Using Probabilistic Generative Models for Ranking Risks of Android Apps

Hao Peng
Purdue University
pengh@cs.purdue.edu

Chris Gates
Purdue University
gates2@cs.purdue.edu

Bhaskar Sarma
Purdue University
bsarma@cs.purdue.edu

Ninghui Li
Purdue University
ninghui@cs.purdue.edu

Yuan Qi
Purdue University
alanqi@cs.purdue.edu

Rahul Potharaju
Purdue University
rpothara@cs.purdue.edu

Cristina Nita-Rotaru
Purdue University
crisn@cs.purdue.edu

Ian Molloy
IBM Research
molloyim@us.ibm.com

ABSTRACT

One of Android's main defense mechanisms against malicious apps is a risk communication mechanism which, before a user installs an app, warns the user about the permissions the app requires, trusting that the user will make the right decision. This approach has been shown to be ineffective as it presents the risk information of each app in a "stand-alone" fashion and in a way that requires too much technical knowledge and time to distill useful information.

We introduce the notion of risk scoring and risk ranking for Android apps, to improve risk communication for Android apps, and identify three desiderata for an effective risk scoring scheme. We propose to use probabilistic generative models for risk scoring schemes, and identify several such models, ranging from the simple Naive Bayes, to advanced hierarchical mixture models. Experimental results conducted using real-world datasets show that probabilistic general models significantly outperform existing approaches, and that Naive Bayes models give a promising risk scoring approach.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Invasive software

General Terms

Security

Keywords

mobile, malware, data mining, risk

1. INTRODUCTION

As mobile devices become increasingly popular for personal and business use they are increasingly targeted by malware. Mobile devices are becoming ubiquitous, and they provide access to personal and sensitive information such as phone numbers, contact lists, geolocation, and SMS messages, making their security an especially important challenge. Compared with desktop and laptop computers, mobile devices have a different paradigm for installing new applications. For computers, a typical user installs relatively few applications, most of which are from reputable vendors with niche applications increasingly being replaced by web-based or cloud services. For mobile devices, one often downloads and uses many applications (or apps) with limited functionality from multiple unknown vendors. Therefore, the defense against malware must depend to a large degree on decisions made by the users. Indeed whether an app is malware or not may depend on the user's privacy preference. Therefore, an important part of malware defense on mobile devices is to communicate the risk of installing an app to users, and to help them make the right decision about whether to choose and install certain apps.

In this paper we study how to conduct effective risk communication for mobile devices. We focus on the Android platform. The Android platform has emerged as one of the fastest growing operating systems. In June 2012, Google announced that 400 million Android devices have been activated, with 1 million devices being activated daily. An increasing number of apps are available for Android. The Google Play (formerly known as Android Market) crossed more than 15 billion downloads in May of 2012, and was adding about 1 billion downloads per month from Dec 2011 to May 2012. Such a wide user base coupled with ease of developing and sharing applications makes Android an attractive target for malicious application developers that seek personal gain while costing users' money and invading users' privacy. Examples of malware activities performed by malicious apps include stealing users' private data and sending SMS messages to premium rate numbers.

One of Android's main defense mechanisms against malicious apps is a risk communication mechanism which warns the user about permissions an app requires before being installed, trusting that the user will make the right decision. Google has made the following comment on malicious apps: *"When installing an application, users see a screen that explains clearly what information and system resources the application has permission to access, such as a phone's GPS location. Users must explicitly approve this access in order to continue with the installation, and they may uninstall applications at any time. They can also view ratings and reviews to help decide which applications they choose to install. We consistently advise users to only install apps they trust."* This approach,

CCS'12, October 16–18, 2012, Raleigh, North Carolina, USA.

however, has been shown to be ineffective. The majority of Android apps request multiple permissions. When a user sees what appears to be the same warning message for almost every app, warnings quickly lose any effectiveness as the users are conditioned to ignore such warnings.

Recently, risk signals based on the set of permissions an app requests have been proposed as a mechanism to improve the existing warning mechanism for apps. In [11], requesting certain permission or combinations of two or three permissions triggers a warning that the app is risky. In [24], requesting a critical permission that is rarely requested is viewed as a signal that the app is risky.

Rather than using a binary risk signal that marks an app as either risky or not risky, we propose to develop risk scoring schemes for Android apps based on the permissions that they request. We believe that the main reason for the failure of the current Android warning approach is that it presents the risk information of each app in a "stand-alone" fashion and in a way that requires too much technical knowledge and time to distill useful information. We believe a more effective approach is to present "comparative" risk information, i.e., each app's risk is presented in a context of comparing it with other apps. We propose to use a risk scoring function that assigns to each app a real number score so that apps with higher risks have a higher score. Given this function, one can derive a risk ranking for each app, identifying the percentile of the app in terms of its risk score. This number has a well-defined and easy-to-understand meaning. Users can appreciate the difference between an app ranked in the top 1% group versus one in the bottom 50%. This ranking can be presented in a more user-friendly fashion, e.g., translated into categorical values such as high risk, average risk, and low risk. An important feature of the mobile app ecosystem is that users often have choices and alternatives when choosing a mobile app. If the user knows that one app is significantly more risky than another for the same functionality, then that may cause the user to choose the less risky one.

To be most effective, we propose the following desiderata for the risk scoring function. First, it should be *monotonic*, in the sense that for any app, removing a permission from its set of requested permissions should reduce the risk score. This way, a developer can reduce the risk score of an app by following the least-privilege principle. Second, apps that are known to be malicious should in general have high risk scores. Third, it is desired that the risk scoring function is simple and relatively easy to understand.

We propose to use probabilistic generative models for risk scoring. Probabilistic generative models [7] have been used extensively in a variety of applications in machine learning, computer vision, and computational biology, to model complex data. The main strength is to model features in a large amount of unlabeled data. Using these models, we assume that some parameterized random process generates the app data and learn the model parameter based on the data. Then we can compute the probability of each app generated by the model. The risk score can be any function that is inversely related to the probability, so that lower probability translates into a higher score.

More specifically, we consider the following models in this paper. In the Basic Naive Bayes (BNB) model, we use only the permission information of the apps, and assume that each app is generated by M independent Bernoulli random variables, where M is the number of permissions. Let θ_m be the probability that the m'th permission is requested (which can be estimated by computing the fraction of apps requesting that permission), then the probability that an app requests a permission is computed by multiplying θ_i's if it requests the i'th permission and $(1 - \theta_i)$ if it does not request the i'th permission. If $\theta_m < 0.5$ for every m, the model has the monotonicity property. The BNB model treats all permissions equally; however, some permissions are more critical than others. To model this semantic knowledge about permissions, we also consider Naive Bayes with informative Priors, which we use PNB to denote. The effect of PNB model is to reduce θ_i when the i'th permission is considered critical. While PNB is slightly more complex than BNB, it has the advantage that requesting a more critical permission results in higher risk than requesting a similarly rare but less critical permission, making it more difficult for a malicious app to reduce its risk by removing unnecessary permissions.

We also investigate several sophisticated generative models. In the Mixture of Naive Bayes (MNB) model, we assume that the dataset is generated by a number of hidden classes, each is parameterized by M independent Bernoulli random variables; these hidden classes are shared among all categories. Each category has a different multinomial distribution describing how likely an app in this category is from a given hidden class. We also develop a Hierarchical Bayesian model, which we call the Hierarchical Mixture of Naive Bayes (HMNB) model. This is a novel extension to the influential Latent Dirichlet Allocation (LDA) [8] model to binary observations that integrates categorical information with hidden classes and allows permission information to be shared between categories.

We have conducted extensive experiments using three datasets: Market2011, Market2012, and Malware. Market2011 consists of 157,856 apps available at Android Market in February 2011. Market2012 consists of 324,658 apps available at Google Play in February/March 2012. Malware consists of 378 known malwares. Our experiments show that in terms of assigning high risk scores to malware apps, all generative models significantly outperform existing approaches [11, 24]. Furthermore, while PNB is simpler than MNB and HMNB, its performance is almost the same as MNB, and very close to the best-performing HMNB model. Based on these results, we conclude that PNB is good risking scoring scheme.

In summary, the contributions of this paper are as follows:

- We introduce the notion of risk scoring and risk ranking for Android apps, to improve risk communication for Android apps, and identify three desiderata for an effective risk scoring scheme.
- We propose to use probabilistic generative models for risk scoring schemes, and identify several such models, ranging from the simple Basic Naive Bayes (BNB), to advanced hierarchical mixture models.
- We conduct extensive evaluations using real-world datasets. Our experimental results show that probabilistic general models significantly outperform existing approaches, and PNB makes a promising risk scoring approach.

The rest of the paper is organized as follows. We present a description of the Android platform and the current warning mechanism in Section 2. Section 3 discusses the datasets that we have collected. In Section 4 we discuss different generative models for risk scoring. We then present experimental results in Section 5, and discuss other findings in Section 6. We finish by discussing related work in Section 7 and concluding in Section 8.

2. ANDROID PLATFORM

In this section we provide an overview of the current defense mechanism provided by the Android platform and discuss its limitations.

2.1 Platform Ecosystem

Android is an open source software stack for mobile devices that includes an operating system, an application framework, and core applications. The operating system relies on a kernel derived from Linux. The application framework uses the Dalvik Virtual Machine. Applications are written in Java using the Android SDK, compiled into Dalvik Executable files, and packaged into `.apk` (Android package) archives for installation.

The app store hosted by Google is called Google Play (previously called Android Market). In order to submit applications to Google Play, an Android developer first needs to obtain a publisher account. After submission, each *.apk* file gets an entry on the market in the form of a webpage, accessible to users through either the Google Play homepage or the search interface. This webpage contains meta-information that keeps track of information pertaining to the application (e.g., name, category, version, size, prices) and its usage statistics (e.g., rating, number of installs, user reviews). This information is used by users when they are deciding to install a new application.

Google recently started the Bouncer [3] service, which provides automated scanning of applications on Google Play for potential malware. Once an application is uploaded, the service immediately [3] starts analyzing it for known malware, spyware and trojans. It also looks for behaviors that indicate an application might be misbehaving, and compares it against previously analyzed apps to detect possible red flags. Bouncer runs every application on their cloud in an attempt to detect hidden, malicious behavior, and analyzes developer accounts to block malicious developers.

Bouncer does not fully solve the security and privacy problems of Android. First, the line between malicious apps and non-malicious apps is very blurred. The behavior of many apps cannot be classified as malicious, yet many users will find them risky and intrusive. Bouncer has to be conservative when identifying apps as malicious to prevent legitimate complaints from developers and backlash from users for instrumenting a walled garden. Second, details about Bouncer are fairly unknown to the security community. At the time of writing this paper, except for the official blog post by Google [3], there are no details about how Bouncer works nor what algorithms it uses to detect malicious apps. Third, researchers have found multiple ways to bypass Bouncer and upload malware on Google Play. For example, a malicious app can try to detect that it is running on Bouncer's emulated Android device, and refrain from performing any malicious activity, or malware can perform malicious activities only when triggered by certain conditions, such as time.

Other third party app websites exist, e.g., Amazon Appstore for Android, GetJar, SlideMe Market, etc. Currently, these third-party app stores have varying degrees of security associated with them.

2.2 In-Place Security and its Limitations

The Android system's in-place defense against malware consists of two parts: *sandboxing* each application and *warning* the user about the permissions that the application is requesting. Specifically, each application runs with a separate user ID, as a separate process in a virtual machine of its own, and by default does not have permissions to carry out actions or access resources which might have an adverse effect on the system or on other apps, and have to explicitly request these privileges through permissions.

In tandem with the sandboxing approach is a risk communication mechanism that communicates the risks of installing an app to a user, hoping/trusting that the user will make the right decision. When a user downloads an app through the Google Play website, the user is shown a screen that displays the permissions requested by the application and the warnings about the potential damages when these permissions are misused. These warnings are worded with a high degree of seriousness (See Table 1 for Android's warnings of some permissions). This provides a final chance to verify that the user is allowing the application access to the requested resources. Installing the application means granting the application all the requested permissions. A similar interface exists when a user is browsing applications from a mobile device.

Despite its serious-wording, Android's current permission warning approach has been largely ineffective. In [15], Felt *et al.* analyzed 100 paid and 856 free Android applications, and found that "*Nearly all applications (93% of free and 82% of paid) ask for at least one 'Dangerous' permission, which indicates that users are accustomed to installing applications with Dangerous permissions. The INTERNET permission is so widely requested that users cannot consider its warning anomalous. Security guidelines or anti-virus programs that warn against installing applications with access to both the Internet and personal information are likely to fail because almost all applications with personal information also have INTERNET.*"

Felt *et al.* argued "*Warning science literature indicates that frequent warnings de-sensitize users, especially if most warnings do not lead to negative consequences [29, 17]. Users are therefore not likely to pay attention to or gain information from install-time permission prompts in these systems. Changes to these permission systems are necessary to reduce the number of permission warnings shown to users.*"

While such ineffectiveness has been identified and criticized [15, 29, 17], no alternative has been proposed. We argue that a promising alternative is to present relative or comparative risk information. This way, users can select apps based on easy-to-consume risk information. Hopefully this will provides incentives to developers to better follow the least-privilege principle and request only necessary permissions.

Comparison with UAC: There is a parallel between Android's permission warning and Windows' User Account Control (UAC). Both are designed to inform the user of some potentially harmful action that is about to occur. In UAC's case, this happens when a process is trying to elevate it's privileges in some way, and in Android's case, this happens when a user is about to install an app that will have all the requested permissions.

Recent research [19] suggests the ineffectiveness of UAC in enforcing security. Motiee *et al.* [19] reported that 69% of the survey participants ignored the UAC dialog and proceeded directly to use the administrator account. Microsoft itself concedes that about 90% of the prompts are answered as "yes", suggesting that "users are responding out of habit due to the large number of prompts rather than focusing on the critical prompts and making confident decisions" [12].

According to [12] in the first several months after Vista was available for use, people were experiencing a UAC prompt in 50% of their "sessions" - a session is everything that happens from logon to logoff or within 24 hours. With Vista SP1 and over time, this number has been reduced to about 30% of the sessions. This suggests that UAC has been effective in incentivizing application developers to write programs without elevated privileges unless necessary. An effective risk communication approach for Android could have similar effects.

3. DATASETS

In this section, we describe the two types of datasets we used in our study of Android app permissions. Below we describe the datasets and their characteristics.

3.1 Datasets Description

Market Datasets: We have collected two datasets from Google Play spaced one year apart. Market2011, the first dataset, consists of 157,856 apps available on Google Play in February 2011. Market2012, the second dataset, consists of 324,658 apps and has been collected in February 2012. For each app, we have the application meta-information consisting of the developer name, its category and the set of permissions that the app requests. We assume that apps in these two datasets are mostly benign. While we believe that a small number of malicious apps may be present in them, we assume that these datasets are dominated by benign ones. We leverage the Market2011 dataset for our model generation and testing, use Market2012 dataset for validation and market evolution analysis.

Malware Dataset: Our malware dataset consists of 378 unique `.apk` files that are known to be malicious. We obtained this dataset from the authors of [31]. For each malware sample, we extract the permissions requested using the AndroidManifest.xml file present inside the package file. For these malicious apps we do not have their category information.

3.2 Data Cleansing

In the two market datasets, we have observed the presence of thousands of apps that have similar characteristics. This kind of "duplication" can occur due to the following reasons:

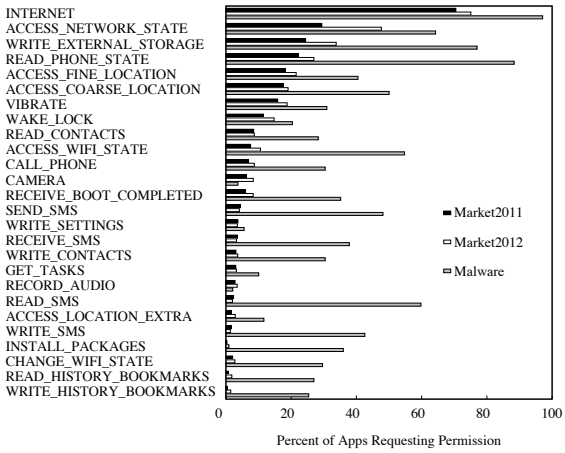
- **Slight Variations (R1)**: One developer may release hundreds or even thousands of nearly identical apps that provide the same functionality with slight variation. A few examples include wallpaper apps, city or country specific travel apps, weather apps, or themed apps (i.e., a new app with essentially the same functionalities can be written for any celebrity, interest group,etc.) such as the one presented in Table 1 in Section 6.

- **App Maker Tools (R2)**: There are a number of tools [1, 2] that enable non-programmers to create Android apps. Often times many apps that are generated by these tools have similar app names and the same set of permissions. This occurs when the developer just uses the default settings in the tool.

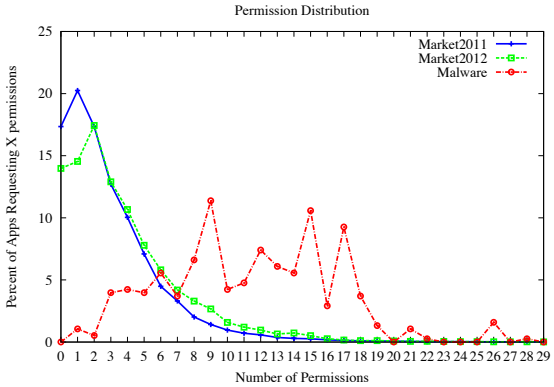
We decided to consolidate duplicate apps from the same developer (R1) into a single instance in the dataset to prevent any single developer from having a large impact on the generated probabilistic model. We detect apps due to R1 by looking for instances where apps belonging to the same developer have the same set of permissions. This is a likely indication that developers are uploading many applications with minor variations in the app content.

We decided to keep apps due to R2 unchanged in the datasets. We do this because: (1) we observed instances where apps due to R2 have different functionality and many developers using these tools do modify the permissions given to their app and (2) the line between such apps and all apps that use a specific ad-network which require a certain set of permissions is blurry.

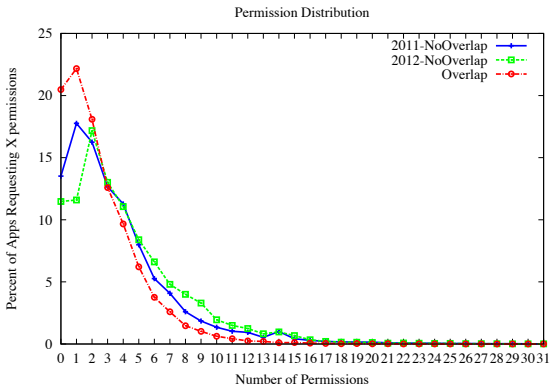
After cleansing is complete we have 71,331 apps in the 2011 market dataset, and 136,534 apps in the 2012 market dataset. This represents a reduction of around 55%, and demonstrates the prevalence of apps that are slight variations of other apps, justifying our



(a) The top 20 most used permissions in the datasets as a percent of apps that request those permissions. Due to overlap in the most used permissions, we need to show 26 permissions to cover the most used in all datasets. 21st for Market 2012, and last 5 for Malware.



(b) The percent of apps that request a specific number of permissions for each dataset.



(c) The percent of apps that request a specific number of permissions in the market datasets. Apps that only appear in 2011, only in 2012, and the intersection of those two datasets

Figure 1: Permission information for various data sets

decision to combine these so as not to allow one developer to overly influence any model.

For some experiments, we break up market dataset into three sets. The intersection of the 2011 and 2012 data is called 'overlap', this contains 38,024 apps which have the same name and permissions in the two datasets. Then we have 2011-NoOverlap, the 2011 dataset with this overlap removed, containing 33,307 apps, and 2012-NoOverlap, the 2012 dataset with this overlap removed, containing 98,510 apps.

3.3 Dataset Discussion

The top 20 most frequently requested permissions in each dataset are presented in Figure 1(a). There are 26 permissions in this table, which represent the top 20 for all 3 datasets. ACCESS_LOCATION_EXTRA_COMMANDS was added for Market2012, and the last 5 were added for the malware dataset. For some permissions, the percentage of malware apps requesting a specific permission is much higher than those in the market dataset. For example, READ_SMS is requested by 59.78% of the malicious apps, but only 2.33% from Market2011, and 1.98% from Market2012. This might be due to the fact that a class of malware apps attempt to intercept messages between a mobile phone and a bank for out-of-band authentication.

Another observation from Figure 1(a) is that for almost every permission a higher percent of apps in Market2012 request it when compared to the Market2011 dataset. This shows a trend that proportionally more applications are requesting sensitive permissions. The one notable exception to this is related to SMS, where Market2012 actually saw a slight decrease for all permissions related to SMS.

Figure 1(b) shows the percent of apps that request different numbers of permissions. From this graph, we observe in general, malicious apps are requesting more permissions than the ones in the market datasets. However, there are many market dataset apps that are requesting many permissions as well. Between Market2011 and Market2012, we also see a confirmation that apps are requesting a greater number of permissions on average. With proportionally fewer apps requesting 0 or 1 permissions in Market2012, and then for two permissions and greater, we see slight gains in the percent of apps requesting permissions over Market2011. Overall, this information is an indication that the malicious apps are requesting permissions in different ways then normal apps, and leads us to believe that looking at permission information is in fact promising. It also shows that there may be a slow evolution in the market dataset.

Figure 1(c) shows a similar graph when we divide the datasets into the overlap dataset and the two datasets with overlapping apps removed. Interestingly, apps in the overlap dataset, which are the "long-living" and stable apps generally request fewer permissions than other apps.

4. MODELS

We aim at coming up with a risk score for apps based on their requested permission sets and categories. Let the i'th app in the dataset be represented by $a_i = (c_i, \mathbf{x}_i = [x_{i,1}, \ldots, x_{i,M}])$, where $c_i \in C$ is the category of the i'th app, M is the number of permissions, and $x_{i,m} \in \{0,1\}$ indicates whether the i'th app has the m'th permission. Our goal is to come up with a risk function $\mathsf{rscore} : C \times \{0,1\}^M \to \mathbb{R}$ such that it satisfies the following three desiderata. First, the risk function *should* be monotonic. This condition requires that removing a permission always reduces the risk value of an app, formalized by the following definition.

DEFINITION 1 (MONOTONICITY). *We say that a risk scoring function* rscore *is* ***monotonic*** *if and only if for any $c_i \in C$ and any $\mathbf{x}_i, \mathbf{x}_j$ such that*

$$\exists k \, (\mathbf{x}_{i,k} = 0 \wedge \mathbf{x}_{j,k} = 1 \wedge \forall m (m \neq k \Rightarrow \mathbf{x}_{i,m} = \mathbf{x}_{j,m})) \\ \Rightarrow \mathsf{rscore}(c_i, \mathbf{x}_i) < \mathsf{rscore}(c_i, \mathbf{x}_j).$$

The second desideratum is that malicious apps generally have high risk scores. And the third is that the risk scoring function is simple to understand.

Given any risk function, we can assign a risk ranking for each app relative to a set A of reference apps, which can be, e.g., the set of all apps available in Google Play:

$$\mathsf{rrank}(a_i) = \frac{|\{a \in A \mid \mathsf{rscore}(a) \geq \mathsf{rscore}(a_i)\}|}{|A|}$$

If an app has a risk ranking of 1%, this means that the app's risk score is among the highest 1 percent.

The above gives a risk ranking relative to all apps in all categories. An alternative is to rank apps in each category separately, so that one has a risk ranking for an app relative to other apps in the same category.

Probabilistic generative models. We propose to use probabilistic generative models for risk scoring. That is, we assume that some parameterized random process generates the app datasets and learn the parameter value θ that best explain the data. Next, for each app we compute $p(a_i|\theta)$, the probability that the app's data is generated by the model.

The risk score of an app can be any function that is monotonically decreasing with respect to the probability of an app being generated, such that a lower probability means a higher risk score. For example, using $\mathsf{rscore}(a_i) = -\ln p(a_i|\theta)$ satisfies the condition.

In the rest of this section we describe three generative models—from simple Naive Bayesian models, to mixture of Naive Bayes models and to novel hierarchical Bayesian models. We present estimation methods to learn the parameters for these models from the data, and evaluate whether they satisfy our desiderata.

4.1 Naive Bayes Models

In the Naive Bayes models, we ignore the category information c_i; thus each app is given by $\mathbf{x}_i = [x_{i,1}, \ldots, x_{i,M}]$. We assume that each $\mathbf{x}_i$ is generated by M *independent* Bernoulli random variables, where M is the number of permissions:

$$p(\mathbf{x}_i) = \prod_{m=1}^{M} p(x_{i,m}) = \prod_{m=1}^{M} \theta_m^{x_{i,m}} (1-\theta_m)^{(1-x_{i,m})} \quad (1)$$

where $\theta_m \equiv p(x_{i,m} = 1)$ is the Bernoulli parameter.

To avoid overfitting in our estimation (i.e., fitting the model to noise), we use a Beta prior $\text{Beta}(\theta_m|a_0, b_0)$ over each Bernoulli parameter θ_m. Using this prior, the Maximum a posteriori (MAP) estimation is

$$\hat{\theta}_m = \frac{\sum_i^N x_{i,m} + a_0}{N + a_0 + b_0} \quad (2)$$

where N is the total number of apps for this Naive Bayes model estimation.

The Basic Naive Bayes Model (BNB). In the Basic Naive Bayes (BNB) mode, we use *uninformative* prior and set $a_0 = b_0 = 1$, so that the Beta prior becomes a uniform distribution on [0,1]. With

such an uninformative prior, $\hat{\theta}_m$ is very close to the the frequency of the m'th permission being requested in the dataset.

The BNB model is easy to explain, satisfying the third desideratum. Furthermore, if $\theta_m < 0.5$ for every m, then the probability provided by this model satisfies the monotonicity property. Changing any $\mathbf{x}_{i,m}$ from 0 to 1 changes the probability by a factor of $\frac{\theta_m}{1-\theta_m}$, which is less than 1 when $\theta_m < 0.5$, and thus decreases the probability and increases the risk score. As there is only one permission, namely Internet, requested by over 50% of the apps, removing the INTERNET permission from the feature set suffices to ensure the monotonicity property. Finally, the BNB model intuitively satisfies the second desideratum, i.e., known malicious apps generally have lower generated probabilities, because as we have seen in Section 3.3, malicious apps generally request more permissions.

NB with Informative Priors (PNB). BNB treats all permissions equally, and a malicious app can reduce its risk by not requesting rare permissions that are not critically needed for carrying out malicious activities. We thus consider a Naive Bayes model with *informative* priors to incorporate semantic information of app permissions. Such approach is commonly used in Naive Bayes models to model knowledge not available in the dataset. The desired goal is to make requesting a more critical permission to increase risk more than requesting a less critical one, even though the two permissions have similar frequencies.

To identify critical permissions, we start from a list of 26 permissions identified in [24] as critical. We remove the INTERNET permission, and add another that we believe is critical, namely INSTALL_PACKAGES. Furthermore, among the 26 permissions, we manually selected 9 of them as very high risk permissions.[1]

To incorporate the semantic information in the Naive Bayes models, we uenckse informative Beta prior distributions Beta$(\theta_m|a_m, b_m)$: for the most risky 9 permissions, we set $a_m = 1, b_m = 2N$ and N is the number of apps in our data set, discouraging the use of these permissions; for the other 17 risky permissions, we set $a_m = 1, b_m = N$ with less penalty effect; and for the remaining permissions, we set $a_m = 1, b_m = 1$ as in BNB models.

When compared with BNB, PNB is slightly more complex than BNB. However, it has the advantage that requesting a more critical permission results in higher risk, when compared with requesting a similarly rare but less critical permission. One key benefit of PNB is that it is more difficult for malware apps to reduce their risks by removing rare permissions that they do not need, since it likely needs some of the critical permissions to carry out its malicious activities. For this reason, we prefer PNB to BNB when other things are equal.

4.2 Mixture of Naive Bayes (MNB) Models

The assumption in BNB and PNB that all apps follow a simple factorized Bernoulli distribution does not appear to be very realistic. Thus, we develop more sophisticated probabilistic generative models and experimentally compare the effectiveness of BNB with these models.

We improve the Naive Bayes model by assuming each app is sampled from multiple—instead of only one—*latent* topics, each of which follows a factorized Bernoulli distribution. Unlike the Naive Bayes model, this mixture model allows us to use different latent topics to capture different aspects of the apps. These topics could describe fine grained classes of applications, such as geotagging apps that request LOCATION, INTERNET, and CAMERA permissions, or applications that leverage common frameworks.

Specifically, we use an unknown indicator variable $z = 1, \ldots, K$ (K is the number of latent topics) to represent which topic an app is sampled from. We assign an uninformative uniform prior over z and assume that the topic distribution is the same as the Naive Bayes model conditioned on z; that is, $p(\mathbf{x}_i|z, \boldsymbol{\theta}_z) = \prod_{m=1}^{M} p(x_{i,m}|z, \theta_{zm})$ is a factorized Bernoulli distribution where $\boldsymbol{\theta}_z = [\theta_{z1}, \ldots, \theta_{zM}]$. Let $\Theta = [\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_k]$ denote parameters for the app distributions for all the topics. Then the probability of the data is

$$p(\mathbf{x}|\Theta) = \sum_z p(z) \prod_{i=1}^{N} p(\mathbf{x}_i|z, \boldsymbol{\theta}_z), \tag{3}$$

which is a mixture of Naive Bayes models.

To obtain the MAP estimation of both assignments, we use an expectation maximization approach that loops over two steps, Expectation (E) and Maximization (M) steps, until convergence. In the E step, we compute the posterior of z given the current estimate of Θ:

$$p(z = k|\mathbf{x}, \Theta) = \frac{\prod_m \theta_{k,m}^{\sum_{i=1}^N x_{i,m}} (1 - \theta_{k,m})^{N - \sum_{i=1}^N x_{i,m}}}{\sum_k \prod_m \theta_{k,m}^{\sum_{i=1}^N x_{i,m}} (1 - \theta_{k,m})^{N - \sum_{i=1}^N x_{i,m}}}$$

In the M step, we maximize the expected joint probability $Q = \sum_{i=1}^{N} \mathbf{E}_z[\ln p(x_i|z, \Theta) + \ln p(z) + \ln p(\Theta)]$. Note that we use the updated $p(z = k|\mathbf{x}, \Theta)$ in the E step to obtain the expectation. We thus obtain

$$\theta_{km} = \frac{\sum_i^N p(z = k|\mathbf{x}, \Theta) x_{i,m} + a_0}{\sum_i^N p(z = k|\mathbf{x}, \Theta) + a_0 + b_0}. \tag{4}$$

MNB models, however, no longer guarantee the monotonicity property. We have observed that the learned hidden topics can request certain permissions with probability over 0.5, resulting in the estimated θ_{km} being greater than 0.5. When this happens, the monotonicity property does not hold.

Mixture of Naive Bayes with Categories (MNBC). We also extend MNB to consider category information and call the resulting models Mixture of Naive Bayes with Categories (MNBC). In MNBC, teh latent topics are shared among all categories, but each category has a different multinomial distribution describing how likely an app in this category is from a particular latent topic.

4.3 Hierarchical Mixture of Naive Bayes (HMNB) Models

Finally, we develop Bayesian hierarchical mixture models that we can train using apps across all categories and, at the same time, account for the difference between categories. We still produce a mixture model for each category. To share information between categories we set the latent topics to be the same across categories and sample the probabilities of choosing these topics from a common Dirichlet distribution—thus these probabilities (i.e., mixture weights) are similar. Our model extends Latent Dirichlet Allocation (LDA) models [8], a popular document model, to the case of binary vector observations (each app corresponds to a word in a document and each category is a document in the latent Dirichlet allocation models).

Let us succinctly denote the permissions of app i in category c by $\mathbf{x}_{ci}$, the parameter in the multinomial topic distribution for category c by ψ_c, the topic assignment variable for each each app i in category c by z_{ci}, and the hyparameter of the Dirichlet prior on

[1]They are ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION, PROCESS_OUTGOING_CALLS, CALL_PHONE, READ_CONTACTS, WRITE_CONTACTS, READ_SMS, SEND_SMS, INSTALL_PACKAGES.

the topic distribution by α. Then formally speaking, we have the following stochastic data generation process:

1. For each topic k and permission m, draw the app probabilities $\theta_{k,m} \sim \text{Beta}(a_0, b_0)$.
2. For each category c, sample the parameter for topic distributions $\psi_c \sim \text{Dir}(\alpha)$.
3. For each app i in category c,
   (a) Sample the topic assignment $z_{ci} \sim \text{Multi}(\psi_c)$.
   (b) Generate the permissions via the factorized Bernoulli distribution (let $z_{ci} = k$)
   $\mathbf{x}_k \sim \prod_m \theta_{k,m}^{\sum_{i=1}^N x_{im}} (1-\theta_{k,m})^{N-\sum_{i=1}^N x_{im}}$.

To estimate this Bayesian model, we develop a variational algorithm. It enables us to accurately approximate the exact Bayesian posterior distributions of the model parameters with a low computational cost. We give the detailed variational updates in the Appendix.

5. EXPERIMENTAL RESULTS

In the experiments we aim at understanding how well the different models satisfy the second desideratum, namely, able to assigning high risks to known malware apps, and compare them to methods in the literature [11, 24].

Methodology. Most of our experiments are conducted with the 2011 dataset, with 10 fold cross validation. We divide the 2011 dataset randomly into ten groups. In each of the 10 rounds, we choose one different group as the test dataset, and the remaining 9 groups as the training dataset. The models are trained on the training set, the generated model is used to compute the probabilities of apps in the testing set and the malware dataset, and rank them together.

When reporting the results, we use ROC curves, which plot the true positive rate against false positive rate if one chooses a particular risk value as indicative of malicious app. We use Area Under Curve (AOC) to quantify the quality of the ROC curves for a method. Here, AUC is the probability that a randomly selected malicious application will have a higher risk score than a randomly selected benign application. When reporting AUC values resulted from 10-fold cross validation, we plot the mean and stand error of the AUCs of the ten rounds.

Parameter Selection. Both MNB and HMNB can be used with different parameters, and we need to select the best parameters for them to compare with other methods. One parameter is the number of hidden topics. Another parameter is how to use category information. This is needed because malware apps do not have category information. Thus when we compute the probability of apps in the test dataset, we also strip their category information.

To estimate an app's likelihood using the MNB model, there are a few ways to choose when we do not know its category information. The first method, called 'max', is to compute the probability of the app for every category and choose the maximum probability, that is the category in which the app fits the best, and assume that the app was in that category. The second method, called 'mean', is to compute the app's probability for every category and take the weighted average of all probabilities. For HMNB model, in addition to the previous two methods, we can also use the mean of our Dirichlet prior as the topic distribution to compute the probability. This method is called 'prior' method.

Figure 2 shows the AUC values for choosing different parameters for MNB and HMNB. From our experiments, we find that the maximum mean of AUC for MNB model is achieved by using 'max' method with 5 hidden classes. And the maximum for HMNB is achieved by using 'mean' method with 80 hidden classes. We use these parameters when comparing with other methods.

Comparing Different Methods. In Figure 3, we compare the generative models with other approaches in the literature. Figure 3(a) shows the ROC curves. Because several curves are clustered together, we use Figure 3(b) to show a close-up of the ROC curves for x axis of up to 0.1. Figure 3(c) show the AUC values.

The methods we compare against include Kirin, RCP, and RPCP. Kirin [11] identifies 9 rules for apps to be considered risky. As Kirin is represented by a single decision point, we only illustrate it as a point in Figure 3(a), and has no AUC value. It can identify close to 39% malware apps at 4% false positive rate. RCP and RPCP are proposed in [24]; they rely on the rarity of critical permissions and the rarity of pairs of critical permissions.

We note that all generative models have AUC values of over 0.94; they significantly outperform RCP and RPCP. The results clearly show that HMNB is best performing, with MNB, BNB, and PNB close behind and almost the same. We note that even a difference of 0.01 is statistically significant given the small standard deviation. And the difference between the generative models and other methods is clearly seen in the ROC curves.

Permissions vs. Risk Scores. The fact that HMNB has the highest AUC makes it somewhat attractive as a risk scoring method. We know that it is not guaranteed to have the monotonicity property; however, it is possible that it preserves the property in most cases. To check whether this is the case, in Figure 4 we plot the average number of permissions for each percentile of the apps in the market2011 dataset, when they are ranked by the risk value according to the PNB model and to the HMNB model. It is clearly seen that in the PNB model the average number of permissions is almost non-decreasing as the risk goes up. On the other hand, in the HMNB model we observe apps with large number of permissions that have low risk. This suggests that HMNB flatly fails the monotonicity requirement.

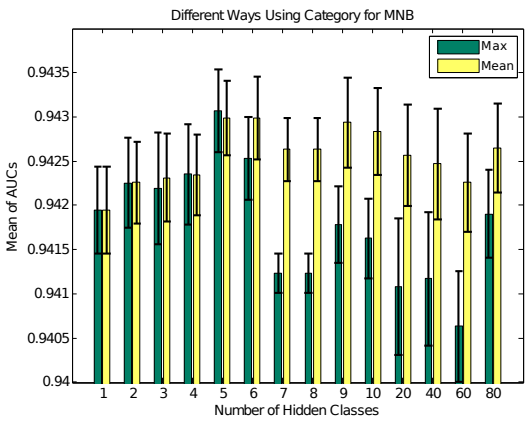
Model Stability. Finally, we conducted experiments to check whether models trained on one dataset can be used without retraining to compute the risk scoring on a new dataset. For this purpose, we use the divided datasets described in Section 3. That is the overlap data between 2011 and 2012, and the 2011 dataset with overlap removed and 2012 dataset with overlap removed.

For each of the six possible ordered pairs, we train on one dataset and then test on the other together with the malware dataset. Figure 5 shows the result. Somewhat interestingly, when testing on the overlap dataset, training either on the 2011-NoOverlap dataset or the 2012-NoOverlap dataset gives excellent result. However using any other combination leads to results that perform worse. This is to some degree to be expected from Figure 1(c). As the "overlap" apps generally request fewer permissions than the other two datasets. The other apps appear to be more varied and require training using part of them to get good results.

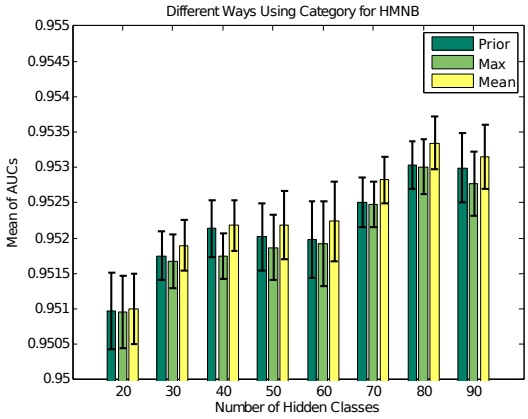
As we have seen in Figure 1 the permission data has changed over time. Therefore, if a system like this were to be implemented, the models should be periodically regenerated to achieve the best results and to keep up to date with the trends that are occurring within the market.

6. DISCUSSION

In the introduction, we mention that while Windows UAC may not be very effective in helping the users make more secure decisions, one of its advantages is that it encouraged developers to make

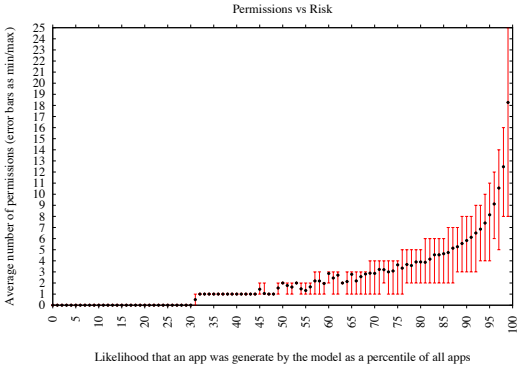


(a) Different number of hidden components for MNB

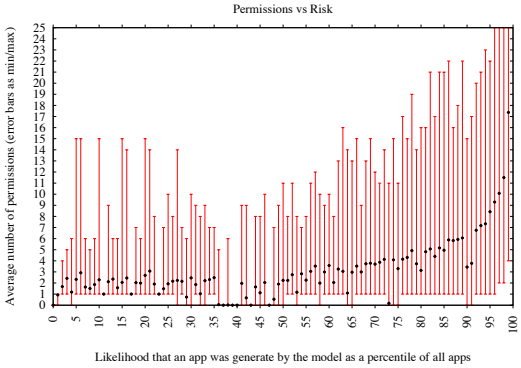


(b) Different number of hidden components for HMNB

Figure 2: Parameter selection for different number of hidden classes. Mean, Max and Sum represent different methods to relate the malicious applications, which don't contain category information, into a system which utilizes category information.



(a) PNB



(b) HMNB80

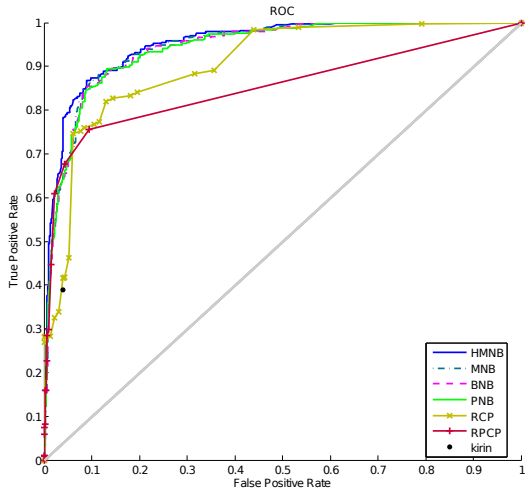
Figure 4: Average number of permissions for every 1% percent division of apps, sorted in descending order on the basis of likelihood. The points represents the average number of permissions requested, and the error bars indicate the min and max at that percentile

conservative decisions in order to improve the user experience by avoiding UAC prompts. One possible positive result of assigning a risk to each application is that it generates a feedback mechanism for the developers which could encourage them to reduce the risk that an app introduces to a mobile device. In essence, an effective risk score mechanism may lead to different decisions by users, creating an economic motivation for developers to reduce the risk of an application. It is also possible that this mechanism could drive additional revenue through application markets since if users are concerned enough to use lower risk applications, then they might be willing to purchase different apps as a low risk alternative.

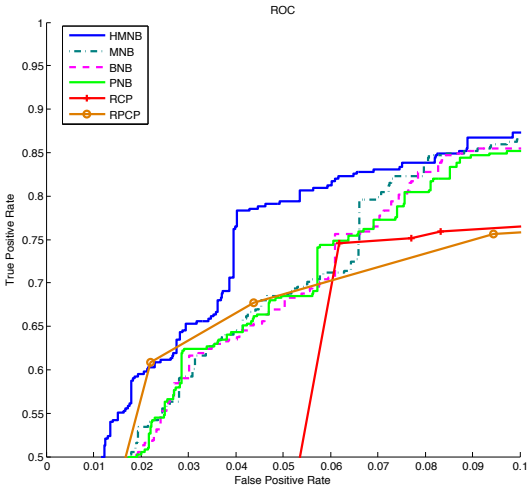
The goal of creating a simple feedback mechanism is the motivation behind our recommendation for the PNB model as an effective risk communication mechanism. This model, with the monotonic property, gives direct feedback to a developer who wishes to lower the risk score of an app. This is demonstrated in Figure 4(a), where the number of permissions directly correlates to the relative risk of an app. There is some variation in this figure because some permissions introduce more risk then others; however, the mathematical properties of PND is such that removing a permission from a set of permissions always reduces the risk score, and adding one permission always increases the risk score.

In the rest of this section, we discuss a particularly interesting app. The application presented in Table 1 represents more than a thousand applications by the same developer with different keywords. This set of apps intercepts all text messages and displays the message on the screen with a new background based on the keyword. Looking at the app's decompiled code it does not appear to be performing any obviously malicious tasks; however, depending on a user's definition of privacy, it could be considered a risky application. One major reason for the high permission count is that this app contains several different ad networks, each of which requests different permissions to achieve their data collection requirements to show relevant adds. The ad networks along with the general functionality of the app leads to 17 different permissions, many of which could have serious privacy issues if misused. Sending and receiving SMS messages is part of the core functionality of the app, however, the ability to read the contact list is used in order to extract names of contacts given the phone number. The app also extracts the user's phone number in order to send a test text message. Additionally, the app collects the email address of the user to notify them that a new app for a specific keyword has been generated. While there is no obvious data leakage beyond what one would expect, there is data leakage over time. That is to say, they are not collecting and exfiltrating all of this information off the phone the first time the app runs, but over time, they are able to paint a picture of the user when they activate different functionality.

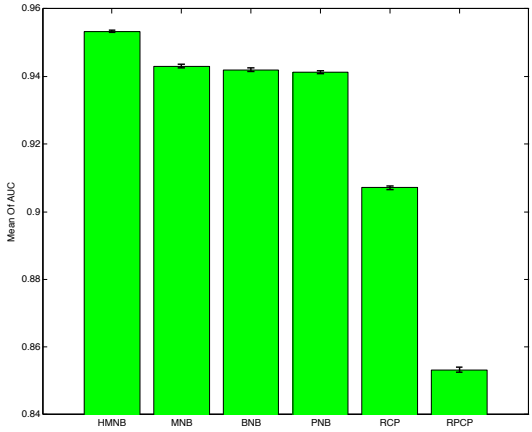
The application also has 2 permissions that are requested but un-



(a) ROC for the best performing parameters for each method.



(b) Close up of Figure 3(a) to capture performance differences in the first 10% false positives



(c) AUC for ROC curves presented in Figure 3a

Figure 3: Comparison of different models using the best performing parameters for each models

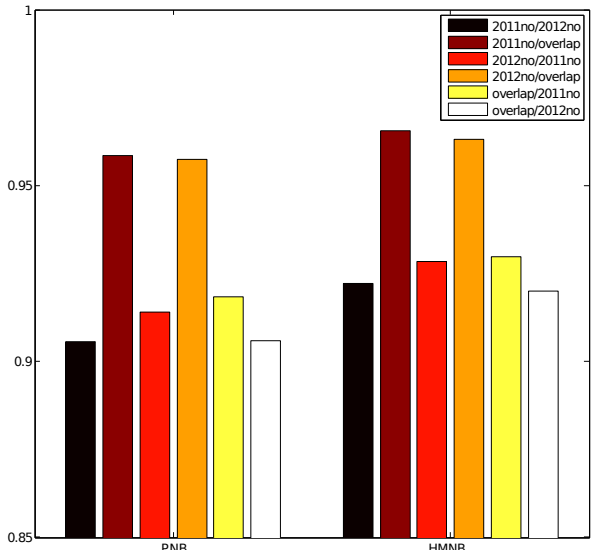


Figure 5: Comparison of 2011 and 2012 Data for PNB and HMNB models. 'no' = no overlap, 'over' = only overlap. 'first/second' means the first dataset was used to train, and the second dataset was used to test along with the malware. Then the AUC was generated and generated.

used, one of these is the permission to intercept phone calls. While most of the other permissions can be justified by some functionality in the app, either from the app itself or the related ad networks, this one cannot be justified. We note that even though an app may not use this permission in the current version, the fact that it has requested this permission still introduces some risk to the user. The reason for the risk is that during an update, if a new version of the app contains the same permissions as the previous version, then the app update can occur silently. Whereas if the app requests new permissions, then the user is notified that the app is changing its requested permissions. So just requesting a permission, even if it is not used, does increase the overall potential risk of the app in this sense.

7. RELATED WORK

Felt *et al.* [13] use static analysis to determine whether an Android application is overprivileged. It classified an application as overprivileged if the application requested a permission which it never actually used. They apply their techniques to a set of 940 applications and find that about one-third are overprivileged. Their key observation was that developers are trying to follow least privilege but sometimes fail due to insufficient API documentation. Another work by Felt *et al.* [14] surveys applications (free and paid) from the Android Market. Their key observation was that 93% of free apps and 82% of paid apps request permissions that they deem as "dangerous". While this does not reveal much out of context, it demonstrates that users are accustomed to granting dangerous permissions to apps without much concern. Neither of these works actually attempt to detect or categorize malicious software.

Enck et al. [10] make an effort to decompile and analyze the source of applications to detect further leaks and usage of data. Another work by Enck et al. [11] developed a system that examined risky permission combinations for determining whether the permissions declared by an application satisfy a certain global safety policy. This work manually specifies permission combinations such as WRITE_SMS and SEND_SMS, or FINE_LOCATION and IN-

App Name	Justin Bieber SMS-G
Description	View photos when you receive a message! These pictures are selected using the keyword "Justin Bieber", so they change whenever you receive a message. You will find the photo best for you!
Permissions	17 in total, some are listed below
	ACCESS OTHER GOOGLE SERVICES: Allows apps to sign in to unspecified Google services using the account(s) stored on this Android device. VIEW CONFIGURED ACCOUNTS: Allows apps to see the usernames (email addresses) of the Google account(s) you have configured. SEND SMS MESSAGES: Allows the app to send SMS messages. Malicious apps may cost you money by sending messages without your confirmation. READ CONTACT DATA: Allows the app to read all of the contact (address) data stored on your tablet. Malicious apps may use this to send your data to other people. INTERCEPT OUTGOING CALLS: Allows the app to process outgoing calls and change the number to be dialed. Malicious apps may monitor, redirect, or prevent outgoing calls.

Table 1: An App available on Google Play

TERNET, that could be used by malicious apps, and then performs analysis on a dataset of apps to identify potentially malicious apps within that set. Sarma et al. [24] take another approach which uses only permissions to evaluate the risk of an app by examining how rare permissions are for certain apps in specific categories.

Barrera *et al.* [6] present a methodology for the empirical analysis of permission-based security models using self-organizing maps. They apply their methodology to analyze the permission distribution of close to one thousand applications. Their key observations were (i) the INTERNET permission is the most popular and hypothesized that most developers request this to request advertisements from remote servers, (ii) Location-based permissions are usually requested in pairs i.e. access to both fine and coarse locations is requested by applications in a majority of cases by developers and (iii) there are some categories of applications such as tools and messaging category where pairs of permissions are requested.

Au *et al.* [5] survey the permission systems of several popular smartphone operating systems and taxonomize them by the amount of control they give users, the amount of information they convey to users and the level of interactivity they require from users. Further, they discuss several problems associated with extracting permissions-based information from Android applications.

Dynamic Analysis: Another research direction in Android security is to use dynamic analysis. Portokalidis [22] propose a security solution where security checks are applied on remote security servers that host exact replicas of the phones in virtual environments. In their work, the servers are not subject to the constraints faced by smartphones and hence this allows multiple detection techniques to be used simultaneously. They implemented a prototype and show the low data transfer requirements of their application.

Enck *et al.* [9] perform dynamic taint tracking of data in Android, and reveal to a user when an application may be trying to send sensitive data off the phone. This can handle privacy violations since it can determine when a privacy violation is most likely occurring while allowing benign access to that same data. However, there is a whole class of malicious apps that this will not defend against, namely security and monetary focused malware which send out spam or create premium SMS messages without accessing private information.

Security & Access Control: Research in this direction is geared towards furthering usable security associated with mobile phones by improving the fundamental security and access control models currently in use. This type of research entails introducing developer-centric tools [30] that enforce principle of least privilege, extending permission models and defining user-defined runtime constraints [20, 21] to limit application access and detecting applications with a malicious intent [9, 23].

Nauman et al. [20] present a policy enforcement framework for Android that allows a user to selectively grant permissions to applications as well as impose constraints on the usage of resources. They design an extended package installer that allows the user to set constraints dynamically at runtime. Ongtang [21] present an infrastructure that governs install-time permission assignment and their run-time use as dictated by application provider policy. Their system provides necessary utility for applications to assert and control the security decisions on the platform. Vidas [30] presents a tool that aids developers in specifying a minimum set of permissions required for a given mobile application. Their tool analyzes application source code and automatically infers the minimal set of permissions required to run the application.

Machine Learning in Security: Naive Bayes has been extensively used both in the context of spam detection [25, 18, 16, 28] and anomaly detection [26, 4] in network traffic flows. In the context of Android, however, there has been limited work. Shabtai *et al.* [27] presents a behavioral-based detection framework for Android that realizes a host-based intrusion detection system that monitors events originating from the device and classifies them as normal or abnormal. Our work differs in that we use machine learning for the purpose of risk communication.

8. CONCLUSIONS

We have discussed the importance of effectively communicating the risk of an application to users, and propose several methods to rate this risk. We test these methods on large real-world datasets to understand each method's ability to assign risk to applications. One particular valuable method is the PNB model which has several advantages. It is monotonic, and can provide feedback as to why risk is high for a specific app and how a developer could reduce that risk. It performs well in identifying most current malware apps as high risk, close to the sophisticated HMNB model. And it can differentiate between critical permissions and less-critical ones, making it more difficult to evade when compared with the BNB model.

9. ACKNOWLEDGMENTS

We would like to thank Xuxian Jiang and Yajin Zhou who provided us with their collection of Android malware samples, and for checking the app mentioned in Section 6. Work by C. Gates, B. Sarma, N. Li were supported by the Air Force Office of Scientific Research MURI Grant FA9550-08-1-0265, and by the National Science Foundation under Grant No. 0905442. H. Peng and Y. Qi were supported by NSF IIS-0916443, NSF CAREER award IIS-1054903, and the Center for Science of Information (CSoI), an NSF Science and Technology Center, under grant agreement CCF-0939370. Work by R. Potharaju and C. Nita-Rotaru were supported by NSF TC 0915655-CNS.

10. REFERENCES

[1] Andromo. http://andromo.com.

[2] Appsgeyser. http://appsgeyser.com.

[3] Google Bouncer. http://goo.gl/QnC6G.

[4] N. Amor, S. Benferhat, and Z. Elouedi. Naive bayes vs decision trees in intrusion detection systems. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 420–424. ACM, 2004.

[5] K. Au, Y. Zhou, Z. Huang, P. Gill, and D. Lie. Short paper: a look at smartphone permission models. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 63–68. ACM, 2011.

[6] D. Barrera, H. Kayacik, P. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 73–84. ACM, 2010.

[7] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 2007.

[8] D. Blei, A. Ng, and M. Jordan. Latent dirichlet allocation. *J. Mach. Learning Research*, 3, 2003.

[9] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–6. USENIX Association, 2010.

[10] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of Android application security. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association.

[11] 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*, CCS '09, pages 235–245, New York, NY, USA, 2009. ACM.

[12] B. Fathi. Engineering windows 7 : User account control, October 2008. MSDN blog on User Account Control.

[13] A. 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*, pages 627–638. ACM, 2011.

[14] A. Felt, K. Greenwood, and D. Wagner. The effectiveness of application permissions. In *Proc. of the USENIX Conference on Web Application Development*, 2011.

[15] A. P. Felt, K. Greenwood, and D. Wagner. The effectiveness of install-time permission systems for third-party applications. Technical Report UCB/EECS-2010-143, EECS Department, University of California, Berkeley, Dec 2010.

[16] J. Goodman and W. Yih. Online discriminative spam filter training. In *Proceedings of the Third Conference on Email and Anti-Spam (CEAS)*, 2006.

[17] W. A. Magat, W. K. Viscusi, and J. Huber. Consumer processing of hazard warning information. *Journal of Risk and Uncertainty*, 1(2):201–32, June 1988.

[18] V. Metsis, I. Androutsopoulos, and G. Paliouras. Spam filtering with naive bayes-which naive bayes. In *Third conference on email and anti-spam (CEAS)*, volume 17, pages 28–69, 2006.

[19] S. Motiee, K. Hawkey, and K. Beznosov. Do windows users follow the principle of least privilege?: investigating user account control practices. In *Proceedings of the Sixth Symposium on Usable Privacy and Security*. ACM, 2010.

[20] M. Nauman, S. Khan, and X. Zhang. Apex: Extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 328–332. ACM, 2010.

[21] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in android. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 340–349. Ieee, 2009.

[22] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid android: versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 347–356. ACM, 2010.

[23] R. Potharaju, A. Newell, C. Nita-Rotaru, and X. Zhang. Plagiarizing smartphone applications: Attack strategies and defense. In *Engineering Secure Software and Systems*. Springer, 2012.

[24] B. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Android permissions: A perspective combining risks and benefits. In *SACMAT '12: Proceedings of the seventeenth ACM symposium on Access control models and technologies*. ACM, 2012.

[25] K. Schneider. A comparison of event models for naive bayes anti-spam e-mail filtering. In *Proceedings of the tenth conference on European chapter of the Association for Computational Linguistics-Volume 1*, pages 307–314. Association for Computational Linguistics, 2003.

[26] A. Sebyala, T. Olukemi, and L. Sacks. Active platform security through intrusion detection using naive bayesian network for anomaly detection. In *London Communications Symposium*. Citeseer, 2002.

[27] A. Shabtai and Y. Elovici. Applying behavioral detection on android-based devices. *Mobile Wireless Middleware, Operating Systems, and Applications*, pages 235–249, 2010.

[28] Y. Song, A. KoÅĆcz, and C. L. Giles. Better naive bayes classiiňĄcation for high-precision spam detection. In *Software Practice and Experience*, 2009.

[29] D. W. Stewart and I. M. Martin. Intended and unintended consequences of warning messages: A review and synthesis of empirical research. *Journal of Public Policy Marketing*, 13(1):1–19, 1994.

[30] T. Vidas, N. Christin, and L. Cranor. Curbing android permission creep. In *Proceedings of the Web*, volume 2, 2011.

[31] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, 2012.

APPENDIX

The posterior distribution of the hidden variables is

$$p(\psi, z|x, \alpha, \theta) = \frac{p(\psi, z, x|\alpha, \theta)}{p(x|\alpha, \theta)} \tag{5}$$

The computation of the *exact* posterior distribution is, however, intractable. Thus, we approximate the posterior distribution by

$$q(\psi, z|\beta, r) = \prod_{c=1}^{C} q(\psi_c|\beta_c) \prod_{n=1}^{N_c} q(z_{c,n}|r_{c,n}) \tag{6}$$

To obtain an accurate approximation, we use a variational Bayes approach. Specifically, we minimize the KL divergence of p and q via the following iterative variational updates.

Update r:

$$\rho_{c,n,k} = exp\{\digamma(\beta_{c,k}) - \digamma(\sum_{k=1}^{K} \beta_{c,k})\} \prod_{m=0}^{M} \theta_{k,l}^{x_{c,n,l}} (1-\theta_{k,l})^{1-x_{c,n,l}} \tag{7}$$

$$r_{c,n,k} = \frac{\rho_{c,n,k}}{\sum_{k=1}^{K} \rho_{c,n,k}} \tag{8}$$

Update β:

$$\beta_{c,k} = \alpha_k + \sum_{n=1}^{N_c} r_{c,n,k} \tag{9}$$

Update θ:

$$\theta_{k,m} = \frac{a_{0k,l} + \sum_{c=1}^{C} \sum_{n=1}^{N_c} r_{c,n,k} x_{c,n,m}}{a_{0k,l} + b_{0k,l} + \sum_{c=1}^{C} \sum_{n=1}^{N_c} r_{c,n,k}} \tag{10}$$

Update α via Newton's method:

$$g_k = C[\digamma(\sum_{k=1}^{K} \alpha_k) - \digamma(\alpha_k)] + \sum_{c=1}^{C} [\digamma(\sum_{k=1}^{K} \beta_{c,k}) - \digamma(\beta_{c,k})] \tag{11}$$

$$q_k = -C\digamma'(\alpha_k) \tag{12}$$

$$z = C\digamma'(\sum_{k=1}^{K} \alpha_k) \tag{13}$$

$$b = \frac{\sum_{k=1}^{K} g_k/q_k}{1/z + \sum_{k=1}^{K} 1/q_k} \tag{14}$$

$$\alpha_k^{new} = \alpha_k^{old} - \frac{g_k - b}{q_k} \tag{15}$$

The $\digamma(.)$ denotes the digamma function.