysis. It achieves an excellent performance via the kNN algorithm. DREBIN [7] utilizes linear support vector machine (SVM) on the comprehensive feature sets and provides explainable results.

B. Dynamic Analysis Approaches

In contrast with static analysis approaches, dynamic analysis approaches monitor the runtime behavior of applications. DroidScope [4] provides a virtualization-based malware analysis platform where the behavior of malware can be understood in different level. TaintDroid [22] is another similar work which is able to tracking multiple sources of sensitive data. Crowdroid [6] monitors Linux Kernel system calls in the local devices. The information gathered is then sent back to a remote server and clustered to distinguish malware from the benign applications. Andromaly [5] monitors a few system information like memory, CPU, and battery consumption and applies different machine learning algorithms.

III. CLASSIFICATION MODELS

Before introducing our methodology, we first model the problem formally and define the mathematical symbols and machine learning approaches used in the following context.

Definition 1. The problem of Android malware detection is defined as:

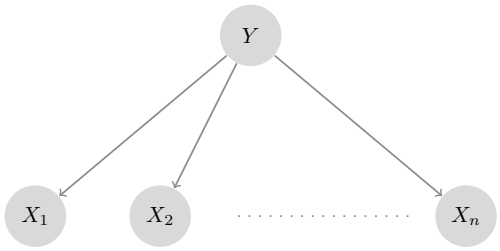

Fig. 1: The structure of naive Bayes models

Input: a set of m applications $(\mathbf{x}^j, y^j)$, $j = 1, 2, \ldots, m$ (the training set), sampled from some distribution D, where $\mathbf{x}^j \in \mathbb{R}^n$ and $y^j \in \{0, 1\}$. The i-th component of $\mathbf{x}^j$, x_i^j, is termed i-th feature, which can be either the presence or absence of a permission or anything else. If $y^j = 1$ the application will be referred to as malicious, and if $y^j = 0$ it will be referred to as benign.

Output: a function $f : \mathbb{R}^n \to \{0, 1\}$ which classifies an additional application $\{\mathbf{x}^k\}$ sampled from the same distribution D.

To solve the supervised learning problem above in which we wish to estimate the unknown target function $f : X \to Y$, or equivalently $P(Y|X)$, we propose using generative classifiers. That is, in contrast with discriminative classifiers which model the posterior $P(Y|X)$ directly, we model the joint probability $P(X, Y) = P(X|Y)P(Y)$, and make predictions by applying Bayes' theorem [23].

A. Naive Bayes Models

We first introduce the general case of naive Bayes models, Bayesian networks. A Bayesian network is a probabilistic graphical model that uses a directed acyclic graph (DAG) to represent a set of random variables and their conditional dependencies. Figure 1 shows an example of the graph in which a node represents a feature and an arc represents a feature dependency.

It is common to apply Bayesian networks to solve classification problems. It needs a training set with class labels to construct its structure. After the construction, consider an application from the testing set given by $\mathbf{x}^k = \langle x_1^k, \ldots, x_n^k \rangle$. By applying Bayes' theorem, the posterior for each class $y^l \in Y$ can be represented as

$$P(y^l|\mathbf{x}^k) = \frac{P(\mathbf{x}^k|y^l)P(y^l)}{\sum_{y \in Y} P(\mathbf{x}^k|y)P(y)}. \tag{1}$$

The Bayesian network classifier compares all the posteriors and classifies the sample into the class $\hat{y}$ which posterior is the highest. Therefore, the classifier represented by a general Bayesian network is defined as:

$$\hat{y} = \underset{y \in Y}{\operatorname{argmax}} \left\{ P(\mathbf{x}^k|y)P(y) \right\}. \tag{2}$$

Naive Bayes model is the simplest form of Bayesian network. It assumes that, given the class Y, each feature X_i is

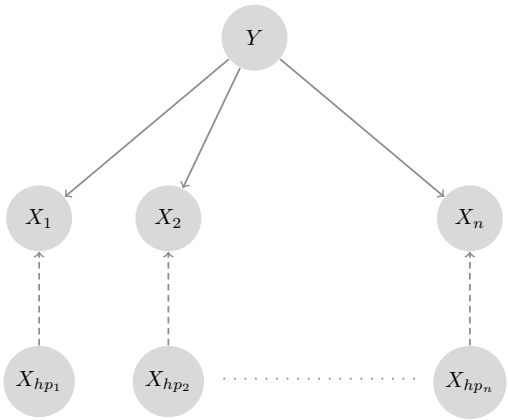

Fig. 2: The structure of hidden naive Bayes models

conditionally independent of every other feature X_j for $i \neq j$. With these assumptions and Equation 2, we have

$$\hat{y} = \underset{y \in Y}{\operatorname{argmax}} \left\{ \prod_{i=1}^{n} P(x_i^k|y)P(y) \right\}. \quad (3)$$

B. Hidden Naive Bayes Models

Apparently, the conditional independence assumption of naive Bayes models is too strong. For instance, an Android application creating a `FileOutputStream` instance to save a file into the external storage must request the `WRITE_EXTERNA_STORAGE` permission in the manifest file. This example implies that if an application uses a method in the class `FileOutputStream`, it is likely that it requests the `WRITE_EXTERNAL_STORAGE` permission. A general Bayesian network classifier is another choice. However, learning the optimal structure of an Bayesian network is an NP-hard problem [24]. Thus, hidden naive Bayes (HNB) is a better choice [25], which not only considers dependencies but also avoids structure learning.

Figure 2 shows the structure of a HNB model. It inherits the structure of naive Bayes. Besides, each feature X_i has a hidden parent X_{hp_i} (each pair is connected by a dashed line). The HNB classification model is defined as follows:

$$\hat{y} = \underset{y \in Y}{\operatorname{argmax}} \left\{ \prod_{i=1}^{n} P(x_i^k|x_{hp_i}^k, y)P(y) \right\} \quad (4)$$

where

$$P(X_i|X_{hp_i}, Y) = \sum_{j=1, j\neq i}^{n} W_{ij} \times P(X_i|X_j, Y). \quad (5)$$

From Equation 5, the hidden parent X_{hp_i} for a feature X_i is substantially a weighted sum of all other features. The weights W_{ij} is defined in:

$$W_{ij} = \frac{I(X_i; X_j|Y)}{\sum_{l=1, l\neq i}^{n} I(X_i; X_l|Y)} \quad (6)$$

where $I(X_i; X_j|Y)$ is the conditional mutual information

$$I(X_i; X_j|Y) = \sum_{x_i, x_j, y} P(x_i, x_j, y) \log \frac{P(x_i, x_j|y)}{P(x_i|y)P(x_j|y)}. \quad (7)$$

C. Principal Component Analysis

Although the HNB classifier considers the dependencies among features, the hidden parents added to each feature are still an approximation to real dependencies. It cannot learn the actual structure completely. Therefore, to alleviate the influence of feature dependencies, principal component analysis (PCA) is applied.

PCA can be considered as a axes rotation that transforms the original coordinate system to a new set of variables such that the first few retain most of the variation in all of the original variables [26]. The new variables are called the principal components (PCs). The k-th derived PC is a linear combination of the original variables

$$\eth_k^T X = a_{k,1}X_1 + a_{k,2}X_2 + \ldots + a_{k,n}X_n \quad (8)$$

where $\eth_k$ is the eigenvector of the covariance matrix corresponding to the kth largest eigenvalue λ_k.

Although PCA of binary data has its critics, it is equivalent to principal coordinate analysis with a very plausible measure of similarity [27]. The basic objective of PCA—to summarize most of the variation that is present in the original set of variables using a smaller number of derived variables—can be achieved regardless of the nature of the original variables [26].

IV. Methodology

The flowchart of the proposed procedure is depicted in Figure 3. Taking an unknown Android application as an input, static analysis is performed to get all the features of it. Then the necessary features are selected, transformed with PCA, and discretized to generate a feature vector. Finally, the prediction is made by passing the feature vector to the classifier learned from the dataset.

A. Feature Extraction

Two kinds of features are needed in this paper: one is permission and the other is application program interface (API) call. Permissions can represent macro behavior of an application while API calls are actual implementation.

Every Android application is distributed and installed from an Android application package (APK) file, which is a variant of JAR file format. To get the necessary features, we need to parse `AndroidManifest.xml`, which contains permissions, and decompile `classes.dex`, which contains all the Dalvik bytecodes, i.e. the API calls. In this paper, we use Python API provided by Androguard [16] to perform all the static analysis.

1) Parsing Manifest: An Android manifest file provides the essential information about an application to the Android system. Our target is to get all the `<uses-permission />` elements in the manifest. These tags specify the permissions requested by an application to the system.

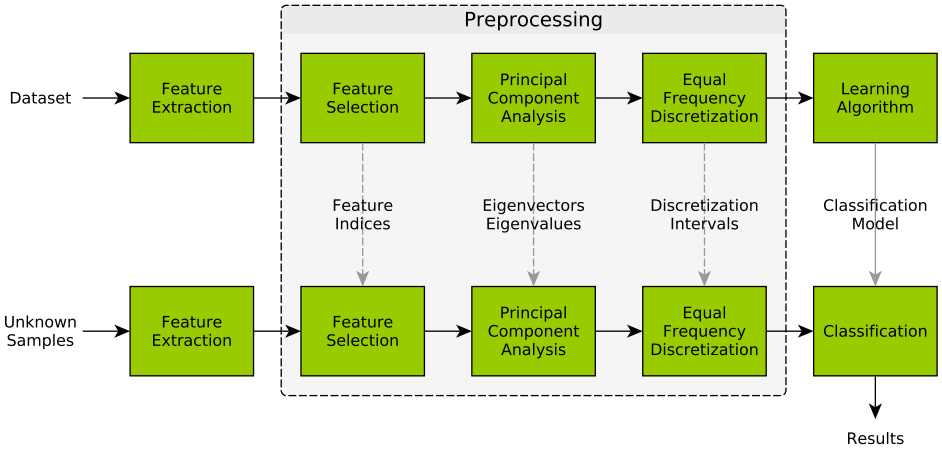

Fig. 3: Flowchart of the proposed approach.

The usual way to extract the permission tags is decompressing the APK file, converting the binary manifest file into readable XML format, and using XML parser to get them. We use Androguard to combine these three steps directly.

2) Decompiling Dalvik Executable: We use Androguard to accomplish the decompilation. A regular expression pattern is then applied to get all the methods from the DEX file. Every class and method defined by application inventors is excluded. Since Java provides method overloading, several features actually indicate the same behavior. We ignore the parameters to avoid the condition.

B. Feature Selection

The total number of permissions and API calls are huge. There are 145 permissions in Android, not to mention those defined by developers. Each application can also reference up to 60,000 methods. If all the features were used, the complexity of the classifier would be too large and hence impractical. Besides, some features reveal no information about the dataset or even confuse the classifier. Using all the features can result in performance degradation [28]. Hence, selecting the most relevant features and discarding the irrelevant ones are of crucial importance.

Some papers use only the 145 permissions [8] [10] or use the most frequently used ones [29], which have reasonable computational complexity and excellent performance. However, such classifiers can be easily broken by requesting much more permissions in the malicious application. Since Android applications tend to become overprivileged [19], it is difficult to discriminate the malicious ones from the normal ones. Figure 4 shows the number of permissions requested by the benign and malicious applications. Although the distribution of the benign applications is smoother, two distributions are almost identical. To prevent the classifier from being deceived by malware disguised as benign application, we perform frequency analysis and mutual information (MI) to select

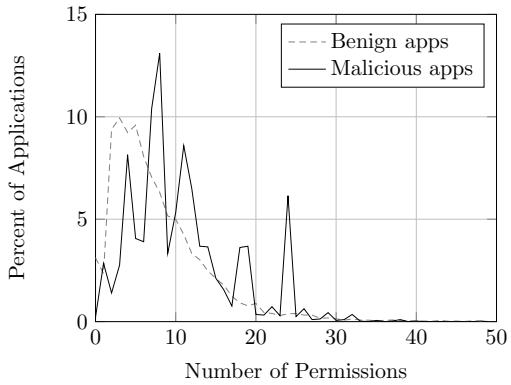

Fig. 4: Percentage of applications that request a specific number of permissions.

those features that emphasize malware only. We could utilize MI directly to select features like [9], but, without domain knowledge, it would be hard to sieve out the features that appear frequently in the benign applications.

In our classifier, we first mix all kinds of features, i.e., API calls and permissions. The frequency analysis is then performed to retrieve those used more often in malware than in normal applications and followed by mutual information to get the most relevant features.

1) Mutual Information: Mutual information (MI) is an approach that measures mutual dependence of a pair of random variables. It estimates the similarity between the joint distribution and the products of marginal distributions. In our classification problem, MI is given by

$$I(X_i;Y) = \sum_{y\in Y}\sum_{x\in X_i} P_{X_i,Y}(x,y)\log_2\frac{P_{X_i,Y}(x,y)}{P_{X_i}(x)P_Y(y)} \quad (9)$$

where X_i is the i-th feature and Y represents the label. The higher the score, the more relevant the feature.

2) Permissions: The permission `INTERNET` (95%) occupies the top position due to the popularization of Internet, but it provide exceedingly little information as both malicious and benign applications require it often (96% and 93%). The MI score of `INTERNET` is 0.0015, which ranks only 49^{th} in all the permissions. Nonetheless, the permissions such as `READ_SMS` and `SEND_SMS` have lower usage but are requested by malware much more frequently than by benign applications. The MI scores are 0.1370 and 0.1057 respectively, which provide more information and thus can properly enhance the classifier. Permissions that are seldom requested like `PROCESS_OUTGOING_CALLS`, which allows an application to see the number dialed, `INSTALL_PACKAGES`, which allows an application to install packages, and `WRITE_APN_SETTINGS`, which allows an application to change the APN settings, do frequently appear in malware.

3) API Calls: We analyze the use frequency of API calls with the highest MI scores. The methods `getSubscriberId` and `getLine1Number` from `TelephonyManager` require the `READ_PHONE_STATE` permission, and `sendTextMessage` requires the `SEND_SMS` permission, all of which have relatively high MI scores. Furthermore, some API calls that are common in malware like `exec` from `Runtime` do not require permissions. The API calls provide more specific descriptions to applications than permissions and hence reinforce the reliability of the classifier.

V. Evaluation

A. Dataset

In network security, Android devices is a relatively new field. There are only few datasets available and reliable. Our dataset comprises 3,150 malware from ContagioDump [11] and 15,573 benign applications collected from Google Play.

Contagio Malware Dump blog provides a collection of the latest malware samples, threats, observations and analyses. It is a place for security information exchange. The objective of the website is "take a sample, leave a sample." Some zero-day attacks were even chronicled on the blog quickly. The 3,150 malware were obtained from the website, and then we used a free online scanning service called *VirusTotal* to label them with the families which each of the malware belongs to. The top 20 malware families in the dataset are shown in Table I.

The benign set were built by collecting the applications listed in the top 600 free of each category (46 categories totally) in Google Play in May 2014. There were 20,850 applications initially. Each of them were uploaded to *VirusTotal* and scanned by all the antivirus. Any application that was considered as malicious by at least two of the scanners was discarded. The final benign set contains 15,573 applications.

TABLE I: Top 20 malware families in the dataset.

Id	Family	#	Id	Family	#
A	DroidKungFu	473	K	FakePlayer	74
B	DorDrae	420	L	Wroba	74
C	Meds	221	M	Plankton	63
D	Fakeguard	203	N	DroidDreamLight	52
E	Boxer	202	O	Cawitt	51
F	Kmin	183	P	Badao	46
G	Rooter	117	Q	Fake10086	46
H	Boqx	114	R	Cupi	39
I	DroidAp	106	S	Coogos	39
J	DroidKungFu3	93	T	DroidDream	39

B. Evaluation Metrics

Several metrics used for the performance evaluation are listed as follows:

- **Accuracy**, the number of benign and malicious applications that are correctly classified divided by the total number of applications in the dataset:
$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$
where true positives (TP) and false positives (FP) are the malicious instances that are correctly and incorrectly identified respectively while true negatives (TN) and false negatives (FN) are the benign instances that are correctly and incorrectly identified respectively.
- **True-positive rate**, the percentage of the malware that are correctly identified:
$$TPR = \frac{TP}{TP + FN}.$$
- **False-positive rate**, the fraction of the benign instances that are incorrectly identified:
$$FPR = \frac{FP}{TN + FP}.$$
- **Precision**, the fraction of the instances identified as malicious that are correctly identified:
$$Precision = \frac{TP}{TP + FP}.$$
- **Area under the curve**, the area under the receiver operating characteristic (ROC) curve, a curve created by plotting the true-positive rate against the false-positive rate at different decision boundaries, which is a measure of the classifier:
$$AUC = \int_{\infty}^{-\infty} TPR(t)FPR'(t)dt.$$

C. Experimental Results

In all the experiments, we use 10-fold cross-validation to evaluate the performance of our classifier. Namely, the dataset is first partitioned into 10 subsets with equal size randomly. In a validation process, a subset is regarded as testing data and the remaining 9 subsets are as training data. Then the process is repeated 10 times such that each subset is used for testing

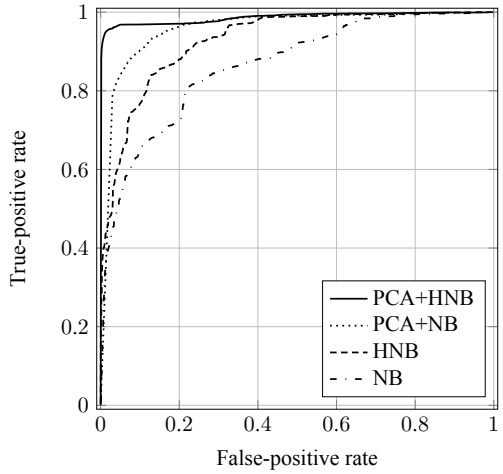

Fig. 5: Performance evaluation as ROC curve for each method.

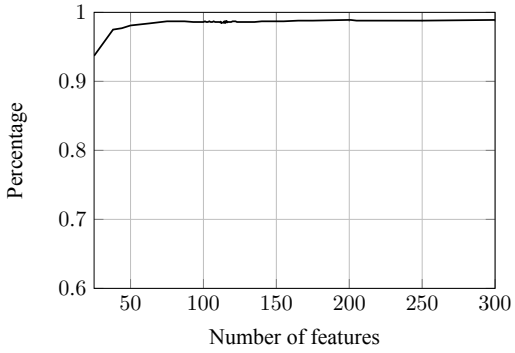

Fig. 6: AUC versus different number of features in the proposed method.

exactly once.

The experimental results are shown in Figure 5 as ROC curves for different decision thresholds of each methods. Since the naive Bayes model does not consider the feature dependencies, it has the worst performance with 81.8% of malware identified and 24.9% false-positive rate. We further consider prior information to improve the performance, but only lower the false-positive rate to around 10% with the true-positive rate nearly unchanged. The HNB classifier has a better performance as it takes the feature dependencies into account. The true-positive rate is 81.4% and the false-positive rate is 11.8%. In addition, considering that the hidden parents in HNB are approximation, we apply PCA to aggregate the variances. The results shows that with PCA the HNB classifier by selecting 118 features can detect 94.5% of malicious applications with 1.0% false-positive rate. The performance of the naive Bayes classifier is also improved: 91.0% of malicious applications are identified with 10.8% false-positive rate.

To select the most relevant features, we calculate mutual information scores on those features whose usage in malware is greater than the benign applications. There are 72,818 distinct features in the dataset including 68,939 API calls and 3,879 permissions. Only 3,367 features satisfy the condition. Figure 6 shows the results of the HNB model with PCA preprocessing on top 300 features with the highest MI scores in terms of AUC, TPR and FPR. From the top 100 to 250 features, we get almost the same results. However, we should choose the number of features as less as possible so that the influence from the benign applications can be mitigated. In our dataset, the top 118 features including 11 permissions and 107 API calls are considered the most fitting parameters with AUC equal to 0.986.

TABLE II: Reimplementation results

	Performance			
	ACC	TPR	FPR	Prec.
DroidAPIMiner	98.7%	96.2%	0.7%	96.3%
Proposed method	98.2%	94.5%	1.0%	94.9%
MAMA	97.8%	91.9%	1.0%	94.9%
Peng et al.	88.8%	80.0%	9.4%	63.1%

1) Performance against Related Works: We compare the performance of our classifier with some similar works: MAMA [10], Yerima et al. [9], Peng et al. [8], DREBIN [7], and DroidAPIMiner [21]. The comparison is not straightforward since: *a*) neither the class distributions nor the instances of all the datasets are identical, and *b*) it is nearly impossible to extract the same features at the specific works without source code. For comparison, we reimplement the approaches as possible as we can. The results of reimplementation are shown in Table II.

MAMA [10] applies random forests algorithms while Peng et al. [8] applies multiple probabilistic methods. Both of them use permissions as their features. We conduct the same experiment as Peng et al. by applying the naive Bayes model with all the permissions as features, but the true-positive rate is down to 80.0% with 9.4% false-positive rate. The reason is that their dataset is collected from 2011 to 2012 and ours is much newer. As described in Section IV-B, the average number of permission requested per application is increasing, which make it more difficult to identify the malware. This performance drop indicates such permission-based classifiers are getting worse day by day.

Yerima et al. [9] and our work are very similar. They not only use Bayesian classification, but also mix API calls, permissions and some system commands as their features. The difference is that they apply the naive Bayes model and we apply HNB with PCA. Moreover, applying the naive Bayes model on our dataset with API calls and permissions only produces a poor result with 81.8% true-positive rate and 24.9% false-positive rate (See Figure 5). The experiment applying the naive Bayes model in MAMA shows the same result as us. The difference of performance implies that the naive Bayes model without preprocessing is not suitable for some datasets other than Android Malware Genome project [14].

As shown in Table II, we improve the performance of the Bayesian classification approaches very close to the perfor-

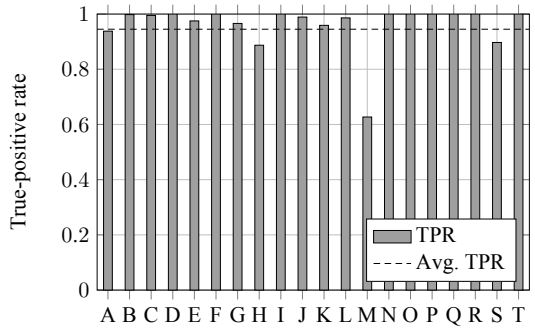

Fig. 7: Detection rate for the top 20 most malware families with 1% false-positive rate.

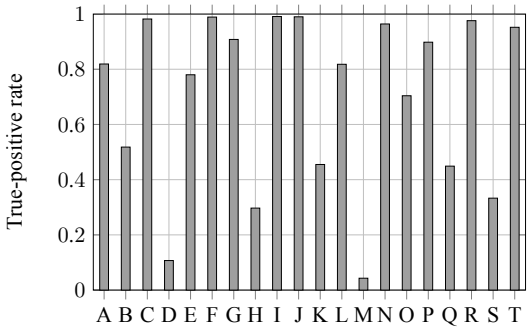

Fig. 8: Detection of unknown families

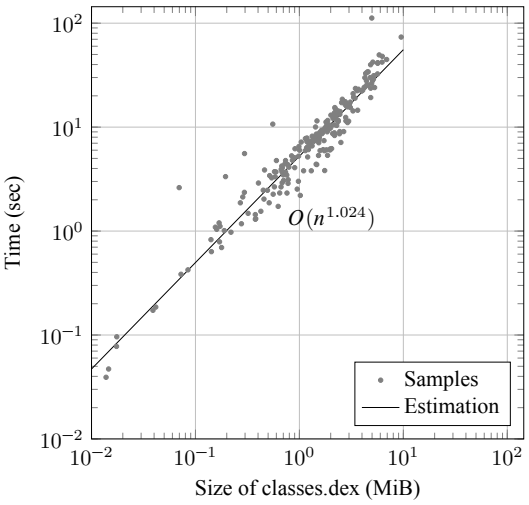

Fig. 9: Runtime analysis of API call retrieval by Androguard.

mance of the discriminative ones in our dataset. One of the approaches DREBIN [7] utilizes linear support vector machine (SVM) [30] on comprehensive feature sets. The precision of DREBIN is low due to their imbalanced dataset. The performance might be better if they reduced the size of benign instances. Since we use a different dataset and the feature sets in DREBIN is unclear, we cannot tell which one is better. DroidAPIMiner [21] is another approach using discriminative model. They select features by the frequency analysis on API calls and apply the k-nearest neighbors (kNN) algorithm to classify. To the best of our knowledge, it outperforms all the related work. For comparison, we reimplement their approach on our dataset. The experimental result are extremely excellent with 98.7% accuracy, 96.2% true-positive rate, 0.7% false-positive rate, and 96.3% precision. However, the kNN algorithm is a memory-based learning method, whose time and memory cost scale with training set size. The optimization for large training sets is a trade-off between time and accuracy.

2) Detection for Malware Families: Figure 7 shows the detection rates of the top 20 most malware families listed in Table I. Our approach is capable of detecting all the malware families with an average true-positive rate of 94.5% at 1.0% false-positive rate. Only 3 malware families have relatively low performance: Boqx (H), Plankton (M), and Coogos (S). Boqx is a malicious repackaged version of a popular gaming application, which is able to download additional executable content from a remote server. Plankton has the dynamic capability of fetching and executing payload at runtime. Coogos is a backdoor Trojan that can receive a remote connection and perform actions on the compromised system.

We conduct an additional experiment to evaluate the ability to detect unknown malware families. The instances from a particular family and 10% of benign applications are testing data, and the others plus the remaining 90% of benign instances are training data. The false-positive rate is fixed at 1%. The results for the top 20 most malware families are shown in Figure 8. Some detection rates drop but slightly as we are unable to remove the variants of a specific malware family without complete evolution information.

3) Runtime Performance: It is of crucial importance that a detection system has reasonable runtime performance. As illustrated in Figure 3, the classification of an unknown sample in our system consists of two phases, the analytic phase and the prediction phase. In this section, we measure execution time of the two phases and study the feasibility. All the experiments are conducted on a machine with Intel Xeon E5-1620 CPU, 3.60 GHz, 32 GB RAM, the Ubuntu 14.04 operating system, Python 2.7.6, and Weka 3.6.11 [31].

In the first phase, we use Androguard to process an APK file to get the API calls and permissions. Since permission retrieval needs parsing XML only, the execution time of the phase is dominated by API call retrieval, i.e., disassembling the DEX file. We randomly select a total of 200 applications from each category, and plot the execution time of API call retrieval against DEX file size on Figure 9. The linear regression on the log-log plot shows a relationship modeled as $y = 5.36x^{1.024}$, that is, the execution time is nearly directly proportional to the DEX file size. Although only 118 features are required, we still have to scan the whole classes.dex to know whether a particular feature exists. It takes an average of 11 seconds to retrieve the features from an application.

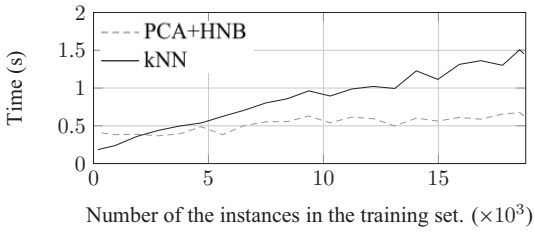

Fig. 10: Testing time on a Nexus S phone.

In the second phase, the feature vector generated in the first phase is transformed by PCA, discretized, and labeled by the model trained offline. The computational complexities of the three operations are $\Theta(n^2)$, $O(n \log k)$, and $O(n^2)$ where n is the number of features and k is the bin number of the discretization. The overall complexity of prediction phase is thus $O(n^2)$. The experimental results are shown in Figure 10, where we also measure the testing time of kNN for comparison. The testing time of the proposed method is increasing in a lower rate as compared to the kNN approach.

VI. Conclusion

We have found that the Bayesian classification models does not fit every dataset, especially the naive Bayes model, which is often seen but has impoverished performance on the Contagio Malware Dump dataset. Besides, since the number of permissions requested by an Android application increases, the permission-based approaches may not work anymore.

In this paper, we propose using the hidden naive Bayes model instead of the naive Bayes model. With the data pre-processed by the principal component analysis, our approach, which selects malicious API calls and permissions as features by mutual information, can achieve 98.2% accuracy, 94.5% true-positive rate at 1.0% false-positive rate. Except the ability to avoid benign attacks, it is also highly capable of detecting unknown malware families provided that they are variants of existing malware. The result shows the reliability and feasibility of the proposed approach.

References

[1] T. Vidas, N. Christin, and L. Cranor, "Curbing android permission creep," in *Proceedings of the Web*, vol. 2, 2011.

[2] [Online]. Available: http://officialandroid.blogspot.tw/2012/02/android-and-security.html

[3] J. Oberheide and C. Miller, "Dissecting the android bouncer," *SummerCon2012, New York*, 2012.

[4] L.-K. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis." in *USENIX Security Symposium*, 2012, pp. 569–584.

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

[6] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 15–26.

[7] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket," 2014.

[8] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Using probabilistic generative models for ranking risks of android apps," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 2012, pp. 241–252.

[9] S. Yerima, S. Sezer, G. McWilliams, and I. Muttik, "A new android malware detection approach using bayesian classification," in *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*, March 2013, pp. 121–128.

[10] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, J. Nieves, P. G. Bringas, and G. Álvarez Marañón, "Mama: Manifest analysis for malware detection in android," *Cybernetics and Systems*, vol. 44, no. 6-7, pp. 469–488, 2013.

[11] M. Parkour, "Contagiodump," 2013. [Online]. Available: http://contagiominidump.blogspot.com

[12] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security." in *USENIX security symposium*, vol. 2, 2011, p. 2.

[13] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 3–14.

[14] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 95–109.

[15] Baksmali. [Online]. Available: https://code.google.com/p/smali/

[16] A. Desnos and G. Gueguen, "Android: From reversing to decompilation," *Proc. of Black Hat Abu Dhabi*, 2011.

[17] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 627–638.

[18] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 217–228.

[19] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Permission evolution in the android ecosystem," in *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 2012, pp. 31–40.

[20] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 235–245.

[21] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *Security and Privacy in Communication Networks*. Springer, 2013, pp. 86–103.

[22] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones," *Communications of the ACM*, vol. 57, no. 3, pp. 99–106, 2014.

[23] A. Y. Ng and M. I. Jordan, "On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes," in *Advances in Neural Information Processing Systems 14*, T. Dietterich, S. Becker, and Z. Ghahramani, Eds. MIT Press, 2002, pp. 841–848.

[24] D. M. Chickering, "Learning bayesian networks is np-complete," in *Learning from data*. Springer, 1996, pp. 121–130.

[25] L. Jiang, H. Zhang, and Z. Cai, "A novel bayes model: Hidden naive bayes," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 21, no. 10, pp. 1361–1371, 2009.

[26] I. Jolliffe, *Principal component analysis*. Wiley Online Library, 2005.

[27] J. C. Gower, "Some distance properties of latent root and vector methods used in multivariate analysis," *Biometrika*, vol. 53, no. 3-4, pp. 325–338, 1966.

[28] M. Nikravesh, I. Guyon, S. Gunn, and L. Zadeh, *Feature Extraction: Foundations and Applications*. Springer, 2006.

[29] S. Liang and X. Du, "Permission-combination-based scheme for android mobile malware detection," in *Communications (ICC), 2014 IEEE International Conference on*. IEEE, 2014, pp. 2301–2306.

[30] N. Cristianini and J. Shawe-Taylor, *An introduction to support vector machines and other kernel-based learning methods*. Cambridge university press, 2000.

[31] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.