8-2016

A study of security issues of mobile apps in the android platform using machine learning approaches

Lei Cen
Purdue University

Graduate School Form
30 Updated 12/26/2015

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Lei Cen

Entitled

A Study of Security Issues of Mobile Apps in the Android Platform Using Machine Learning Approaches

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Luo Si
Chair

Ninghui Li
Co-chair

Elisa Bertino

David Gleich

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy of Integrity in Research" and the use of copyright material.

Approved by Major Professor(s): Luo Si

Approved by: Sunil Prabhakar 6/24/2016

Head of the Departmental Graduate Program Date

A STUDY OF SECURITY ISSUES OF MOBILE APPS IN THE ANDROID PLATFORM USING MACHINE LEARNING APPROACHES

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Lei Cen

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2016

Purdue University

West Lafayette, Indiana

ACKNOWLEDGMENTS

First and foremost I would like to thank my advisor Prof. Luo Si. It has been an honor to be Prof. Si's Ph.D. student and he has taught me so much during this five years journey. Prof. Si generously founded most of my researches and guided me step by step into so many interesting topics. I appreciate all his time and patient with me and his contributions made my Ph.D. experience much more productive and much less confusing. The professional spirit he showed to me stimulated and encouraged me in my Ph.D. pursuit during my hardest times.

Prof. Ninghui Li provided enormous help and support as my co-advisor in the Android security project, I realy appreciate it as I was new to security field. I would also like to thank Prof. Elisa Bertino and Prof. David Gleich for joining my defence committee and providing valuable comments to help me improve on this dissertation.

Prof. Si provided an excellent lab environment for research and I am grateful to all my lab mates. The members of our IR lab have been a source of solid friendship and reliable source of ideas and collaborations in research. It has been a joyful few year spent with you all. Thank you Dr. Dan Zhang, Dr. Yi Fang, Dr. Dzung Hong, Dr. Suleyman Cetintas, Dr. Qifan Zhang, Ning Zhang, Zhiwei Zhang, Tao Wu, Kexin Pei, Chang Li and Dr. Jingang Wang.

Lastly, I would like to thank my parents for encouraging me for my pursuit and being supportive for the whole time. It would not be possible for me if I was not raised with your love and encouraging.

Lei Cen

July 1st, 2016

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

ABSTRACT

Cen, Lei PhD, Purdue University, August 2016. A Study of Security Issues of Mobile Apps in the Android Platform Using Machine Learning Approaches . Major Professors: Luo Si and Ninghui Li.

Mobile app poses both traditional and new potential threats to system security and user privacy. There are malicious apps that may do harm to the system, and there are mis-behaviors of apps, which are reasonable and legal when not abused, yet may lead to real threats otherwise. Moreover, due to the nature of mobile apps, a running app in mobile devices may be only part of the software, and the server side behavior is usually not covered by analysis. Therefore, direct analysis on the app itself may be incomplete and additional sources of information are needed.

In this dissertation, we discuss how we can apply machine learning techniques in multiple tasks for security issues in regard of mobile apps in the Android platform. These include malicious apps detection and security risk estimation of apps. Both direct sources of information from the developer of apps and indirect sources of information from user comments are utilized in these tasks. We also propose comparison of these different sources in the task of security risk estimation to point out the necessity of usage of indirect sources in mobile app security tasks.

1 INTRODUCTION

There has been a steady rise of smart mobile devices for both personal and business use. These devices run applications (apps) that have unprecedented access to private information, including contacts, emails, geo-location data, personal and business files, and much more. There is also explicit monetary risk associated with these devices since phone calls, messages, and data usage can cost money. More directly, these devices often have access to users' bank accounts through an application or as a means to authenticate to a bank, and in the future it seems likely that phones may act as a digital wallet, directly accessing the bank accounts as part of the functionality. While the existence of this information and access creates much of the value found in a smart mobile device, it also makes these devices attractive targets for malicious entities. Furthermore, the threats casted by the mobile App security problem present new properties. On one hand, comparing with traditional software markets, markets like Google Play and Apple Store have lower entry threshold for developers and faster financial payback, hence greatly encouraging more and more developers to invest in this thriving business. One result out of this is the huge amount of mobile apps with great diversity. Therefore controlling the quality of apps, especially the security risk of them across the whole markets, becomes an important issue to all that involved. On the other hand, public concerns about privacy issues with on-line activity and mobile phones are also elevating, demanding a mobile environment with more respect to users' privacy.

1.1 Security Issues of Mobile apps and Machine Learning

We may investigate the problem of mobile App security from two perspectives, in regarding of its properties compared to traditional security problems.

From the Security issue category perspective: One perspective is that the mobile App security includes both issues which are more common to traditional software security, like

malicious App classification and permission control etc.; and those which are less common like advertise abusing, unresolved In-App-Purchase (IAP) etc. While the former are clear security threats, the later sometimes could only be seen as misbehavior that may lead to real problems like phishing, fraud, privacy issues etc.

Real Threats Malware (malicious App) detection is an typical task in mobile security. Many works have been done for this task, including code analysis (static and dynamic) permission analysis etc. Please refer to Section 2.2.2 for a more detailed introduction.

Mis-behaviors Some behaviors are commonly discussed by users due to their potential of leading to real threats. Advertising abusing in mobile phone may lead to phishing attack (i.e. from pop-ups), unsolved IAP may be signs of fraud and permission over requesting may lead to privacy information stealing. These behaviors, though could be acceptable in many occasions, do cast potential threats and are worried and complained by uses all the time. Therefore, it would be necessary to estimate the security risk of mobile apps in regard of these misbehaviors.

From the source of analysis perspective: As software running in mobile phone, which have much limited computation power and display capability, mobile apps usually consist both the part that runs in mobile phones and the part supporting the functionality via Internet. Therefore the analysis of security issues of a mobile App may consider both part of it. However, due to the complication of communication between an App and its service, it would be hard to analyze the outside part directly from the mobile devices. For example, we may be able to detect an transaction of IAP in a game App, but it would be hard to track whether the transaction succeed or not. If not, whether it is just a rare bug in the server side or it is actually a case of fraud. Therefore when we gather source of information for the analysis, it would be reasonable to include source that reflect the experience of App using indirectly.

Direct Source We define the direct source of information of a mobile App as those information released by the developer of the the App. These include the .apk App file, the permission requested and App description etc.

Indirect Source We define the source of information of a mobile App as those information that is not from the developer of the App, but from the user of the App. These include mostly the comments made my the users of the App.

Machine learning technique have been introduced into mobile security by many works [1–3], and it is very adaptive to different kind of task including classification for malware detection and ranking / regression for risk estimation. With probabilistic machine leaning model, the result of security analysis may adapt to more flexible interpretation and easier for further analysis and integration. In our work, we mainly adopt probability machine learning technique as our tool for the analysis.

1.1.1 Direct Analysis

Many works have been done to analysis the security issues of mobile apps directly from the information released from the developer.

Code Analysis Code analysis tries analysis the security issues through analyzing the code of the App. These includes static analysis that extract features from the executables of mobile App, and dynamic analysis that utilize features captured while the App is running.

Permission Analysis Permission analysis analyze mobile App by evaluating the permission requested and / or used of the App.

Please refer to Section 2.2.2 for a more detailed introduction for related works. Following this line of work, we propose to conduct a experiment based work that utilized discriminated probabilistic model for detecting malicious mobile apps based on decompiled .apk package and the permissions requested. Experiments demonstrate that our approach may detect malicious mobile App in a over 95% manner in the measure of F1 value.

1.1.2 Indirect Analysis

Indirect source of mobile apps comes mostly from user comments submitted to online App store (e.g. Google Play for Android apps). There have not been much indirect analysis of mobile apps for the purpose of security concern. However, user comment analysis have been utilized in many previous works (please refer to Section 3.2.2 for more detail).

By considering the user comments as an indirect source of information, we propose a novel model for estimating the security risk of mobile apps. There are two steps of these line of work which we separated into two tasks: how to extract relevant security topics from user comments and how to estimate the security risk of apps from the security topics.

1.1.3 Comparison and Combination

Both direct and indirect source are valuable in analyzing security issues for mobile apps, but the different perspectives may rise some interesting question when we compare them. For example, the permission analysis task in many works[] compare the difference of the requested permissions to the ones actually used in the code of the App, and predict security risk by the difference of the two, but from the users' point of view, the requested permissions are always compared to the role and functionality of the App that the user comprehended. We propose to compare the two perspective by comparing code analysis in multiple security related tasked on mobile apps to user comment analysis on those similar tasks, and identify the strength and weakness of both source.

Moreover, combining both source of direct and indirect provide a promising approach in improving the performance of risk estimation of mobile apps. We propose a multi-view learning approach to incorporate both source in a mobile App security risk estimation task, and expect to show better performance from the combining of source than each individual ones.

1.2 Research Problems

In this dissertation, my main focus is on how to solve the following research problems:

- How can we apply machine learning technique in traditional security problem, like code analysis in malware detection, for Android apps?
- How can we utilize information from non-traditional perspective like the user comments in security analysis for Android apps?
- How to compare the usage of code analysis and user comment analysis in mobile App security.
- Can we combine information from both perspective and improve performance in estimation mobile App security risk?

The rest of the dissertation will be organized as follows: Chapter 2 presents a experiment based analysis on how to utilize discriminative probabilistic model on decompiled code of mobile App for malware detection. Chapter 3 presents a novel method for extracting security related topics from user comments of mobile apps. Furthermore, Chapter 4 proposes a joint algorithm that estimates both the credibility of users who give comments and the security risk of the apps. Moreover, Chapter 5 described our work that compare our user comment based analysis to traditional code analysis based work with case studies. And finally Chapter 6 will summarize the proposal.

2 A PROBABILISTIC DISCRIMINATIVE MODEL FOR ANDROID MALWARE DETECTION WITH DECOMPILED SOURCE CODE

2.1 Motivation

In this chapter, we demonstrate how we can apply machine learning technique in detecting malicious mobile apps. The task fall into the real threats category of security issues and the source of information is the decompile code of apps. Without complicated code analysis, this work treat decompiled code of apps as text document and extract function name from it.

2.2 Introduction and Related Works

2.2.1 Introduction

There has been a steady rise of smart mobile devices for both personal and business use. These devices run applications that have unprecedented access to private information, including contacts, emails, geo-location data, personal and business files, and much more. There is also explicit monetary risk associated with these devices since phone calls, messages, and data usage can cost money. More directly, these devices often have access to users' bank accounts through an application or as a means to authenticate to a bank, and in the future it seems likely that phones may act as a digital wallet, directly accessing the bank accounts as part of the functionality. While the existence of this information and access creates much of the value found in a smart mobile device, it also makes these devices attractive targets for malicious entities.

The paradigm for program distribution on these mobile devices also differs from that of the traditional PCs. Many developers are releasing applications to one or a few central application markets. While there are third party application stores, currently all popular

mobile device platforms have central application stores as the primary mechanism of application distribution. Android has Google Play as the primary store, Kindle uses the Amazon Appstore for Android, iOS has iTunes App Store, Windows RT has the Windows Store, and BlackBerry has AppWorld. This new paradigm presents both challenges and opportunities for malware defense. Instead of most programs coming from a relatively small number of reputable vendors, which enables protection based on whitelisting and signed software distributions, in mobile devices there are many more developers, many of which have insufficient history to establish reputation. On the other hand, centralized markets provide opportunities for techniques to analyze applications by extracting some set of measurable features, and identifying potentially malicious applications in the set.

In [1–3], researchers have developed several approaches that use the permissions requested by an Android application to identify whether the application is potentially malicious. In [2], requesting a certain permission or a certain combination of two or three permissions triggers a warning that the application is risky. In [3], requesting a critical permission that is rarely requested is viewed as a signal that the application is risky. In [1], four probabilistic generative models are used to to identify potentially malicious applications including Basic Naive Bayes (BNB), Naive Bayes with informative Priors (PNB), Mixture of Naive Bayes (MNB), and Hierarchical Mixture of Naive Bayes (HMNB). Experimental results show that these models significantly outperform prior approaches in [2, 3] using the Area Under Curve (AUC) for the Receiver Operating Characteristic (ROC) curve as evaluation metric.

However, permissions only provide a high-level and inaccurate view of the behavior of an application. An application may request a permission without actually using the permission [4]. Furthermore, a permission often controls multiple actions. For example, the READ_PHONE_STATE permission gives access to the devices IMEI via getDeviceId() which can be misused, the current caller is available via getCallerInfo(...) which has privacy implications, but this permission also grants access to more commonly used functions and intents such as the "android.intent.action.PHONE_STATE" intent to detect changes in the network connection type and similar changes to phone state. On the other hand, we

observe that applications are generally distributed in a form that can be decompiled into source code which enables more detailed analysis of the applications. In particular, mobile platforms like the Android system provide rich and well-defined APIs with useful semantic values for accessing the underlying rich types of data. If an application needs to access a user's contact information, it generally needs to achieve this goal via well-defined API calls, which can be captured by the decompiled source code.

Therefore, we expect that using decompiled source code of Android applications can provide more detailed information than the list of permissions that the applications request. We thus hypothesize that properly designed probabilistic models with decompiled application source code, either in place of or in addition to, permission data, are able to provide highly accurate results in identifying potentially malicious applications.

Generative probabilistic models in [1] assume that some parameterized random process generates the application data (i.e., permissions) and learn model parameters which optimize the fit of the model to the applications used in training. Then one can compute the probability of each application being generated by the models, and identify those with low probabilities as potentially malicious applications. The strength of generative models is that they work with unlabeled data, in other words without information on malicious applications. On the other hand, discriminative models have been shown to effectively utilize labeled training data in many applications such as text categorization [5] and image classification [6]. Discriminative models maximize objective functions that reflect classification accuracy with respect to labeled training data. As many malicious Android applications have already been identified in previous work [7], the labeled training information may enable discriminative learning to achieve more accurate results for Android malware detection.

We propose a probabilistic discriminative model for Android malware detection as a binary classification problem by using decompiled source code to generate features. In particular, a Regularized Logistic Regression (RLR) model is designed to generate probabilistic outputs that enable users to better interpret the probabilistic results of Android

malware detection, which may be more desired than alternative discriminative learning models like support vector machine with non-probabilistic outputs.

One approach is to view decompiled source code as texts, and use techniques for document classification for Android malware detection. However, source code is different from documents, and this difference presents some interesting new questions. An interesting question is what features to extract from the source code. For example, one can choose to use a word-count representation commonly used in document classification or a binary representation. In addition, one can choose features at different granularity such as the package, class, or function level representation. When classes or functions are used, the number of features may be very large. Feature selection techniques may prove beneficial for accuracy and efficiency. Another interesting question is whether combining source code with permissions can present better results than using source code and permissions separately.

There are many options for an evaluation metric. AUC of ROC is a popular choice used by many researchers [1, 8]. However, AUC of ROC is reported to be "overly optimistic" of the performance in case of a highly imbalanced dataset [9]. Therefore other metrics like F1 value may be considered as a more reliable metric, since in reality, malicious applications are a very small portion of all the applications, and the precision metric used by F1 can better reflect the performance under imbalanced data.

In summary, the research in this paper makes significant contributions for Android malware detection as follows:

- We propose a probabilistic discriminative model based on regularized logistic regression for Android malware detection with decompiled source code, which can generate much more accurate detection results than previous research with application permissions or with source code.

- We discuss and present experiments to show that the F1 value is a better metric to understand and compare the performance of malicious application detection.

- We provide thorough empirical studies and discussion for exploring desired representation of decompiled source code such as feature extraction, representation granularity and feature selection and modeling strategies for effective and efficient Android malware detection.

- We further extend the proposed discriminative probabilistic model for utilizing both decompiled source code and permissions, which generates even better results for Android malware detection. As far as we know, this is the first research work that combines analysis of both source code and permissions for the task.

The rest of the paper is organized as follows. Section 2.2.2 reviews related research work in Android malware detection. Section 2.3 proposes our new research for the task. Section 2.4 presents an extensive set of experiments to demonstrate the advantage of proposed research. Section 2.6 concludes the contribution of this paper.

2.2.2 Related Works

Malware Detection - Static: One issue for malware detection is to determine the right features to extract from executables. One research thrust has focused on binary analysis, looking at the bytes of the binary on disk or in memory. Kolter *et al.* [10] look at *n-grams* from the binary for classification, where an *n-gram* is *n* consecutive bytes. They use these features in conjunction with several machine learning techniques including naive Bayes, decision trees, support vector machines (SVM) and boosting, and obtain reasonable performance. BitShred [11] extends the previous idea while focusing on modern malware and scalability, extracting features in several ways to handle encrypted code and using feature hashing to compress the feature space effectively while still maintaining the performance of standard machine learning techniques.

In the context of mobile, some work on static analysis has also been performed, typically focusing on all functions that are used by an application. One common problem in third party application stores is piracy and malware. It is possible to buy an application, repackage the application unchanged or with malicious code added, and then submit it to

a third party application store for others to install. Desnos [12] uses compression and distance on source code to find applications that have high overlap in order to detect piracy and malware between the official market and third party markets. Schmidt *et al.* [8, 13] use static function call analysis to detect mobile malware. They apply their technique to both Symbian and Android to extract function calls from binaries and then perform basic machine learning techniques such as nearest neighbor and clustering. They focus on traditional unix/linux elf binaries; however, most known mobile malware comes in the form of an application, i.e. an $.apk$ file, and not as an elf. Their focus on elf binaries is likely due to the timing of the work, which precedes most known malware and comes shortly after Android's initial release.

Enck *et al.* [14] perform static data flow analysis on application files by decompiling and analyzing the source code to detect data leaks. They find many applications that leak information off the phone, but do not find any evidence of malicious applications in their data. RiskRanker [7] and DroidRanger [15] focus on the task of finding malicious applications in various Android markets. They use a few detection techniques to identify possibly risky behavior in an application, and then perform further analysis to identify true malicious behavior and false positives. The detection techniques look for actions expected from malware: specific code signatures for known attacks, behavioral indicators such as sending an SMS automatically and not associated with a user clicking a button, encrypted native code, and dynamic code loading. These techniques are all basically signature based, looking for pre-specified patterns or behavior. By contrast, our approach is data driven, using labeled data to determine which features are important for classification.

Aafer *et al.* [16] perform static data flow analysis to extract feature from the bytecode of applications. Both APIs and the parameters of the APIs in the bytecode are extracted and APIs are filtered by the relative usage frequency between benign and malicious applications. K-nearest-neighbor method is proposed to work with these features and compared to permission feature. To distinguish our work with theirs, we propose a probabilistic model that could output meaningful probabilistic results and learns the weight of APIs and permissions automatically instead of picking the highest relatively frequent ones. We also

present results that combines both API feature and permission feature, which performs the best in our experiments.

Malware Detection - Dynamic: Another major approach focuses on behavioral analysis of malware. In the desktop setting, Christodorescu *et al.* [17] allow code to execute and monitor system calls in order to identify behavioral traces of a binary. They collect traces for both benign and malicious applications and determine what traces characterize malicious binaries in order to identify unknown malware that performs similar functionality. This work uses dependence graphs to construct minimal representations for the malicious behavior, and then looks for these same behaviors in other binaries. Other work has used the same general idea of behavioral analysis, but extended the technique of matching malicious code using several machine learning techniques such as support vector machines (SVM) [18] and clustering [19].

For dynamic detection of mobile malware, Shabtai *et al.* [20] present a behavioral-based detection framework for Android that monitors certain observable events originating from applications and classifies them via lightweight machine learning techniques. The focus here is relatively efficient behavioral detection since it is performed directly on the device where power may be limited. Portokalidis [21] propose a security solution where security checks are applied on remote security servers that host 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. CrowDroid [22] provides a framework to dynamically analyze applications behavior to detect malware on the Android platform. They collect the system traces from many real users and send them to a central server for analysis to detect behavioral differences between applications that should generally have the same behavior. Their goal is primarily to detect malware that has been repackaged and distributed on third party application stores.

While dynamic approaches can offer the most information regarding application behavior, it is difficult to explore all possible behaviors of an application ahead of time, and it can be resource intensive to collect directly on a device. Additionally it is difficult to

capture fine grained behavior profiles in Android without rooting a device and installing a system designed to collect this information. On the other hand, since the application store is centralized, an approach which can more directly utilize data from all applications is preferred. Due to the complexity of permission and system level functions, it seems likely that performing static analysis of the API calls can provide a robust feature space, which is the approach we take in this work.

Android Permissions Analysis: PScout [23] performs static code analysis on the Android source to extract function to permission mappings. They find that the Android permission system has little redundancy and it remains relatively stable as the Android OS evolves. They also show how many functions require specific permissions, demonstrating the complexity of the system and that a permission may have many reasons for being requested. In [24] Felt *et al.* conduct a more general survey of applications (free and paid) from the Android Market. One key observation was that 93% of free applications and 82% of paid applications request permissions that they deem as "dangerous". This demonstrates that users are accustomed to installing applications with potentially intrusive or dangerous permissions requests. This also highlights a need to identify possibly malicious applications and communicate that risk to users. Felt *et al.* [4] also use static analysis to check if an Android application requests permissions which it is never actually used. They evaluate 940 applications and find that about one-third are over-privileged, showing that it is not always the case that the permissions requested reflect the true underlying functionality of an application. This difference between permission requests and permission use is another reason that we believe source code analysis will benefit the detection task.

Evaluation Metrics: In [25], the relationship between ROC and Precision Recall Curve (PRC) is analyzed, connections and differences between the two curves are illustrated. Furthermore, [9] systematically studied the learning process in imbalanced data, and pointed out AUC of ROC is too optimistic with imbalanced data, and PRC can better reflect the performance under the situation of imbalanced data. Therefore F1 score that computed on a proper point of PRC may be a better evaluation metric and is adopted in this paper.

2.3 Algorithm

Our goal is to develop effective techniques that can classify whether an application is likely to be malicious, given its $.apk$ file, which includes both the code of the application and the list of permissions that the application requests.

In this section, we first describe how to extract features from the $.apk$ files, and then describe our proposed Probabilistic Discriminative Model.

2.3.1 Source Code Feature Extraction

Android applications are packed as *.apk* files. To extract features from them, we first decompile the *.apk* files into Java source code files, and then extract features from the source code.

Decompile

We first unpack *.apk* files to get *.dex* files, and then use the *dex2jar*[1] tool to convert the *.dex* file to *.jar* file. We then use the *jad*[2] tool to decompile the *.class* files in a *.jar* file to Java source files. For each application, it takes approximately 2-3 minutes to decompile it into Java files. The decompiled Java files misses some information about the names of classes and variables (e.g., in "private static class a", the name "a" is not meaningful). However, this does not affect our feature extraction because the calls to the Android API are maintained and these are the functions that we care about as they reveal the actual behavior of an application.

Feature Granularities

In the Android platform, most critical operations are carried out by making API calls; thus it is natural to extract API usage information from the source code as features. One

[1] https://code.google.com/p/dex2jar/
[2] http://www.varaneckas.com/jad/

can view the source code as documents, and apply document classification techniques to the problem. In document classification, term frequency (TF) is typically used as features. There are three natural levels of granularity: package, class, and functions in Android for API feature representation. For each level of granularity, we obtain from the source code the TF. For the package-level and class-level features, we scan the import statements to count how many times a package/class is imported. For the function-level, we need to scan all the Java source files to obtain the number of occurrences of each function in one application.

We obtain the lists of package, class and function names from Google Android API document Version 4.2. See Table 2.1 to see the number of features at different granularities. The rules we use to extract the features are:

1. For the package-level feature, since in the import statements, the substring before the last dot is the name of a package, e.g., "import java.io.File", "java.io" is a package. We count these substrings to obtain the package-level features.

2. For the class-level feature, we count the whole string after "import" to obtain the class-level features. Notice that, the number of class here is the sum of the number of packages and the number of classes in Google Android API document. The reason is that some applications use the import statements with "*", e.g., "import java.util.*;". When we extract the class-level features, in order to deal with this case, we discard the "*" and just consider it as "import java.util". Because "java.util" is a package, we add the packages into the class-level features. In fact, the class-level features used in this paper contain both the packages and classes.

3. For the function-level feature, we scan the whole Java files to obtain the count of each function. However, some function names are shared by multiple classes, e.g., "toString()", the method which is used to extract the function-level feature can only get an estimated count of the occurrence of each function, not an accurate count. Because Java is an Object Oriented Programming (OOP) language, functions may be overloaded or inherited by subclasses, the class that one function actually belongs to

can only be determined during the run time. In other words, we can obtain the accurate count of the occurrence of each function only by dynamic analysis or advanced software engineering techniques. Therefore, we just use the estimated count in this paper. We discard all constructors listed in Google Android API document.

In general, higher-dimension and finer-grained features provide more information and thus may result in higher accuracy. On the other hand, higher-dimension representation may lead to more sparsity and potentially causes overfitting, and thus requires techniques such as feature selection and/or regularization to avoid overfitting.

Table 2.1.: Feature dimension of three granularities

	Dimension
package-level	179
class-level	3497
function-level	22136

Source Code Feature Representation

Besides the granularity of the source code features, how the feature values are calculated is also an important issue. We extract the Term Frequency (TF) of the features, indicating how many time a feature term (package, class or function) appears in the decompiled source files of an $.apk$ file. Some other choices are listed below:

- **Binary**: Truncate the TF to either 0 or 1. That is, if TF≥ 1, set it to 1. This feature simply indicates whether a term appears or not. While this is uncommon in document classification, the nature of programming makes this an interesting representation. This is due to the fact that as long as an API function is included once inside one wrapper function, it can be used elsewhere without directly referring to the function name, but instead merely by calling the wrapper.

- $\log(\mathbf{TF}+1)$: Transform the TF to log scale. It is a common usage for TF in document classification to remove the influence of a large TF value.

- **Inverse Document Frequency (IDF)**:
  IDF $= \frac{\#\text{Documents}}{\#\text{Documents have the term}+1}$. The intuition is that a term more unique for a document is more indicative for the document than other more common terms. The $+1$ is used to avoid division by zero. We apply $\log(\text{IDF})$ to the TF, Binary and $\log(\text{TF}+1)$.

Feature Selection Methods

Feature selection technique is widely used for data preprocessing. The purpose is to find out the most valuable features. The benefit of feature selection is two fold. First, it may improve efficiency. Reduced feature dimensionality results in fewer parameters in the model and less training and testing time. Second, it may improve accuracy. The abandoned features may actually be noisy for the task and may cause over-fitting, hence the performance may be improved after feature selection.

We use Information Gain (IG) and Chi-square test (CHI) for feature selection [26], which can be calculated very efficiently (i.e. , in one pass of source code files). IG measures the mutual dependence of the label and features, while CHI measures the lack of independence between them, hence both are reasonable choices for feature selection. Both feature selection methods assign scores to all the features, and the features with high scores will be selected. Let $\{c_i\}_{i=1}^{m}$ be the labels for m categories ($m = 2$ for binary classification) and t be the feature dimension to be evaluated, the score function for IG is:

$$\begin{aligned} G(t) = & - \sum_{i=1}^{m} Pr(c_i)\log(Pr(c_i)) \\ & + Pr(t)\sum_{i=1}^{m} Pr(c_i|t)\log(Pr(c_i|t)) \\ & + Pr(\bar{t})\sum_{i=1}^{m} Pr(c_i|\bar{t})\log(Pr(c_i|\bar{t})) \end{aligned}$$

And for CHI it is:

$$\begin{aligned} \chi^2(t,c) &= \frac{N \times (AD - CB)^2}{(A+C) \times (B+D) \times (A+B) \times (C+D)} \\ \chi^2_{avg}(t) &= \sum_{i=1}^{m} Pr(c_i)\chi^2(t,c_i) \end{aligned}$$

where N is the size of samples, A is the number of times that t and c co-occur, B is the number of times t occurs without c, C is the times c occurs without t, D is the times neither t nor c occurs. Then $\chi^2_{avg}(t)$ is the score function.

2.3.2 Probabilistic Discriminative Model for Classification

As describe previously, the malware detection task is treated as a feature-based classification problem. The malware applications are the "positive" samples, and the benign are the "negative" samples. The purpose of a classifier is to learn a "model" from the training samples to most effectively predict labels for the training data. Then the learned "model" can be applied to a new sample, to predict whether the new sample gets a positive or negative label.

In a classification problem, we use X to denote the input, i.e. , the features from a sample, and Y to denote the output, i.e. the class label. A probabilistic (statistical) model builds probabilistic relationship between the input features and the class label. In particular, one computes the posterior probability of the output variable Y given the input X, i.e. , $Pr(Y|X)$. If one constructs models for how the sample (features) is generated by the class as $Pr(X|Y)$, and then apply the Bayes rule to compute $Pr(Y|X)$, it is called a probabilistic generative model; if one models the $Pr(Y|X)$ directly, it is called a probabilistic discriminative model.

This section proposes a generative model, the 2-class Naive Bayes with Prior (2-PNB) and a discriminative model, the Regularized Logistic Regression (RLR) to work with source code features for the Android malware detection task. As baselines for compari-

son, we also include some discussions for two other generative models, PNB and HMNB, proposed in [1].

2 Class Naive Bayes with Prior

Naive Bayes (NB) is frequently used as a base line for classification since it performs well and is straight forward to implement and use. NB assumes the independence of the individual features, i.e.

$$Pr(X|Y) = \prod_{i}^{n} Pr(x_i|Y) = \prod_{i}^{n} \theta_{i,Y}^{x_i}(1-\theta_{i,Y})^{(1-x_i)} \tag{2.1}$$

where $X = [x_1, x_2, \cdots, x_n]$ is the feature vector, and a Bernoulli distribution with parameter $\theta_{i,Y}$ is associated with a binary feature value as the ith feature in class Y. To learn this model, given class $Y \in \{0, 1\}$, negative training sample $X^0 = \{X_1^0, X_2^0, \cdots, X_{N_0}^0\}$ and positive training sample $X^1 = \{X_1^1, X_2^1, \cdots, X_{N_1}^1\}$, the Maximum Likelihood Estimation (MLE) of the parameters $\theta_{i,Y}$ would be:

$$\hat{\theta}_{i,Y} = \frac{\sum_{X \in X^Y} x_i}{N_Y}$$

With proper prior $\theta_{i,Y} \sim Beta(\alpha_{i,Y}, \beta_{i,Y})$, the Maximum A Priori (MAP) becomes:

$$\hat{\theta}_{i,Y} = \frac{\sum_{X \in X^Y} x_i + \alpha_{i,Y}}{N_Y + \alpha_{i,Y} + \beta_{i,Y}}$$

The prior can be informative, like PNB in [1], giving different features different importance based on some expert knowledge. Or, it can be uninformative, i.e. using uniform distribution for the prior. For our work with source code features, the uninformative prior is adopted to avoid zero probability in computation.

To predict the label of a new sample X', the following probability is computed:

$$\begin{aligned} & Pr(Y=1|X') \\ \propto\ & Pr(X'|Y=1)Pr(Y=1) \\ =\ & \frac{Pr(X'|Y=1)Pr(Y=1)}{Pr(X'|Y=1)Pr(Y=1)+Pr(X'|Y=0)Pr(Y=0)} \end{aligned}$$

where $Pr(X'|Y)$ is computed as in equation (2.1), and $Pr(Y) = \frac{N_Y}{N_0+N_1}$. Then a threshold may be found in training set for $Pr(Y=1|X')$ to make the prediction.

This is a common way to use NB as a binary classifier, and will be called 2 class Naive Bayes with Prior (2-PNB) in this paper, to distinguish it from the Naive Bayes with informative Prior (PNB) from [1]. The PNB model only utilizes negative samples to train the model, and predicts a label based on the likelihood of a new sample given the model. The 2-PNB model on the other hand utilizes both negative and positive samples to train the model and predict based on posterior probability of the label given test sample, hence it is expected to have better performance in classification. 2-PNB also works as an intermediate model between PNB and LR, which will be discussed in the next section.

Logistic Regression

Logistic Regression (LR) is a popular classifier as a probabilistic discriminative model. Given a feature vector X and the class label $Y \in \{1, -1\}$, the posterior probability is used to predict the label Y of X:

$$Pr(Y|X) = \sigma(w^T X + b)$$

where $\sigma(a) = (1 + exp(-a))^{-1}$ is the sigmoid function. w and b are the model parameters that need to be estimated in learning process. Note that the posterior probability $Pr(Y|X)$ is directly modeled by the sigmoid function, indicating that LR is a probabilistic discriminative model.

Gradient descent can be used in the learning phase of LR to find the MLE of w and b. The objective function would be the likelihood of the training samples. Given training samples $\{(X_1, y_1), (X_2, y_2), \cdots, (X_N, y_N)\}$, the Negative Log Likelihood (NLL) function would be:

$$NLL(X, w, b) = \sum_{i=1}^{N} ln(1 + exp^{-y_i(w^T X_i + b)})$$

The L-BFGS quasi-Newton method [27] is applied in this work to find the best w and b that minimizes the NLL. The required gradient of NLL is computed as follows:

$$\begin{aligned} \frac{\partial NLL}{\partial w} &= -\sum_{i=1}^{N} \sigma(-y_i(w^T X_i + b)) X_i y_i \\ \frac{\partial NLL}{\partial b} &= -\sum_{i=1}^{N} \sigma(-y_i(w^T X_i + b)) y_i \end{aligned}$$

One aspect of LR is that the learned w parameter provides a hint about how important individual features are in the classifier. A high magnitude positive value in w implies that the corresponding feature is favored by the positive samples, and negative value in w has the same meaning for negative samples. This is a good property for inspecting the importance of each feature. For example, if the function level feature is applied, the learned LR model will show which function increase (decrease) the probability of an app being malicious, and to what extend. Therefore even all feature dimensions are treated equally in feature representation, their different importances are learned automatically during the model learning process.

It can be seen that LR directly models the posterior probability for classification as a discriminative model, while the NB related methods focus on input generation probability (i.e. $Pr(X|Y)$) and use Bayes rule to model posterior probability as generative models. Therefore LR models may have an advantage when modeling posterior probability and achieving better classification performance. In Section 2.4, it will be thoroughly studied and compared to other models mentioned in this section.

Model Regularization

Regularization is usually applied with LR to avoid the overfitting problem. Without it, LR and many other models may "overfit" the training data and perform very well in training but much worse on the test set. Regularization is achieved by adding a penalty term into the objective function to make the trade-off between the optimization of the original objective function (i.e., related with accuracy or loss) and the complexity of the learned model. The objective function of Regularized LR (RLR) looks like this:

$$min_{w,b}(NLL(w,b) + \lambda \cdot \text{penalty}(w,b))$$

where λ is a parameter to tune the trade-off of the loss function (NLL) and the penalty term. These parameters are tuned in the training set by cross validation. In our experiments, 5-fold cross validation is adapted. The training set is evenly divided into 5 pieces and each piece of data is rotationally selected as a pseudo testing set with the rest 4 pieces as pseudo training set. With these 5 pairs of sets, a possible range of the parameters are evaluated and the best in average will be used in testing.

Some popular forms of the term include Ridge ($|w|_2^2$, i.e. l_2 norm), Lasso ($|w|_1^2$, i.e. l_1 norm), and Similarity based norm ($w^T L w$) [28]. Ridge norm represents the "natural" distance metric in Euclidean space and the norm and its derivative are easy to compute. When using Lasso, the penalty term will try to force the optimization process to produce a more sparse result of model parameters, which means the resulting w tends to have more zero values than using Ridge. This is particularly useful when the feature dimension is high and the feature representation is sparse. The similarity based norm utilizes the Laplacian Matrix to embed the similarity between the features into the optimization process. It tries to force the resulting w to assign similar values to those features that are considered similar or related. When applied to source code features, the co-occurrence matrix is used to compute the Laplacian matrix L, as

$$A = \{a_{ij}\}, a_{ij} = \frac{1 + \sum_X x_i \& x_j}{1 + \sum_X x_i | x_j}$$

where $\&$ and $|$ are binary "and" and "or" operator, x_i are assumed binary and the $+1$ is used to avoid division by zero. Then $L = D - A$, D is a diagonal matrix with its diagonal elements $D_{ii} = \sum_j A_{i,j}$. All the three regularization strategies have their advantage, and it is hard to say beforehand which one maybe better on this task. The performance comparison of these strategies will be shown in Section 2.4.6.

2.4 Experiment

In this section, a series of experiments will be conducted to explore the desired feature representation and modeling strategies for Android malware detection. Many aspects of the classification process will be investigated in different stages. Each stage of the experiments is actually used to answer one of the following questions:

1. What granularity should be used to extract the features from applications? (Section 2.4.3)
2. How to calculate feature values for better Android malware detection? (Section 2.4.4)
3. Can feature selection help? If it does, in which way? Performance or dimension reduction? (Section 2.4.5)
4. Does the model require regularization? If it does, what kind of regularization should we choose? (Section 2.4.6)
5. How is the performance of proposed model compared to other learning methods? (Section 2.4.7 and Section 2.4.8)
6. Can the source code features be combined with the permission features to further improve the detection performance? (Section 2.4.9)

The default setting for experiments are as following:

model: Regularized Logistic Regression (RLR) with lasso (L1) norm is chosen as the default classifier.

feature: The binary features (no IDF) in granularity of Java Function level are used as features.

Many experiments are conducted to explore one component of the default setting (e.g., granularity in Section 2.4.3) by fixing other components.

2.4.1 Dataset Settings

The datasets we use are described in Table 2.4. They are collected from different sources. $C11$, $C12$, and CM are used in [1]. $C11$ contains 71331 applications collected from Google Play in February 2011 and $C12$ contains 98510 applications collected from Google Play in February 2012. $C11$ and $C12$ only contain permission and category information. Applications in them are presumably benign. CM contains 378 malicious applications, shared from the authors of [29]. We use these datasets to compare directly with results in [1].

O consists of 12,801 presumably benign *.apk* files from Google Play, which were obtained from authors of [16]. OM consists of 1260 malicious applications, shared by the authors of [29]. Since OM is collected later than CM, CM is a subset of OM. We can extract source code features for O and OM.

To ensure reliable evaluation, we remove the duplicate applications within each set and the overlap applications between related sets. The reason for this cleaning is to make sure that no developer or application can overly influence the model, which could have implications in testing. Duplicates are removed from the $C11$ and $C12$ sets by looking at developer information and permission vectors, removing all exact matches. Duplicate applications within O and OM and CM are cleaned by looking at package names which are obtained from the corresponding *manifest.xml* files and permission vectors. The original size and size after cleaning are shown in Table 2.4.

In order to make a thorough comparison, we create 5 training/testing settings by combining the datasets, as shown in Table 2.4. $S5$ is the only setting where *.apk* files are available, and is the main experimental setting in which we evaluate methods using source

code features. $S1$ to $S4$ are used to evaluate methods using only permission features. $S1$ is the setting used in [1], so that we can directly compare with the results there. $S2$, $S3$ and $S4$ vary the combinations to provide different situations for comparing three generative models and our discriminative model using permission features.

Table 2.4 also gives the malicious sample ratio (i.e., the ratio of malicious applications to all applications) in all 5 pairs of training/testing settings. Note that, in some training/testing settings (e.g., S4), the malicious sample ratios of training are different from the ratios in the testing set, which reflect the fact that training data and testing data may not always be consistent in realistic setting for malware detection.

It's worth noting that the ground truth for those benign apps are assumed benign only because they are from Google Play. The evaluation is set out to test the proposed method for detecting known malicious apps from those in Google Play, instead of finding a new strain of malware. How to apply the proposed method to find malware in Google Play would be an interesting future work.

2.4.2 Evaluation Metric

We consider two evaluation metrics, the F1 score and AUC of ROC score. Given the ground truth information and the prediction/classification results, there are four possible outcomes: true positive (TP), true negative (TN), false positive (FP), and false negative (FN). These are shown in the Table 2.2, which also gives an example of the number of each outcomes. For example, TP means an application is malicious with respect to ground truth and it is classified as malicious, and similarly for other terms.

Table 2.2.: Terms related in evaluation metrics and an example

		Actual class	
		True	False
Predicted class	Positive	TP (95)	FP (190)
	Negative	FN (5)	TN (3610)

From these we can compute Precision $= \frac{\text{TP}}{\text{TP+FP}} \approx 0.33$, and Recall $= \frac{\text{TP}}{\text{TP+FN}} = 0.95$. Recall is also known as the detection rate or the True Positive Rate (TPR), and gives the percentage of malicious applications detected. Precision is the Bayesian detection rate, and gives the percentage of detected applications actually being malicious. F1 score is the harmonic mean of precision and recall.

$$\text{F1} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2 \times \text{TP}}{2 \times \text{TP} + \text{FN} + \text{FP}} \approx 0.49$$

When a method has a threshold parameter for classifying whether an application is malicious or not, we apply the trained model to the training set, and the selected threshold is the one that maximizes the F1 value in training set.

The AUC score is the area under the ROC (Receiver Operating Characteristic) curve. ROC curve considers two variables: TPR (i.e., the Recall) and False Positive Rate (FPR). TPR stands for the probability that a classifier classifies true samples correctly. The definition of FPR is FPR $= \frac{\text{FP}}{\text{FP+TN}}$. FPR stands for the probability that a classifier classifies false samples incorrectly. For binary-class classification problem, a threshold is needed to classify the results into two classes: positive or negative. Therefore, for different threshold choices, different FPR-TPR combinations (as well as Precision-Recall combinations) can be calculated. Each combination is a point in the ROC space, linking all the points produces the ROC curve. The larger the AUC score, the better the classifier performs. AUC of ROC is computed w.r.t. a ranked list of the results.

We do not plot the ROC curves in this paper as an evaluation metric. The reason is that many of the ROC curves in our experiments are mixed together to the top-left corner (when the AUC of ROC is over 90%), requiring a zooming in to tell the difference. This is due to the disadvantage of ROC curve in classification task comparing to F1: the ROC curve is too optimistic for imbalanced data.

The physical meaning of AUC of ROC is the probability that a randomly selected positive sample has high score than a randomly selected negative sample [30]. This probability may not be so meaningful under the situation of imbalanced data, due to the Base Rate

Table 2.3.: Performance of different granularity of source code features in $S5$

Granularity	F1
Package	0.7984
Class	0.9174
Function	**0.9513**

Fallacy [31]. When the number of malicious applications is small relative to the total number of applications, it is possible to have high TPR and low FPR (and thus high AUC of ROC), yet low precision (namely, Bayersian detection rate), and thus low F1. The example in Table 2.2 has TPR$= 0.95$ and FPR$= 0.05$, but a F1 score of less than 0.5.

Accuracy is another metric used by some work [16], it is computed as $Accuracy = \frac{TP+TN}{TP+TN+FP+FN} = 0.95$ for the example in Table 2.2. It is also clearly very optimistic for imbalanced data. Actually, a trivial classifier that always predicts Negative would give an accuracy of 0.97 already.

2.4.3 Experiments for Granularity of Source Code Features

The experiments in the next subsections are conducted with $S5$, where the source code features are available. In this stage of experiments, the granularity of the source code features are explored. The candidates are Java package level, class level and function level. The numbers dimensions are 179, 3497, and 22136 for package, class and function respectively. Table 2.3 shows the performance of the three granularity levels using binary feature expression. The function level representation outperforms the others. The result indicates that finer level of the source code feature provides better performance, which is consistent with our expectation since a package or class may have various functionality that is not all informative.

2.4.4 Experiments for Source Code Feature Representation

Various types of feature representations introduced in Section 2.3.1 are tested in this stage of experiments with function level granularity. In particular, we investigate the usage of term frequency, log term frequency and binary representation. We will also investigate how inverse document frequency affects classification performance.

As shown in the Figure 2.1, the best choice for feature representation is Binary. One reason for this is that the TF representation is not an accurate reflection of how often a function is used in an application. A function can appear in source code multiple times but not be used, or it can be present only once but be invoked many times. The actual function use can only be determined at runtime or through advanced code analysis techniques due to the dynamic nature of Object Oriented language like Java. Hence using only binary features, i.e. whether a function is present or not, provides less noise and is more reliable. The same results have also been observed for feature representation with class level granularity. The reason why using IDF gives similar or mixed results may be related to the RLR model that we use. The training process of the RLR model finds a weight vector which indicates the importance of each individual feature, and thus provides similar functionality to IDF.

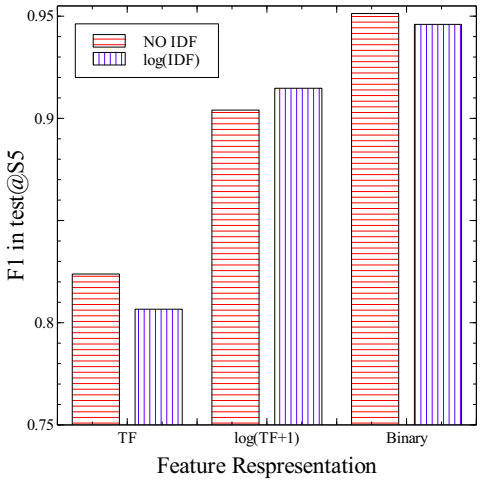


Figure 2.1.: Performance of source code feature representation in $S5$

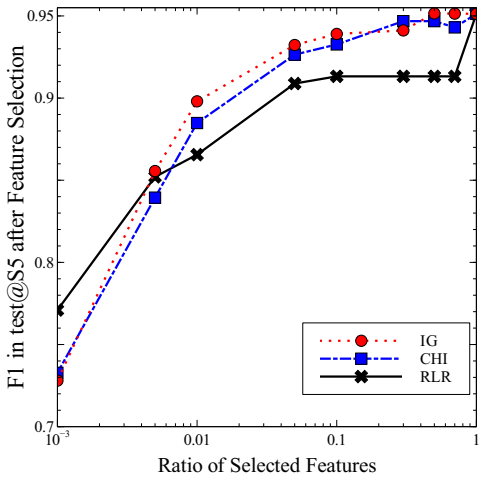


Figure 2.2.: Performance of binary function level feature representation for $s5$ using RLR for classification

2.4.5 Experiments for Feature Selection on Source Code Feature

Three feature selection methods are used in this stage to find out whether the feature dimension can be reduced or the performance can be improved. We focus on function level representation, where the original dimension of the feature space is $22,136$. IG and CHI, as introduced in Section 2.3.1, are used as two feature selection methods. Moreover, as discussed in 2.3.2, the parameters of RLR model can indicate the importance of the individual features, and thus it is also enlisted to provide feature selection result.

Figure 2.2 shows the performance of the RLR method using 0.1% to 100% of the total features selected by the three methods mentioned above. As shown in the figure, the performance of using 10% of features degrades very little compared to using all features especially for the IG and CHI selection methods. Therefore, if efficiency is an issue in real world applications, a good trade-off between effectiveness and efficiency can be achieved by the feature selection methods.

We list the top 20 functions which are selected by IG in Figure 2.3. For each function in the figure, we calculate the percentage of benign and malicious applications which use this function. This information is also shown in Figure 2.3. As shown in the figure, some critical functions, e.g. "getSubscriberId()" and "getInstalledPackages()", are

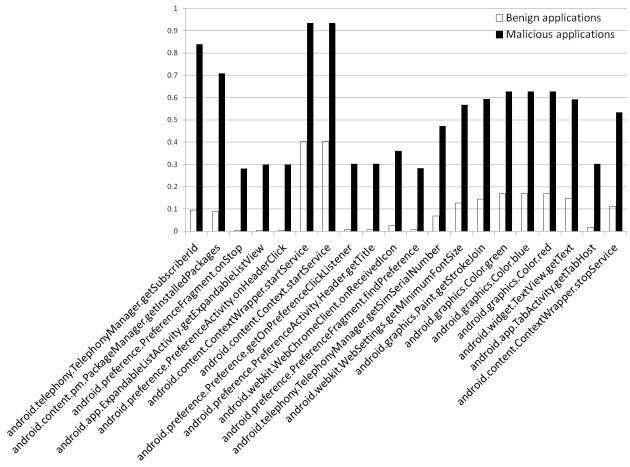


Figure 2.3.: Percentage of benign/malicious applications using the top 20 functions selected by IG.

more likely to be used by malicious applications. However, some normal functions like "android.graphics.Color.blue" are also in the list. This could be an coincident due to the dataset property, since the number of malwares is much lesser than the benign ones. Also, normal functions, if their popularities are statistically different in malware from benign ones, are still useful for detecting malware. Moreover, it is worth clarifying that the functions shown in Figure 2.3 are far from what are used in detection. In fact, as shown in Figure 2.2, we need at least about 50% (over $11,000$ functions) of the features to maintain a similar performance to the best that uses all features. Therefore, in order to fully evaluate the performance of the proposed method, the full sized feature ($22,136$ functions) are used in the rest of the experiments.

2.4.6 Experiments for Model Regularization

Different regularization methods for the proposed RLR method are explored in this stage of experiments with binary feature presentation in the function level. In particular, we compare the no regularization approach, ridge regularization, lasso regularization and similarity based regularization as introduced in Section 2.3.2. Table 2.5 shows the results with the 4 different regularization settings. First, the method without using regulariza-

tion is outperformed by all the other methods since this method suffers from over-fitting in the high dimensional space of functions. The lasso regularization gives the best performance, while the ridge regularization gives the second best performance. One possible reason why lasso works well is the feature dimensionality is high and the feature vectors are quite sparse, hence the lasso regularization, which constrains the objective function to find a sparse solution better fits in the situation. The similarity matrix used in similarity regularization is computed using the function co-occurrence matrix. Yet two function calls co-occur in the source code of an application may actually belong to two different functional modules of the application, it is hard to say if the two functions are closely related or not. Hence the co-occurrence probability may not be as reliable as expected.

2.4.7 Model Comparison with Permission Feature

The above experiments are all conducted with RLR and features from source code. Two generative models called Naive Bayes with Informative Priors(PNB) and Hierarchical Mixture of Naive Bayes(HMNB) proposed in [1] use permission features to detect malicious application in the Android platform. It would be interesting to compare RLR, which is a discriminative model, to PNB and HNMB, with permission features.

The experiments in [1] use the evaluation metric AUC of ROC. A comparison experiment is first done on the $S1$, which is built by following the instruction of [1], to compare AUC of ROC to F1 and verify our implementation of those two methods. Table 2.6 shows the performance of our implemented PNB and HMNB with both AUC and F1 for evaluation. As shown in the table, the AUC of PNB and HMNB are 0.937 and 0.948 respectively, which matches the result posted in [1]. It is worth noting that although the AUC of ROC scores look very good, the corresponding F1 scores are not very high. Moreover, since the two generative methods use only negative samples for training, there is not a good way of finding a threshold for classification solely based on their training data. Since the training dataset does not contain samples of malicious applications, we cannot use cross validation to choose the threshold. The F1 scores shown in Table 2.6 are actually computed by choos-

ing the threshold to maximize the F1 scores against the testing data; thus they are the best F1 scores that can be found in the testing set by using the trained generative models and the information of ground truth of the testing set. Therefore the F1 scores of the two generative models are somehow overestimated. We show that even these overestimated F1 scores of the generative models are still low compared with our approach.

Table 2.7 shows the performance for comparing three generative models (i.e., PNB, HMNB and 2-PNB) and our discriminative model using permission features over three training/testing settings. The prior used for PNB and 2-PNB are from [1]. The results show that, RLR outperforms the others in all cases for F1 score, and is still the best in most of the cases for AUC score of the ROC metric. Considering the variety of three different training/testing settings with different sources of the samples and different ratios of positive samples, the results suggest that RLR is better in classification performance using permission features than the generative models. 2-PNB method, as a generative method but utilizing both positive and negative samples in training, performs better than PNB and HMNB in three settings (i.e., $S2$, $S3$ and $S5$). This indicates the value of using both positive and negative samples in training. The results of HMNB is similar to PNB, with a slight advantage in $S4$. This is consistent to the conclusion in [1]. In summary, the discriminative model RLR is recommended against the generative models when permission features are used.

The two settings $S4$ and $S5$ share the sample testing set, but $S4$ is a superset of $S5$ with more negative samples. The results indicate that with the smaller training set, the results are actually better. This is expectable for 2-PNB and RLR, because the larger training set only includes more negative (benign) samples, the imbalance of positive and negative samples is more serious, hence hurting the training of models. This is especially the case for 2-PNB, which applies the positive (negative) sample prior to the testing set. If the positive sample ratio is quite different in training and testing sets, it hurts the classification performance of 2-PNB. It is interesting that the performance of both PNB and HMNB drops with more training data too. There may be two possible reasons. On one hand, as discussed in Section 2.4.1, the benign samples are only assumed benign since they are

from the Google Play store. There may be some malicious applications in the benign sample set, and mixing all the "benign" samples together actually increases the chance of having malicious samples in training. And both PNB and HMNB use only benign samples for training, there is no way they can avoid this effect without information from malicious samples. On the other hand, when the sample size increases, the generative models built by PNB and HMNB will become more general for fitting a variety of applications in the big training set, hence the probability of malicious samples may increase as well and this may cause the drop of performance.

2.4.8 Model Comparison with Source Code Feature

Performance of several methods using functional level source code features are compared in this section. Table 2.8 shows the results in $S5$. Uninformative prior are applied to PNB and 2-PNB. The results show that the discriminative model RLR is also better against the generative models with source code features. The results for PNB and HMNB with source code features are much worse than 2-PNB and RLR. It is mainly because some of the assumptions for these two one-class generative models to work with permission features are no longer valid for source code feature. For example, malicious applications tend to request more permissions but may not use more functions.

The results for K Nearest Neighbor (KNN), Decision Tree (DT) and Support Vector Machine (SVM) are also included here for comparison. The KNN ($K = 1$ selected from $K \in \{1, 3, 5, 7, 9\}$) is trained on $S5$ with the negative sample downsized, otherwise it cannot get any comparable results. Down-sampling is also utilized for DT to generate better result. SVM uses a radial basis function kernel and explores the parameter space using the training set to find optimal training parameters. KNN performs similar to 2-PNB, DT performs much better than KNN and SVM performs similar to RLR, but RLR has the advantage of directly generating probabilistic outputs.

2.4.9 Combination of Source Code and Permission Feature

It is natural to think whether source code features (i.e., function level) and permission features can be combined to further improve the performance of Android malware detection, since both types of features have been proven to be useful for the task. Table 2.9 shows the results of combining both features. The early fusion method combines the two types of features by concatenating them into one feature vector before applying the RLR. And the late fusion method trains RLR on both types of features separately and combine the two learned models afterwards with a meta logistic regression classifier. It can be seen from the table, using source code feature is better than using permission feature; the combined features work better than each individual type of features; and the late fusion gives the best result. The results suggest that although source code feature and permission features are related, they are still somehow complementary. For example, malicious applications may request permissions that may not be reflected by the functions. It is better to consider both types of features in detecting malicious applications in the Android platform.

In addition, Table 2.10 shows the confusion table of the best result (Combined (Late Fusion)) in $S5$. As we can see, this is a imbalanced dataset with 389 malicious apps from 5190 apps in total. The algorithm successfully found 374 out of 389 malicious apps (hence Recall=96.14%), and for the 392 reported suspicious apps, only 18 of them are false alarms (hence Precision=95.41%). Therefore the F1 value, as the harmonic mean of Recall and Precision, yields 95.77%. Meanwhile, the false positive rate is $\frac{18}{18+4783} = 0.37\%$, looks very optimistic due to the imbalance of the dataset.

2.5 Limitations

Despite the high performance provided by the proposed method, it is worth noting that the arm race between malicious application and malicious application detection technique is still on and the high performance observed in our data sets may not be applied directly to future situations. Therefore it would be very helpful to discuss about the limitations of the proposed method, together with the causes and effects.

As a static technique, this method shares the weaknesses of all static techniques on Android application. An application may utilize Reflection and Native Code [4, 32] to make its real program logic undetectable by static analysis. Other techniques like Bytecode Encryption etc. can also be used to hinder static analysis. We note however, that some tools use the features of using Bytecode encryption and usage of native code as indicators of potentially malicious intention. These are orthogonal to our method. Moreover, Obfuscation and dead code could cause trouble to our method, leading to unreliable feature extraction and representation. It would be an interesting future work to find out how to defend against them (*e.g.* dead code detection.).

Another limitation is introduced by the nature of supervised learning. The proposed method utilizes training data to build probabilistic model, under the assumption that the testing data, which the model will be applied to, is drawn from the same population as the training data. This assumption is generally not true in reality since malicious application may evolve. Hence the model need to be updated with new training data including new benign applications and new malicious applications. This disadvantage may limit the model's power in detecting zero-day malicious applications.

2.6 Conclusion

Android malware detection is an important research task as the Android platform is a leader in the fast growing market of mobile devices. This paper proposes to use probabilistic discriminative learning model with decompiled source code as well as permission features for this problem, and presents thorough evaluations and discussions for Android malware detection with decompiled source code and beyond. We propose a probabilistic discriminative learning model based on regularized logistic regression, which achieves highly accurate detection results (i.e., F1 value of about 0.95). Thorough studies have been conducted to explore desired representation of source code and appropriate modeling strategies, suggesting a combination of binary feature over function level of source code with lasso regularization for RLR. Furthermore, the discriminative learning model

has been shown to achieve even better results for Android malware detection by combining both source code and application permissions. Moreover, a discussion over the evaluation metric suggests that F1 may be a better evaluation metric rather than ROC curve due to the nature of unbalanced data between benign and malicious Android application.

Table 2.4.: Descriptions of datasets and experimental settings. The $*/2$ notation means using part of * at training set, and the other part at testing set.

Description of datasets				
Name	Size	After cleaning	Content	Description
$C11$	$71,331$	$69,179$	permission only	Collected from Google Play in February 2011, used in [1]
$C12$	$98,510$	$95,035$	permission only	Collected from Google Play in February 2012, used in [1]
CM	378	378	permission only	malicious apps obtained from authors of [29] in 2012
O	$12,801$	$9,571$	permission & .apk	Collected from Google Play in 2012 from authors of [16]
OM	$1,260$	808	permission & .apk	malicious apps obtained from authors of [29] in 2012

Description of Experimental Settings							
Setting name	Train			Test			Feature type
	Datasets	Size	Malicious	Datasets	Size	Malicious	
S1	$C11/2$	62,261	0.0%	$C11/2 + CM$	7,296	5.18%	Perm
S2	$C11 + C12 + CM$	164,592	0.23%	$O + OM$	10,052	4.79%	Perm
S3	$O + OM$	10,052	4.79%	$C11 + C12 + CM$	164,592	0.23 %	Perm
S4	$C11 + C12 + (O + OM)/2$	169,403	0.25%	$(O + OM)/2$	5,190	7.5 %	Perm
S5	$(O + OM)/2$	5,189	8.07%	$(O + OM)/2$	5,190	7.5%	Perm & Code

Table 2.5.: Performance of different regularization in $S5$

Reg.	F1
No Reg.	0.9268
Ridge	0.9460
Lasso	**0.9513**
Sim.	0.9319

Table 2.6.: Verifying the implementation of the models in [1]

	F1	AUC
PNB	0.5533	0.9380
HMNB	0.6042	0.9483

Table 2.7.: Performance comparison using permission features.

		PNB	HMNB	2-PNB	RLR
$S2$	F1	0.4106	0.4640	0.5039	**0.5911**
	AUC	0.8747	0.8666	0.9141	**0.9454**
$S3$	F1	0.1112	0.1082	0.1342	**0.2454**
	AUC	0.9223	0.9323	**0.9428**	0.9341
$S4$	F1	0.4241	0.4412	0.4192	**0.7196**
	AUC	0.8577	0.8620	0.9000	**0.9458**
$S5$	F1	0.4442	0.4720	0.6448	**0.8373**
	AUC	0.8717	0.8782	0.9285	**0.9661**

Table 2.8.: Performance comparison using source code features (function level) in $S5$. ACC is for accuracy.

	F1	AUC	ACC
PNB	0.1511	0.3711	-
HMNB	0.1396	0.3947	-
2-PNB	0.6579	0.9381	-
RLR	**0.9513**	**0.9961**	**0.9973**
KNN	0.6990	-	0.9549
DT	0.8448	-	0.9765
SVM	0.9385	-	0.9911

Table 2.9.: Performance of combined source code (function level) and permission features in $S5$

Feature	F1
Permission Only	0.8427
Source Code Only	0.9513
Combined (Early Fusion)	$\mathbf{0.9558}$
Combined (Late Fusion)	$\mathbf{0.9577}$

Table 2.10.: Confusion table for late fusion method in $S5$

		Actual class	
		True	False
Predicted class	Positive	374	18
	Negative	15	4783

3 USER COMMENT ANALYSIS FOR ANDROID APPS AND CSPI DETECTION WITH COMMENT EXPANSION

3.1 Motivation

In this chapter, we introduce our work on how to extract security related topics from user comments. User comment is an indirect source of information that reflects the experience of users in using the app. Before we can apply security analysis from it, we need to first filter out the irrelevant comments which are the majority of all the comments and detect the different security related issues discussed in user comments.

3.2 Introduction and Related Work

3.2.1 Introduction

New challenges come with the exponentially growing markets of mobile apps. On one side, comparing to traditional software markets, markets like Google Play and Apple Store have lower entering threshold for developers and faster financial payback, hence greatly encourage more and more developers to invest in this thriving business. One result out of this is the huge amount of mobile apps with great diversity. Therefore, to control the quality of apps, especially the security risk of them across the whole markets becomes a important issue to all that involved. On the other side, public concerns about privacy issues with online activity and mobile phones are also elevating, demanding a mobile environment with more respect to users' privacy.

The infection rate of real malicious mobile apps over a market can only be estimated. It is reported to be about 0.28% in [33]. The rest of them are assumed benign apps but are not free of security issues. Many misbehaviors of a mobile app may lead to real security/privacy issues. For example, adding too much or inappropriate advertisement may lead to phishing

and scareware; unresolved In-App-Purchase (IAP) may lead to fraud; unnecessary running in background may be connected with unauthorized access to personal data.

User comments are valuable feedback for both new users and developers. Many security/privacy issues can be revealed by user comments, including those which may not be so easy to be discovered from other sources (i.e. unresolved IAP issue). However, most of the negative comments are not necessarily Comments with Security/Privacy Issues (CSPI). Some of them are non-informative comments [34], including vague statement and pure emotional expressions. Others may be complaining about attractiveness, quality or even cost of the app [35], which normally have nothing to do with security/privacy. In order to make use of the comments to reveal the issues that are really related to security/privacy of an app, CSPI need to be detected first to avoid all those irrelevant comments. This paper provides a novel method to deal with this problem. A set of comments are first collected and investigated. Based on observations from the comments, a two dimensional label system is designed to describe CSPI from two different perspectives. With this label system, a CSPI Detection with Comment Expansion (CDCE) method is proposed to first use keyword-based filtering method to narrow down the scale of comments in concern, then applies a supervised multi-label learning method to identify different types of CSPI.

The contribution of this paper are listed as following:

- Instead of using a list of label for different issues, this paper presents a two-dimensional label system, picturing the "What" and "When" of the occurrence of a reported CSPI, providing an additional dimension to better understand the relationship between different issues.

- A supervised learning method is proposed to solve this multi-label problem. Comment expansion is adopted to utilize the relationship between comments and a post-process for the relationship between labels. Both relationships are proven to be useful in improving the CSPI detection performance based on the collected dataset.

The rest of the paper proceeds as follows. Section 3.2.2 describes related works. Section 3.3 discuss the collection of data and the design of the label system. Section 3.4

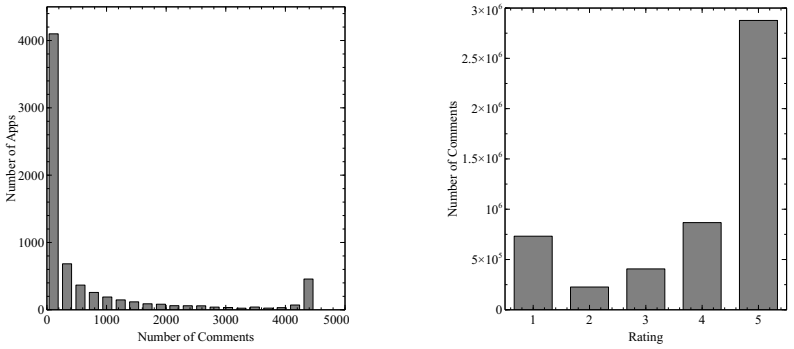
presents the method for CSPI detection. Section 3.5 demonstrates the evaluation process of the proposed method. Section 3.6 lists the limitation of this paper with future work and Section 3.7 concludes the paper.

3.2.2 Related Works

Many efforts have been done to reveal the security risk of mobile apps using information other than user comments. Some works [1,4] utilize the permission required by the app for this purpose, other use code analysis[], static and dynamic analysis are both investigated. Although these works share the general purpose with this paper, they are not closely related to this work. User comment are utilized by some works [34–36] to evaluate mobile apps. Some researchers [36] aim at the task to extract new/changed requirement for new version of the app. It proposes Aspect and Sentiment Unification Model (ASUM) to extract the topics of comments. ASUM is an extension of Latent Dirichlet Allocation (LDA) [37] by incorporating both topic modeling and sentiment analysis. Another work [35] uses LDA model to identify different topics from those negative comments, in order to provide insight about why an app is getting low ratings. Our work focus on a finer level to investigate only CSPI, which is part of the negative comments. Also, some researchers [34] propose a novel method for extracting informative review topics from user comments. A filtering process is applied first to filter out non-informative comments, then LDA is adopted for generating topics from the informative ones. Our work also requires a filtering process, but not from the perspective of the quality of information, but whether the comment is related to security/privacy. All these works mentioned above make use of topic models (LDA), which is an unsupervised method. For our work, the task is not to generate general summarization of the topics of comments of an app, but to identify a specific part (CSPI) of all the comments and further distinguish different types of it. Hence unsupervised method like LDA may not be able to generate the expected and specified types of topics. Therefore, supervised method is adopted in our work, and a label system is manually designed to provide precise task for the learning process.

3.3 Data Collection and Annotation

The data used in this work is collected by crawling the Google Play website. Information about $6,938$ free apps in Google play are downloaded during September through December in 2013. For user comments, the total number of comments collected for these $6,938$ apps is $5,108,538$. The average number of comments for an app in this dataset is 736 and the maximum number is 4500. Figure 3.1(a) shows the distribution of number of comments among the apps.



(a) Histogram for the number of apps against the number of its comments in the collected dataset.

(b) Histogram for the number of comments with different ratings in the collected dataset.

Figure 3.1.: A simple view of the collected dataset.

The rating from user comments are highly skewed as shown in Figure 3.1(b) (this is previously also reported in [35]), indicating that most of the comments are not complaining about an issue. In addition, many of the comments with poor ratings (< 4.0) are not really about security/privacy issues. As observed from the dataset, complains about the functionality and quality (or attractiveness for Game app) take the majority. Therefore it would be necessary to narrow down the huge set of comment to a more feasible size for further analysis and annotation. In order to do that, a coarse filtering is applied first to create a set of suspect comments. This filtering is keyword-based, as any comment that contains at least one of the predefined keywords will be put into the suspect set. These keywords are manually picked in an iterative way. The initial set of keywords is just $\{security, privacy\}$.

New keywords are picked from those that have high co-occurrence probability with current keywords. Comments with the co-occurrence are investigated manually to see how often they are related to security/privacy issues. The final keywords set includes *security, privacy, permission, money, spam, steal, phish*, etc.. To avoid mismatch, different forms of the keywords are also considered, i.g. $unsecure$ and $insecure$ for $security$, $stole$ and $stolen$ for $steal$ etc. A rating filtering is also applied to only include those comments with poor ratings(< 4.0). The resulting suspect set contains $36,464$ comments from $3,174$ apps.

3.3.1 CSPI Annotation

It is worth noting that the suspect set still contains not only CSPI. Many of them are not CSPI, due to the simplicity of the keyword-based coarse filtering. Hence the suspect set only serves as a base candidate set of comments which are suspect for CSPI. Further effort need to be done to actually detect CSPI. And different types of CSPI need to be distinguished and treated accordingly.

Various different issues are found from the comments in the suspect set. It would be exhausting to distinguish each of them without abstract concepts by considering the relationship between them. Also, it would be too vague to just distinguish CSPI from non-CSPI comments without further insight of the actual issues. To make a trade-off between the complexity and functionality of the annotation system for describing CSPI, a label *General* is firstly defined for comments that is in general CSPI. In addition, a two dimensional (*Nature*, *Scenario*) label set is defined as shown in Table 3.1 to provide finer level identification for different CSPI topics. Among these labels, label *Execution* is a super label for *Foreground* and *Background*. And all are sub-label of label *General*.

The two dimensions of label set serve as "What" (*Nature*) and "When" (*Scenario*) of the underlining issues. The dimension *Nature* is used to identify the nature of the security/privacy related misbehavior of the app complained by the comments. These behaviors are normally described explicitly in the comments. On the other hand, the dimension *Scenario* is used to identify the scenario when the issues occur. This is sometimes expressed

implicitly in the comments. System, Privacy and Finance are clearly necessary as the direct issues. The label Spam is assigned to the widely used advertising behavior among apps. As a popular and common behavior, however, spamming is closely related some security issues like phishing, scareware etc., hence is also included as one type of issue mentioned in CSPI. The label *Others* is used for other issues not included in *System*, *Privacy*, *Spam* or *Finance*. For example the topic about some apps that can not be normally uninstalled, it is usually an app pre-installed by the vendor without any real issues, but it clearly violates the users' control over the phone and is complained about by many users. Another example for *Others* labeled comments would be those claims that the app is reported by other security apps for some reason. Since these are not direct opinions from the users but only a suspects, they are put with *Others*, instead of with the other four ones according to the reasons said to be reported by other security apps, which may not be true or even explicitly expressed in the comment. The label *Before* is mostly assigned together with *Privacy* to indicating the complains about the permission required by the app, which is available to the user before the installation. And the label *After* is mostly for the email/SMS spam after uninstall. The label *Execution* stands for the most common scenario, and it is further divided into two sub-labels: *Foreground* for the issues occur when the app is running in foreground (occupying the screen) and *Background* for running in background. If the issue in the comment is not specific about foreground or background, only the label *Execution* is applied, otherwise both *Foreground* (*Background*) and *Execution* will be applied.

The suspect set is then annotated manually with these labels. Each label is validated by the agreement of two people. Some statistics of the suspect set with respect to the labels are shown in Table 3.2.

About 30% of the comment in the suspect set are labeled as CSPI. This also means many of the comments, while mentioning some keywords, are not really talking about security/privacy issues. The sample comment for *Whole Set* is one of them. It mentions *stolen* not to claim a financial issue but a issue about stolen idea, which is not about security or privacy. Some comments talk about "money", but only to express that the app is not worth it (Although all the apps we collected are free apps, people can still talk about In-

App-Purchase and the paid version of the app.). Another example would be the comments for the apps about security (Anti-virus apps, password manager, etc.). A lot of keywords are mentioned in these comments since the apps are exactly about the issues, but the comments are not necessarily reporting an issue of the apps. Similar things happen with Bank apps. Label *Others* has a very low percentage, indicating the coverage of the four main natures of issues is good.

It is worth noting that misspelling may hinder the performance of detection. The "baught" in the sample for *Finance* serves as an example. Also of important is the fact that a CSPI usually has two or more labels besides *General*. For example, the sample comment for *Spam* is also annotated with *Background*, since it mentions that the spam appears in the notification bar. And the one for *Before* is also annotated with *Privacy*.

3.4 Algorithm

The purpose of CSPI detection is to detect certain types of CSPI in the suspect set. This is naturally a multi-label classification problem. Independent Logistic Regression (ILR) is used as a baseline. The proposed CDCE method adds two improvements upon it: Comment Expansion to embed similarity between comments and Post-Process with label correlation to utilize label correlation.

3.4.1 Feature Extraction

To represent comments for detection, Bag Of Word (BOW) feature is extracted from comment text. Proper pre-processes are applied to the text before feature extraction, including removing stop words and stemming. The dimension of the BOW feature is tuned by removing the rarest and most popular words. With the BOW feature and the annotation with the 11 labels, the suspect set can be represented as X and Y. $X = \{X_1, X_2, \cdots, X_N\}$ is the set of the BOW feature vectors for each comments with N the number of comments. And $Y = \{Y_1, Y_2, \cdots, Y_N\}$ is the set of label vectors for each comments with

$Y_i = \{Y_i^1, \cdots, Y_i^{11}\}$ and $Y_i^j \in \{-1, 1\}$ with $Y_i^j = 1$ indicating the presence of the jth label for comment i.

3.4.2 Independent Logistic Regression (ILR)

There are many works that have been done on multi-label learning from different perspective. G. Madjarov [38] made an extensive empirical comparison of different multi-label learning methods. In the comparison, Binary Relevance (BR) [39] performs similar to classifier chaining (CC) method [40]. Although not the best method among all the competitors, BR shows robust performance and has the advantage of simplicity. For the task of CSPI detection, this paper does not seek to investigate and improve the multi-label learning algorithm in general. Therefore, ILR is chosen as the baseline, which is just BR framework with LR [41] as the base classifier. LR is a widely adopted linear classifier due to its robustness and simplicity. To apply LR to the multi-label problem considering independence between labels, one LR classifier is trained for each of the labels. The objective function of ILR with 2-norm is as following:

$$\min_{w_j, b_j} NLL(X, w_j, b_j) + \lambda(|w_j|_2^2 + |b_j|_2^2)$$

with

$$NLL(X, w_j, b_j) = \sum_{i=1}^{N} ln(1 + exp^{-y_i^j(w_j^T X_i + b_j)}), j = 1, \cdots, 11$$

where the $NLL(X, w_j, b_j)$ is the negative log-likelihood function. λ is the regularization parameter which can be obtained by cross validation in training set. This optimization problem can be solved using gradient descent.

3.4.3 Comment Expansion

User comments has some properties distinguishing them from properly compiled documents. They are normally short, with wrongly spelled words (as mentioned in Sec-

tion 3.3.1) and made-up words (i.g. "spamspamspamspam"). Also, different words or phrases may be used for the same opinion. These properties of user comments may harm the performance of CSPI detection since the features may not fully represent the opinion of the comments. This is in a similar situation with the user query in Information Retrieval (IR) problem, where a query submitted to search engine could also be short, misspelled and vary in the choice of words. Due to this motivation, a traditional IR technique called query expansion with pseudo relevant feedback [42] is borrowed here to make comment expansion. Originally, this technique uses the top documents in the retrieval result rank list with respect to the original query as "relevant" documents, and use these document to expand the original query, generating a new query resembling the "relevant" documents. This query expansion is reported to almost always have a positive effect on the retrieval performance [42]. A similar process can be adopted for comment expansion as following. The comment expansion would have two steps:

1. Find "relevant" comments via retrieval.

2. Expand original comment with "relevant" comments.

The relevance between comments is hereby evaluated using the retrieval model. An interesting question would be the scope of the retrieval. Should the candidate "relevant" comments picked from all other comments? or just from the comments from the same app or same category of apps? Besides, the "relevant" comments should only be those posted before the underlining comment. To avoid making complicated query to the retrieval engine, the query contains only the underlining comment with no constrain. A sufficiently long rank list from the retrieval engine will be returned and the scopes and restrictions will be applied to this list to pick "relevant" comments afterwards.

With a set of "relevant" comments, the comment expansion can be conducted by making a convex sum of the original comment and the mean of the set of "relevant" comments on feature level as following:

$$f_{new} = (1 - \alpha) f_{old} + \alpha \cdot \frac{1}{|R|} \sum_{f \in R} f$$

where f_{old} is the BOW feature vector of the original comment, f_{new} the feature vector of the expanded comment, and R the set of "relevant" comments with respect to the underlining comment. α serves as a tunable parameter for the degree of effect of the expansion, and will be tuned in experiment for the best performance, so is the size of R.

Comment expansion is an efficient way to utilize the relationship between comments. Other choices like kernel method may require the similarity computation between each pair of comments, the amount of computation grows exponentially with the number of comments. Comment expansion, on the other hand, relies on retrieval engine, and the cost of retrieval for one comment is normally constant, hence the total cost of time is linear to comment number.

3.4.4 Post-Process with Label Correlation

Label correlation is used in various ways in multi-label learning [38]. As mentioned in Section 3.3.1, labels hardly appear alone. Hence the correlation between labels could be a valuable source of information. For the CSPI detection task, a simple post-process is applied to embed this information. The post-process uses a second round of ILR with different feature for the comments. The probability output $\hat{Y}$ of the first round ILR classifiers are used as feature in this second round of ILR to predict Y. And the correlation matrix of the labels are utilized into a Laplacian norm [28] in these LR problems. Let $\hat{Y} = \{\hat{Y_i}, \cdots, \hat{Y_N}\}$ the estimated label vector set from the ILR. The objective function of the second round LR is as following:

$$\min_{\hat{w_j}, \hat{b_j}} NLL(\hat{Y}, \hat{w_j}, \hat{b_j}) + \hat{\lambda}(\hat{w_j}^T L \hat{w_j} + |\hat{b_j}|_2^2)$$

with

$$NLL(\hat{Y}, \hat{w_j}, \hat{b_j}) = \sum_{i=1}^{N} ln(1 + exp^{-y_i^j(\hat{w_j}^T \hat{Y_i} + \hat{b_j})}), j = 1, \cdots, 11$$

where $L = A - D$ is the Laplacian matrix with A the correlation matrix of the labels Y in training set, and D a diagonal matrix with its diagonal elements the sum of each row of A. These optimization problems can be solved similarly using gradient descent.

3.5 Experiments

3.5.1 Experiment Setting

The evaluation of the proposed CSPI detection method is conducted under the suspect set of comments. As a supervised method, a training set is required for training the model. The suspect set is split in a $50/50$ manner into a training set and a testing set. This splitting is based on app level, so the comments for the same app can only be in training or testing set altogether. The BOW feature vector is of length $13,135$ by removing those words with less than 100 or more than $1,000,000$ appearance (in number of comments) in training set. Considering the hierarchy in label set, a comment labeled with a sub-label will be considered a positive sample for a super-label as well. And a sample labeled with a super-label will not be considered either a positive or negative sample for a sub-label. For the baseline ILR, 5-fold cross validation is adopted for finding λs in training set and L-BFGS quasi-Newton method [27] is applied for solving the optimization problems. For comment expansion, TF-IDF feature with cosine similarity is adopted for the retrieval model and Lemur[1] as the actual tool for the retrieval. Time constrain is enforced to prevent comment expansion with "future relevant" comments. Set constrain is applied to only allow expansion within training/testing set respectively, and the indexes of retrieval model are built for the two sets separately so that the model parameter like document number and IDF values won't interfere between sets. Three types of scope: All, App, Category are tested with each label and a 5-fold cross validation is used to pick the best of the three in training set for each label. Also, the mixture ratio α and the size of "relevant" document set $|R|$ for the expansion are also fixed by the 5-fold cross validation in training set for each label along with the scope. The size of $|R|$ is selected from $\{1, 3, 5, 10\}$. For post-process with

[1] http://www.lemurproject.org/

label correlation, the problem solving method is similar to the baseline ILR. The metric for evaluation is micro F1 value. F1 value is the harmonic mean of Precision and Recall. And micro F1 is computed across all sample and all label at once.

3.5.2 Results and Analysis

The comparison between the proposed CDCE method and the baseline in F1 value is shown in Table 3.3.

The CDCE^{-} method indicates the method using only comment expansion without the post-process. The CDCE^{+} method indicates the method using comment expansion and post-process on all labels. And the CDCE* method indicates the method that pick the models between CDCE^{-} and CDCE^{+} by cross-validation for each labels. This evaluation is based on 10 different training/testing sets splitting, and the reported micro F1 values are the means over these ten settings.

By comparing CDCE^{-} and ILR, the general improvement from using comment expansion is obvious. The expansion makes comments "smoother" among similar comments in feature level, and improves the performance of the classifiers against short comments and those with misspelled words. For example, "ads" may sometimes be misspelled to be "add", the expansion adds similar comments' feature into the underlining feature and the feature dimension with respect to "ads" may not be zero anymore, hence the classifier can capture this feature and tend to put the label "Spam" on the comment. On the other side, the effect of expansion is not always positive for all samples. For example, for comments about a Game app, many ones may be talking about "money" because they paid for some item in the game but never got it. These comments should be labeled "Finance". But one comments for the same app may be just talking about how expensive that item is, hence not worth the "money". This one however, is not a CSPI. After expansion, this comment will look much like the others hence be labeled "Finance" as well. Therefore a negative effect of comment expansion is that it may silence some different voice that are making different

points while using similar words like others. Nevertheless, the general effect of comment expansion is obviously positive.

The difference between CDCE$^-$ and CDCE$^+$ shows that the post-process does not guarantee an improvement for all labels. Hence CDCE* method is propose to pick a better model between CDCD$^-$ and CDCE$^+$ for each label from training set. CDCD* provides the best performance among others.

A label level comparison between CDCE* and ILR can be found in Table 3.4, where CDCE* appears to outperform ILR for all labels except *After*. The model behavior for label *After* is different from others mostly due to lack of sample. There are only 36 samples to be split into training and testing set, which practically makes the training of model insufficient and the comment expansion mostly uses comments with different labels as "relevant" comments. Besides *After*, three labels:*Before*, *System* and *Others* do not pass the statistical significant test at α level 0.05, with p-value 8.2%, 5.5% and 8.1% respectively. Other than these, the highest p-value is 0.9% from *Finance*.

The results of picking better model between CDCE$^-$ and CDCE$^+$ are also shown in the P.P. column of Table 3.4. It appears that all *Nature* labels are not suitable for the post-process, but the *Scenario* ones get a boost based on the results in Table 3.3. The post-process makes use of the ILR result as feature to predict a label. This prediction may be improved by getting correlation information from other labels, but also suffer from poorly predicted results from ILR. Important words (features) for the *Nature* labels are not so diverse as those in *Scenario*. For example, for label *Spam*, no matter what *Scenario* label come with it, the comment would probably still use the words like "spam" or "ads". Similarly for *Privacy* there are "privacy", "invade", or "permission". To distinguish *Scenario* labels, however, different words are of importance under the condition of different *Nature* label. If given *Spam*, *Foreground* is related to over-sized on screen ads, and *Background* most likely to notification bar spamming. On the other hand, if given *Privacy*, *Foreground* may be related to phishing and *Background* to personal information stealing or abuse. Hence the correlation information may be much more helpful for *Scenario* labels than *Nature* ones.

The scopes of retrieval in comment expansion for each label are listed in the Type column of Table 3.4. None of classifiers choose to use All comments as candidates for "relevant" comments, and the scope of using the comments of the same App or same Category (Cat.) of apps are both popular. The scope of All comments makes the "relevant" comments too diverse and noisy and the expansion normally lead to some unexpected result. One the other side, App and Cat. scope serves well, providing much high probability of getting "relevant" comments with both similar text content but also similar topic of issues.

3.6 Limitation & Future Work

As a comment level analysis, one limitation of this work is that it does not provide a risk assessment on the app level. How to evaluate the app's security/privacy risk base on the identification of CSPI would be an interesting work in the future. Another limitation comes from the coarse filtering, where CSPI that do not contain any keywords could be left out by the method. This may due to the variety of language itself or simply misspelling or using made-up words instead of the keywords. Hence further research may lie on how to expand the suspect set as a trade-off between computation overload and detection performance.

3.7 Conclusion

In this paper, a supervised learning method is proposed to detect CSPI for Android apps. This task is formalized as a multi-label learning problem with a two dimensional label system with respect to "What" and "When" of issues reported in CSPI. A coarse filtering is first applied to narrow down the set of comments as suspect. Then comment expansion is adopted to improve the representativity of the feature by making convex combination of the original feature with those of "relevant" comments. Finally, a post-process is used upon some of the labels to make use of the label correlation for further improvement. Experiment results on the collected dataset shows statistical significant improvement in general against ILR as a baseline method.

Table 3.1.: Two dimensional label set

	Label	Definition	Issues
Nature	*System*	Issues causing negative effect to the system	Freezing the phone, unauthorized downloading
	Privacy	Issues about getting unauthorized access to user info.	Stealing phone number, accounts, emails
	Spam	Issues about unpleasant ads and related.	Annoying ads, spam shortcut on home screen.
	Finance	Issues about suspect money stealing.	unresolved purchase.
	Others	Security / privacy issues not included above.	uninstall issues
Scenario	*Before*	Issues occur before installation of the app.	Requiring too much permissions.
	Execution	Issues occur when the app is executing on the phone.	General complains about spam
	Foreground	Issues occur when the app takes the foreground screen.	Ads occupies too much screen, phishing
	Background	Issues occur when the app is running background.	Notification bar spam
	After	Issues occur after uninstall of the app.	email/SMS spam after uninstall

Table 3.2.: Statistics and sample comment pieces on suspect set

	#	%	Sample Comment
Whole Set	36,464	100%	"Idea *stolen* from ***, and it feels unfinished, or unprofessional"
General	10,636	29.17%	all of the below.
System	1,304	3.58%	"It keeps crashing the phone. It becomes a device administrator."
Privacy	2,513	6.89%	"*Leak*ing GPS location to dvertisers=Top 2 fastest ways to get me to hate your app/guts."
Spam	6,164	16.90%	"Since installing this app I get *spam* in my notifications constantly."
Finance	1,191	3.27%	"I baught $2.00 in *** and I never got them. In really mad about it."
Others	191	0.52%	"I buy my own phone with my own *money* and I cant delete this app from my phone?"
Before	1,611	4.40%	"New versions *permissions* can *steal* all my data and contacts!!!"
Execution	8,456	23.19%	"No need for them to *invade* my *privacy*."
Foreground	2,011	5.52%	"game wont even load due to the pop ups before game starts..splash sceen pop ups suck"
Background	4,171	11.44%	"*spam* emails that I did not send were going out from my email address."
After	36	0.10%	"six months after uninstall, *** *Spam* just keeps on coming"

Table 3.3.: Experiment results on suspect set. † shows the statistical significance based on ILR. It is computed over ten different random splits of the training/testing sets, using one-tailed pair-wise t test with $\alpha = 0.05$.

method	micro F1			
	General	*Scenario*	*Nature*	All
ILR	0.7962	0.6713	0.7032	0.7153
CDCE^{-}	0.8037†	0.6740†	0.7159†	0.7223†
CDCE^{+}	0.8004†	0.6814†	0.7096†	0.7225†
CDCE*	**0.8037**†	**0.6836**†	**0.7159**†	**0.7263**†

Table 3.4.: Detailed comparison in label level. The † is computed at $\alpha = 0.05$ with one-tailed t test among 10 different training/testing set splits.

Label	F1		P.P.	Type
	ILR	CDCE*		
General	0.7962	**0.8037**†	No	App
Before	0.6965	**0.7012**	Yes	Cat.
Execution	0.7277	**0.7358**†	Yes	App
Foreground	0.4347	**0.4689**†	Yes	Cat.
Background	0.6674	**0.6750**†	No	App
After	**0.1327**	0.0882	No	App
System	0.3991	**0.4012**	No	App
Privacy	0.7264	**0.7350**†	No	Cat.
Spam	0.8181	**0.8304**†	No	Cat.
Finance	0.5238	**0.5320**†	No	Cat.
Others	0.0235	**0.0384**	No	App

4 MOBILE APP SECURITY RISK ASSESSMENT: A CROWDSOURCING RANKING APPROACH FROM USER COMMENTS

4.1 Motivation

With extracted security related topics from user comments, this chapter demonstrate how we can utilize crowdsourcing algorithm and learning to rank to estimate security risk of mobile apps. Two major problem answers here are how the aggregate security topics from user comment level to app level, and how to estimate the security risk from that.

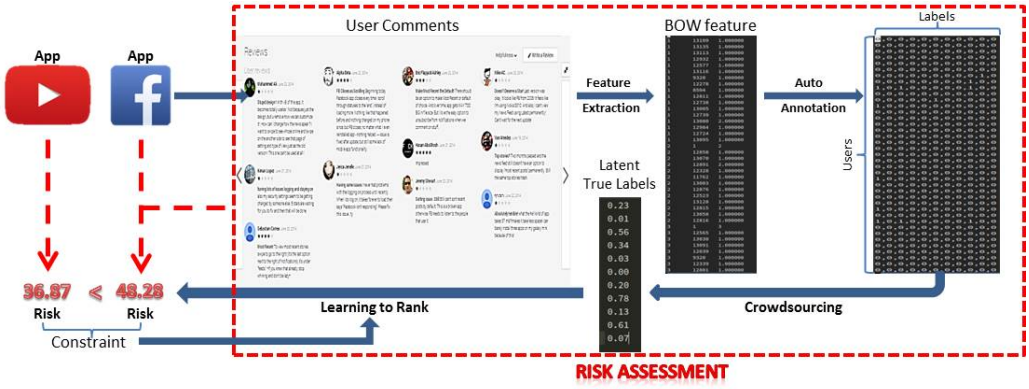


Figure 4.1.: Flowchart of security risk assessment from user comments. Given any app (*e.g.*, `Facebook`), the user comments are collected from `Google Play Store`. In order to infer the security risk of apps, (1) crowdsourcing is used to accumulate user comments into app-level features (shown as "feature extraction","auto annotation" and "crowdsourcing"); (2) *learning to rank* model is used to predict risk scores by utilizing these latent features, where pairwise constraints are enforced between pairwise apps (shown as the relative risk levels of `Youtube` and `Facebook`).

4.2 Introduction and Related Work

4.2.1 Introduction

New challenges come with the exponentially growing markets of mobile apps. On one hand, comparing with traditional software markets, markets like Google Play and Apple Store have lower entry threshold for developers and faster financial payback, hence greatly encouraging more and more developers to invest in this thriving business. One result out of this is the huge amount of mobile apps with great diversity. Therefore controlling the quality of apps, especially the security risk of them across the whole markets, becomes an important issue to all that involved. On the other hand, public concerns about privacy issues with on-line activity and mobile phones are also elevating, demanding a mobile environment with more respect to users' privacy.

User comments are valuable information source for evaluating mobile apps' security and privacy aspects from users' perspective. They are indirect source comparing with the apps' permission requests and binary code, yet they reflect users' experience of and expectation for the apps. After all, whether an app is risky for the users' privacy depends on how the users think about it.

However, user comments are noisy and diverse. And the distribution of user rating shows that user tend to give positive feedback [35]. Therefore how to aggregate comments from multiple users to assess the app would be an interesting problem. Two questions (**Q1, Q2**) need to be answered for solving this problem:

Q1: In order to assess the app from its comments, a set of features of the app representing the security/privacy aspects need to be extracted from the comments. But since the comments are noisy and usually not about security issues and sometimes even not about any issues at all, the challenge is how we can trust the extracted features to truly represent the app for its security risk.

Q2: Security/privacy risk is not an objective term. The risk score of an app will not mean much to a user unless it is compared to other apps. Therefore a question raises: how can we compute the risk score considering the relationship between apps?

For the first question, we suggest building comment level features, then aggregate from comment level to app level. We do not place full confident on the comment level features, but on the app levels which could be learned from the crowds by evaluating the quality of the comments, *i.e.,* the expertise of the users. For the second question, we suggest adopting *learning to rank* model, which can rank apps with respect to their risk scores based on the annotated risk labels.

In summary, this paper presents a crowdsourcing ranking approach to solve the app risk assessment problem from users' comments. The main contributions are highlighted as follows.

- To the best of our knowledge, this paper is the *first one* that treats the inference of security risk problem as a task of crowdsourcing problem from aggregation of user comments.

- A novel 2-stage model is proposed, which can jointly learn the latent security labels from user comments and automatically rank the risks of app based on these learned labels as features. An effective alternative optimization algorithm is proposed to solve the corresponding optimization problem.

- Extensive experiments on two *real-world* datasets show the substantial improvement of our method, i.e., **6%–7%** performance improvements when compared with other state-of-the-art methods.

- Last but not least importantly, our approach can be easily extended for understanding the websites' popularity through users' comments, such as inference of goodness or badness of a restaurant from users' comments on `Yelp`, or the quality of a product on `Amazon` or `ebay`. Since the label system is arbitrary, one may use the same user comments data with different label system design to evaluate different aspects of the product.

4.2.2 Related Works

There has not been much work on risk assessment on mobile apps. Liu *et al.* [43] presented a framework for estimating privacy score for users based on their participations in social network, instead of mobile apps. Peng [1] utilized the permissions required by the apps to classify benign apps from the malicious ones. WHYPER [44] predicted the risk assessment of mobile apps based on analysis of the descriptions of apps from natural language processing perspective using first-order logic. Kong *et al.* [45] predicted the permission required by mobile apps from descriptions. Frank *et al.* [46] analyze the permission request using unsupervised learning (i.e., boolean matrix factorization) to discover the inherent cluster patterns of permission request. In addition, users' privacy preference can also be utilized in personalized mobile app recommendation [47]. However, none of the above works evaluate the risk score of mobile apps from the perspective of user comments. Our work provides a new angle for estimating the app security risks.

The discussion of crowdsourcing problem could be traced back 35 years ago [48]. The main idea is to infer true label of a given item from the annotations of multiple annotators/workers. These annotations are assumed to be of low quality and may contradict with each other. Two-coin model [49] is proposed to model worker expertise, by considering the probability that a worker labels an item correctly which follows two Bernoulli distributions, one for the true positive label, and the other for negative. Minimax entropy is adopted in [50, 51] to model both users and items. The maximum entropy principle is used to naturally infer both item confusability and worker expertise. Guided crowdsourcing [52] is another direction of research, which uses artificial intelligence methods to coordinate workers in crowdsourcing tasks, in order to ensure collective performance goals such as effectiveness, cost or efficiency. Our task is not a traditional crowdsourcing problem, we model the mapping from user comments to app-level latent true labels as a crowdsourcing problem, treating user comments as labels from workers.

Learning to rank is a well studied problem. There are three paths for solving *learning to rank* problem. Point-wise approach (*e.g.,* [53, 54]), computes the model loss from

point-wise perspective, hence can be closely related to regression. In pair-wise approach (*e.g.,* [55–57]), the loss is defined on pair of samples instead of individual sample points. For example, work [55] formalizes the pair-wise learning to rank problem as a maximum margin problem, and can be solved by Support Vector Machine (SVM) [41]. List-wise approach (*e.g.,* [58, 59]), formalizes the loss function directly on rank list. Our work follows the pair-wise approach to rank apps based on their security risks, and evaluates the performance of the proposed method using ranking evaluation metrics.

4.3 Algorithm Overview

4.3.1 Problem Formalization

Given a training set of n_l apps: $(\mathbf{a}_1, \mathbf{a}_2, \cdots, \mathbf{a}_{n_l})$, with their corresponding user comments $\mathbf{X}_{n_l} = (\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_{n_l})$, and their corresponding risk scores $\mathbf{S} = (s_1, s_2, \cdots, s_{n_l})$. The task is to estimate the risk scores $(s_{n_l+1}, s_{n_l+2}, \cdots, s_{n_l+n_u})$, for n_u new mobile apps: $(\mathbf{a}_{n_l+1}, \mathbf{a}_{n_l+2}, \cdots, \mathbf{a}_{n_l+n_u})$ with their corresponding user comments
$\mathbf{X}_{n_u} = (\mathbf{x}_{n_l+1}, \mathbf{x}_{n_l+2}, \cdots, \mathbf{x}_{n_l+n_u})$.

At the first glance of the problem, this can be solved using a standard supervised learning method. However, the challenge of this problem is how to represent the comments from different users, and leverage the different users' opinion into the security assessment model. It is not straightforward how to amalgamate the comments from different users into a reliable feature representation of apps. Moreover, note that security risk of a mobile app is more of a relative concept than an absolute metric. It is more reasonable to talk about security risk by comparing two apps, rather than assigning an absolute numerical value to an apps. Therefore, ranking model is motivated to be introduced to measure the security risks among different apps.

4.3.2 Approch Overview

Basically, our approach solves the mobile app risk assessment problem in two steps.

⊙ The first step is to generate features for each app from its user comments, which is modeled as a *crowdsourcing* problem, where user comments are viewed as "annotations" to the apps and the crowdsourcing results are viewed as features of each app. The goal of this step is to compute the app-security level features $\mathbf{Y}$ via the crowdsourcing model.

⊙ The second step is to generate the risk score by utilizing these features $\mathbf{Y}$ generated in the first step, which is modeled as a *learning to rank* problem, where the relative risk score is computed.

⊙ The above two steps can be done jointly to optimize the mobile app risk ranking results.

Figure 4.1 illustrates the framework of app risk assessment process, the detail of which is discussed below. Note actually the two key steps in our method (§4.2, §4.3) can be solved individually, without any communication. The major drawback of separate treating is that, the risk score is computed in one-shot using the obtained features from crowdsourcing, and thus we cannot consistently improve the models with the learned better features and adjusted risk ranking scores, and thus its performance may not work very well in practice.

Step 1: Feature Learning via Crowdsourcing

In order to capture the security risks of different apps, a set of security related labels $L = \{L_1, \cdots, L_r\}$ is defined to describe different security problems mentioned in user comments $\mathcal{X} = \mathbf{X}_{n_l} \bigcup \mathbf{X}_{n_u}$. Then for each user u's comment from each app i: $X_{u,i}(1 \leq u \leq m, 1 \leq i \leq n)$, it is associated with several labels, where m is the number of users and $n = n_l + n_u$. Comment examples for an app are given below:

---

comment:`Seemed ok, but the morning after signing in my account spammed everyone on my contact list with junk mail from me... could be a coincidence...`

labels: `background, spam, privacy`

User: `John Smith`

---

comment: `Why does this need to send sms messages and make calls? Updates not that timely. Uninstalling.`

labels: `before install, privacy`

User: `Jane Doe`

Motivated by our previous work [60], a set of classifiers is then trained for each label. These classifiers serve as auto annotation tools that map the raw text of user comment of each app to a list of security related labels in L. It is worth noting that, the auto annotation process is applied to both training and testing set as pre-processing. For the rest of this paper, $\mathcal{D} = \mathbf{D}_{n_l} \bigcup \mathbf{D}_{n_u}$ is denoted as the annotation results after pre-processing, *i.e.,* labels of the comments, instead of the raw comments $\mathcal{X}$. Moreover, the annotated labels are treated as annotated by different users in the context of crowdsourcing scenario.

More formally, for each user u w.r.t app i, its annotated comments $\mathbf{d}_{u,i} \in \{0,1\}^r$ is denoted as a r-dimensional set, $\mathcal{D} = [\mathbf{d}_{ui}]$, *i.e.*, $\mathbf{d}_{u,i} = \{d^1_{u,i}, \cdots, d^r_{u,i}\}$, where r is the size of label set, and $d^\ell_{u,i}$ the binary auto annotation result of the ℓ-th label for the comment given by user u for app i, *i.e.*,

$$d^\ell_{u,i} = \begin{cases} 1; \text{if user } u \text{ label app } i \text{ as label } \ell \\ 0; \text{otherwise} \end{cases} \tag{4.1}$$

Given all the above labels $\mathcal{D}$ generated from raw comments of different users for each app, we need to learn a mapping which automatically projects the comment-level labels $\mathbf{d}_{u,i} = \{d^1_{u,i}, \cdots, d^r_{u,i}\}, 1 \leq u \leq m$ to app-level security labels $\mathbf{y}_i = [y^1_i, y^2_i, \cdots, y^r_i]$, *i.e.*,

$$\{d_{u,i}\} \rightarrow \mathbf{y}_i, \tag{4.2}$$

for each app i w.r.t different user u. Written in a matrix-format, all app-level latent security labels denoted as $\mathbf{Y} \in \Re^{n \times r}$, where n is the number of apps, and r is the number of security labels, and thus $\mathbf{Y} = [\mathbf{y}_1; \mathbf{y}_2; \cdots; \mathbf{y}_n]$.

In practice, user comments could come from various users with different preferences and backgrounds. Sometimes, user comments could even be random and noisy. Thus, it is not realistic to assume that all $\{d_{u,i}^{\ell}\}$ values are accurate enough with high quality. However, this is naturally a suitable situation for applying crowdsourcing algorithm, which assumes the quality of user annotations are not high, and the latent true label (y_i^{ℓ}) could be learned from crowds ($\{d_{u,i}^{\ell}\}$).

Step 2: Learning to rank to estimate the risks

By considering linear ranking model with projection $\mathbf{w} \in \Re^r$, the security risk score is computed by projecting the security annotated features $\mathbf{y}_i$ using projection $\mathbf{w}$, *i.e.*, $\langle \mathbf{y}_i, \mathbf{w} \rangle$, which is viewed as the final ranking score for app i. By choosing a learning to rank loss function, the task of step 2 is to find a suitable linear ranking model as:

$$\min_{\mathbf{w}} f(\mathbf{w}; \mathbf{Y}, \mathbf{S}) + \Omega(\mathbf{w}) \tag{4.3}$$

where $f(\mathbf{w}; \mathbf{Y}, \mathbf{S})$ is the loss function involved with projection $\mathbf{w}$, annotated labels $\mathbf{Y}$ and risk score $\mathbf{S}$, $\Omega(\mathbf{w})$ is the regularization term to avoid over-fitting. Actually, $f(\mathbf{w}; \mathbf{Y}, \mathbf{S})$ has many choices, and it could be point-wise, pair-wise or list-wise ranking function; it could use hinge loss, logistic loss, *etc*. Note that the latent security labels $\mathbf{y}_i$ are used as features in this step.

Note Ranking-Support Vector Machine (R-SVM) [55] is one of the state-of-the-art *learning to rank* methods. We apply R-SVM model to solve the mobile app risk ranking problem. In R-SVM, it optimizes:

$$\min_{\mathbf{w} \in \Re^r} \frac{1}{2} \|\mathbf{w}\|_2^2 + \frac{C}{2} \|(\mathbf{e} - \mathbf{B}\mathbf{Y}_{\ell}\mathbf{w})_+\|_2^2, \tag{4.4}$$

where $\mathbf{Y}_{\ell} = \{\mathbf{y}_1, \cdots, \mathbf{y}_{n_{\ell}}\}$ is the app labels obtained from training set *in the first step*, $\mathbf{w}$ is a linear transformation which projects the features $\mathbf{y}_i$ w.r.t app i to its corresponding risk score s_i, *i.e.*, $s_i = \mathbf{y}_i^T \mathbf{w}$, $\quad i \in \{1, \cdots, n_{\ell}\}$, $\mathbf{e} \in \Re^p$ is a vector with all ones, $\mathbf{B} \in \Re^{p \times n_{\ell}}$ is a

sparse matrix which encodes the pairwise constraints, i.e., each row of $\mathbf{B}$ contains exactly one $+1$ and one -1 with the rest entries 0. $(\mathbf{x})_+ = \max(\mathbf{x}, \mathbf{0})$, and replaces any element of $\mathbf{x}$ that is less than 0 with 0, thus term $\|(\mathbf{e} - \mathbf{B}\mathbf{Y}_\ell \mathbf{w})_+\|_2^2$ is the hinge loss function. C is the trade-off parameter which balances the loss function part and the model complexity part.

4.4 Mathematical formulation

4.4.1 Preliminary

We introduce the baseline methods that can be used to solve crowdsourcing problem in step 1. For clarity purpose, we use the same notations as in our model.

a. Crowdsourcing with Majority Vote (CMV)

Majority Vote [49] treats every user equally by averaging over the label of each user: $y_i^\ell = \frac{1}{m}\sum_{j=1}^m d_{j,i}^\ell$. By thresholding y_i^ℓ, CMV provides binary crowdsourcing result. However, for our problem, $\mathbf{Y}$ is used as features, hence the original y_i^ℓ is preserved without thresholding. One major problem with CMV is that it treats each user equally hence the contribution of experts would be overwhelmed by crowds' less valuable opinions.

b. Crowdsourcing via Two-coin for User (CTU)

The two-coin model [49] is used to model user behavior, and pays more credit on more trust-worthy users. It assumes two coins for each user. A user would give positive label to an item under the condition that the true label is positive in probability α (sensitivity), and he/she would give negative label to an item under the condition that the true label is negative in probability β (specificity), *i.e.*,

$$\alpha_u^\ell = \Pr(d_{u,i}^\ell = 1 | y_i^\ell = 1), \forall i \in \{1, \cdots, n\} \tag{4.5}$$

$$\beta_u^\ell = \Pr(d_{u,i}^\ell = 0 | y_i^\ell = 0), \forall \ell \in \{1, \cdots, r\} \tag{4.6}$$

Parameter $\alpha = [\alpha_u^\ell], \beta = [\beta_u^\ell]$ can be learned from training data using EM algorithm according to Maximum Likelihood Estimation (MLE) and y_i^ℓ is then computed via Bayesian rules.

4.4.2 Proposed PCMC Model

The basic idea of the proposed PCMC is to amalgamate CTU and R-SVM together, by taking advantages of both approaches. A straight forward objective function would be a combination of the two as

$$J(\theta, \mathbf{Y}_\ell, \mathbf{w}) = \min_{\theta, \mathbf{Y}_\ell, \mathbf{w}} \Big[- \ln \Pr(\mathcal{D}_{n_l}|\theta, \mathbf{Y}_l) - \ln \Pr(\theta) \tag{4.7}$$

$$+ \frac{\lambda}{2}\|\mathbf{w}\|_2^2 + \frac{C}{2}\|(\mathbf{e} - \mathbf{B}\mathbf{Y}_\ell \mathbf{w})_+\|_2^2 \Big]. \tag{4.8}$$

It consists two parts, negative log-likelihood of observations (Eq.(4.7)) using two-coin crowsourcing model, and ranking function (Eq.(4.8)) using standard rank-SVM model.

In Eq.(4.7)[1], $\theta = \{\alpha, \beta\}$ is the parameters as in Eqs.(4.5, 4.6), where α denotes the probability that a user will give positive label to an item under the condition that the label is positive, and β denotes the probability that a user will give negative label to an item under the condition that the label is negative. $\ln \Pr(\mathcal{D}_{n_l}|\theta, \mathbf{Y}_\ell)$ is the log-likelihood of observation $\mathcal{D}_{n_l} = [d_{u,i}]$ (Eq.(4.2)) from different users given the model parameter α, β and annotated app-security level $\mathbf{Y}_\ell$, where

$$\begin{aligned} &\Pr(\mathcal{D}_{n_l}|\theta, \mathbf{Y}_\ell) \\ &= \prod_{i=1}^{n_\ell} \prod_{l=1}^{r} \Big[\Pr(y_i^\ell = 1|\theta) \prod_{u=1}^{m} \Pr(d_{u,i}^\ell | y_i^\ell = 1; \theta) \\ &+ \Pr(y_i^\ell = 0|\theta) \prod_{u=1}^{m} \Pr(d_{u,i}^\ell | y_i^\ell = 0; \theta) \Big]; \end{aligned} \tag{4.9}$$

$$\Pr(d_{u,i}^\ell | y_i^\ell = 1; \theta) = (\alpha_u^\ell)^{d_{u,i}^\ell} (1 - \alpha_u^\ell)^{1 - d_{u,i}^\ell} \tag{4.10}$$

$$\Pr(d_{u,i}^\ell | y_i^\ell = 0; \theta) = (\beta_u^\ell)^{1 - d_{u,i}^\ell} (1 - \beta_u^\ell)^{d_{u,i}^\ell}, \tag{4.11}$$

Let

$$p_i^\ell := \Pr(y_i^\ell = 1|\theta), \quad \mathbf{P} := [p_i^\ell] \tag{4.12}$$

[1] $\mathbf{Y}_\ell, \mathbf{P}_\ell$ refer to the labeled apps, *i.e.*, $i = \{1, 2, \cdots, n_l\}$.

then p_i^ℓ is the probability of label ℓ on app i, which we call "*better features*" as compared to y_i^ℓ, and can be optimized in our method. Clearly,

$$\Pr(y_i^\ell = 0|\theta) = 1 - p_i^\ell. \tag{4.13}$$

$\Pr(\theta)$ is the prior probability for parameters α and β, since α, β represent the probability of a binary event, thus a natural choice is the Beta distribution. *i.e.,*

$$\Pr(\alpha|a_1, a_2) = \text{Beta}(\alpha|a_1, a_2); \tag{4.14}$$

$$\Pr(\beta|b_1, b_2) = \text{Beta}(\beta|b_1, b_2); \tag{4.15}$$

where a_1, a_2, b_1, b_2 are the parameters for Beta distribution.

In Eq.(4.8), $\mathbf{Y}_\ell$ is the estimated annotated app labels in training set from raw comments as in Eq.(4.2), $\mathbf{w}$ is the projection for ranking, $\mathbf{B}$ is the pair-wise constraint, $\mathbf{e}$ is a vector with all ones, which are defined as in Eq.(4.4) in standard R-SVM model, and λ a trade-off parameter.

4.4.3 Alternating Optimization

The optimization of two-coin crowdsourcing model is a Maximum-A-Posteriori (MAP) problem, whereas R-SVM model is a Quadratic Programing (QP) problem. The optimization goals for MAP and QP are divergent. Moreover, $\mathbf{Y}_l$ is output of the MAP but input as feature of the QP, making both feature and model unknown for QP. Therefore we propose an alternating optimization approach to approximate the optimum solution instead.

Alternating optimization [61] is an iterative procedure for minimizing each variable individually while fixing the other variables. Alternating optimization has been well studied, and used in a wide variety of areas. Generally its convergence can be guaranteed. The alternating optimization process for PCMC takes four sub-procedure in each iteration as follows:

(1) $\mathbf{Y}_\ell \rightarrow \theta$, estimate model parameter α, β based on current estimation of app labels.

(2) $\mathbf{Y}_\ell \rightarrow \mathbf{w}$, estimate app ranking $\mathbf{w}$, based on current app labels (as features).

(3) $(\mathbf{w}, \mathbf{Y}_\ell) \rightarrow \mathbf{P}_\ell$, approximate better features $\mathbf{P}_\ell$ based on current ranking model $\mathbf{w}$ and app labels $\mathbf{Y}_\ell$.

(4) $(\theta, \mathbf{P}_\ell) \rightarrow \mathbf{Y}_l$, update app label estimations based on user model and $\mathbf{P}_\ell$ as a prior.

The above four sub-procedures are iterated until the algorithm converges. Now we investigate each of these sub-procedures respectively.

(1) $\mathbf{Y}_\ell \rightarrow \theta$: Given $\mathbf{Y}_\ell$, model parameter θ can be computed as those in standard two-coin crowdsourcing model according to MAP estimation, where

$$\alpha_u^\ell = \frac{\sum_{i=1}^{n_\ell} y_i^\ell d_{u,i}^\ell + a_1 - 1}{\sum_{i=1}^{n_\ell} y_i^\ell + a_1 + a_2 - 2} \tag{4.16}$$

$$\beta_u^\ell = \frac{\sum_{i=1}^{n_\ell} (1 - y_i^\ell)(1 - d_{u,i}^\ell) + b_1 - 1}{\sum_{i=1}^{n_\ell} (1 - y_i^\ell) + b_1 + b_2 - 2}, \tag{4.17}$$

where prior α and β follow Beta distributions, as defined in Eqs.(4.5, 4.6) and Eqs.(4.14, 4.15) with parameters a_1, a_2 and b_1, b_2, respectively.

(2) $\mathbf{Y}_\ell \rightarrow \mathbf{w}$: This is the same as solving the R-SVM problem. The update on $\mathbf{w}$ with Eq.(4.4) in the standard form of ranking SVM, hence can be solved as in [55]. The ℓ_2 norm of $\mathbf{w}$ in Eq.(4.4) can be viewed as a prior to $\mathbf{w}$ in the form of a standard normal distribution.

(3) $(\mathbf{w}, \mathbf{Y}_\ell) \rightarrow \mathbf{P}_\ell$: In order to update $\mathbf{Y}$ based on the newly updated θ and $\mathbf{w}$, an approximation of a better feature representation of apps are generated based on current ranking model $\mathbf{w}$ and app labels $\mathbf{Y}$. The objective function for this approximation is

$$\min_{\mathbf{P}} J(\mathbf{P}) = \min_{\mathbf{P}} \frac{C'}{2} \|(\mathbf{e} - \mathbf{BPw})_+\|_2^2 + \frac{1}{2} ||\mathbf{P} - \mathbf{Y}_\ell||_F^2 \tag{4.18}$$

$$s.t. \;\; \mathbf{P} = [p_i^\ell], \;\; 0 \leq p_i^\ell \leq 1 \tag{4.19}$$

The first term in Eq.(4.18) indicates that $\mathbf{P}$ should fit the ranking model $\mathbf{w}$ with constraints $\mathbf{B}$ well, while the second term regularizes it to be an approximation of $\mathbf{Y}_\ell$. Where C' works

as a trade-off parameter. This is a non-convex optimization problem with box constraints, with the first derivative matrix given by:

$$\frac{\partial J(\mathbf{P})}{\partial \mathbf{P}} = -C'\mathbf{B}^T((\mathbf{e} - \mathbf{BPw})_+)\mathbf{w}^T + \mathbf{P} - \mathbf{Y}_\ell \tag{4.20}$$

Due to the non-convexity of $J(\mathbf{P})$, only the local optimal is available for Eq.(4.18). But since $\mathbf{P}$ is only used to get a better feature representation based on the ranking model around true $\mathbf{Y}_\ell$, a local optimal around $\mathbf{Y}_\ell$ is sufficient. Hence $\mathbf{P}$ is obtained as the local optimal of Eq.(4.18) with $\mathbf{Y}_\ell$ as the initial point. Any gradient based optimization methods which accept box-constraints, *e.g.,* interior-point method, can be adopted here for the actual computation.

(4) $(\theta, \mathbf{P}) \rightarrow \mathbf{Y}_\ell$: Firstly, we need to compute two likelihoods:

$$a_i^\ell = \prod_{u=1}^{m} (\alpha_u^\ell)^{d_{u,i}^\ell} (1 - \alpha_u^\ell)^{1 - d_{u,i}^\ell} \tag{4.21}$$

$$b_i^\ell = \prod_{u=1}^{m} (\beta_u^\ell)^{1 - d_{u,i}^\ell} (1 - \beta_u^\ell)^{d_{u,i}^\ell} \tag{4.22}$$

where a_i^ℓ is the likelihood of app i getting label ℓ, b_i^ℓ is the likelihood of app i not getting label ℓ. Then by Bayesian rule with prior p_i^ℓ from $\mathbf{P}$, y_i^ℓ is computed as

$$y_i^\ell = \frac{a_i^\ell p_i^\ell}{a_i^\ell p_i^\ell + b_i^\ell (1 - p_i^\ell)}. \tag{4.23}$$

4.4.4 Discussions

Initial value The alternating optimization for PCMC requires initial values of $\mathbf{Y}_\ell$, which can be obtained from either CMV or CTU.

Feature Augmentation A possible extension to PCMC is to use feature augmentation on the security feature space (the label space). There are two reasons for this suggestion. For one reason, the primal form of SVM is adopted in Eq.(4.4), hence the kernel method is not directly applicable here. Therefore by utilizing feature augmentation, the model

may gain nonlinear modeling ability over the original feature space. The other reason comes from the multi-label setting. By using feature augmentation on the label space, the linear ranking model can now capture correlations between labels. For example, if order-2 polynomial feature augmentation is applied, co-occurrence of label pairs is captured in the new feature space, and can be utilized by the ranking model to make use of the label correlation.

Parameter Tuning Two parameters need to be tuned for the PCMC model. Both are trade-off parameters. They are C' from Eq.(4.18) and C from Eq.(4.4). Grid search can be used in cross validation [2] in training set for finding the best C and C'. To avoid tuning both parameter together, one may apply CTU and R-SVM separately, and run cross validation to find C in the R-SVM model. Then we can apply this C directly in PCMC and tune only C'. It is worth noting that, in the original object function Eq.(4.8), there is a trade-off parameter λ which no longer exists when applying the iterative optimization, since its role has been taken place by C'.

4.4.5 Applying the Model to Testing Data

Given $\theta, \mathbf{w}$ as the trained model, and the test data $\mathbf{D}_{n_u}$. First apply Eq.(4.23) to get $\mathbf{Y}_u$, where the prior $p_i^\ell(i > n_\ell)$ can be computed as

$$p_i^\ell = \frac{1}{n_\ell} \sum_{j=1}^{n_l} p_j^\ell.$$

Then apply $s_i = \langle \mathbf{y}_i, \mathbf{w} \rangle (i > n_l)$ to get the risk score for each app in testing data set.

[2] http://en.wikipedia.org/wiki/Cross-validation_(statistics)

4.5 Experiments

4.5.1 Datasets

We evaluate the proposed PCMC method on two datasets. The D1 dataset is collected from `Google Play` during May 2014, which contains $6,526$ apps. The D2 dataset is collected also from `Google Play` during December 2013, which contains $6,257$ apps. There are no overlaps for the above two datasets, since many new apps have been updated on `Google play`. The details of number of comments and users are shown in Table 4.1.

Table 4.1.: Datasets details. The Mean, Max and Min are statistics for the number of comments per app.

Data	#app	#comment	#user	mean	max	min
D1	6,526	6,021,244	2,614,186	923	4,020	1
D2	6,257	5,108,539	1,496,554	816	4,500	2

4.5.2 Label System

Motivated by our previous research [60], we use the same label system and labeled dataset to train the classifiers as auto annotation tools. This label system contains 11 labels with about $4k$ labeled user comments. These 11 labels consist of a general label for security/privacy related issues and 10 finer labels divided into two groups. One group is about the *nature* of the issues (system, privacy, spam, finance and others) and the other group is about the *scenario* of the issues (before install, executing, foreground, background and after uninstall). Logistic Regression (LR) [41] is used as the classifier and basic Bag-Of-Word (BOW)[3] feature is used as comment feature. Feature augmentation is adopted to leverage the correlation between labels and the final label set consists of the original labels and each pair of them ($\mathbf{C}_{11}^2 = 55$), making a $r = (11 + 55) = \mathbf{66}$ dimensional label set.

[3] `http://en.wikipedia.org/wiki/Bag-of-words_model`

4.5.3 Methods in Comparison

In order to evaluate the performance of the proposed PCMC method, we enlist four other methods into comparison. These four methods come from combinations of two crowdsourcing methods and two learning to rank methods in a separated two-step manner.

The two crowdsourcing methods are CMV and CTU, while the two learning to rank methods are Support Vector Regression (SVR) [62] and R-SVM [55] which are state-of-the-art ranking methods. The CTU method here uses a fix prior p_i^ℓ for each label while the PCMC uses $\mathbf{P}$. We also added a method that ranks the app by its user ratings only to provide a baseline. Therefore the six methods in comparison are:

(1-4) CMV+R-SVM, CTU+R-SVM, CMV+SVR, CTU+SVR: These four methods train crowdsourcing model (CMV or CTU) and learning to rank model (R-SVM or SVR) separately. **(5)** Our method: Use PCMC to train CTU model and R-SVM model jointly.

(6) Ranking by Ratings.

Note methods mentioned in [44], [1] are not used for app risk ranking, thus we cannot compare against them in our experiment settings. For ranking SVM, we apply the code from Olivier Chapelle[4], and LIBSVM [63] for SVR.

4.5.4 Evaluation Metrics

Since the task is defined as a ranking problem, we adopt Discounted Cumulative Gain (DCG) and Normalized DCG (nDCG)[5] as the evaluation metrics. Both DCG and nDCG are computed with respect to a given position K at the rank list. They also require relevant score of each app as ground truth. DCG is computed as $\mathbf{DCG}@K = \sum_{i=1}^{K} \frac{\text{rel}_i}{\log_2(i+1)}$. Where rel_i is the risk score of the app at rank i. And nDCG is computed by normalizing DCG with the DCG score of an ideal rank list at position K: $\mathbf{nDCG}@K = \frac{\mathbf{DCG}@K}{\mathbf{IDCG}@K}$,

[4] http://olivier.chapelle.cc/primal/
[5] http://en.wikipedia.org/wiki/Discounted_cumulative_gain

where $\mathbf{IDCG@}K$ is the DCG score of the ideal rank list at position K. And larger values of $\mathbf{DCG@}K$ and $\mathbf{nDCG@}K$ indicate better performance.

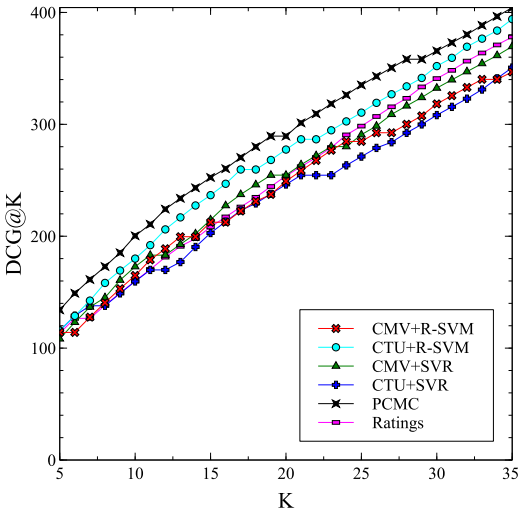
In the following experiments, the rel_i is provided by Android Guard[6], an open source software which can evaluate the privacy risk of an app by its permission and code analysis. And higher rel_i means higher risk. These rel_i scores are used in $\mathbf{nDCG}$ for evaluation in testing data, and also used for generating the pair-wise constraint matrix $\mathbf{B}$ for R-SVM and PCMC in training. Also, they are used as target values in regression for SVR. It is worth noting that, although the results from Android Guard are used in the experiment as ground truth, the purpose of this work is not to achieve the functionality of Android Guard or other tools. First of all, Andorid Guard uses the application binaries as input which is a marjor difference from this work. Secondly, the choice of Android Guard is due to the novelty of the proposed problem and lack of real risk evaluation from the user perspective. The proposed methed may be adapted for new label system and risk scores as well.

We pick four position $5, 10, 15$ and 30 to make point-wise comparison and also show the trend of the metrics in the range of 5 to 30. These points are picked for the following reasons: First of all, although the nDCG value will converge to 1 for any ranking function, it has consistent distinguish ability for different ranking functions [64]. Therefore we only need to evaluate the top rank list for comparison. Secondly, when nDCG is used in pure ranking task (*e.g.*, retrieval), it is popular to use $\mathbf{nDCG@}3, @5, @10$. However, the risk assessment is not a pure ranking task, we do hope to evaluation longer in the rank list, hence K is extended to 30 and the trend of metrics on the range of K will also be shown.

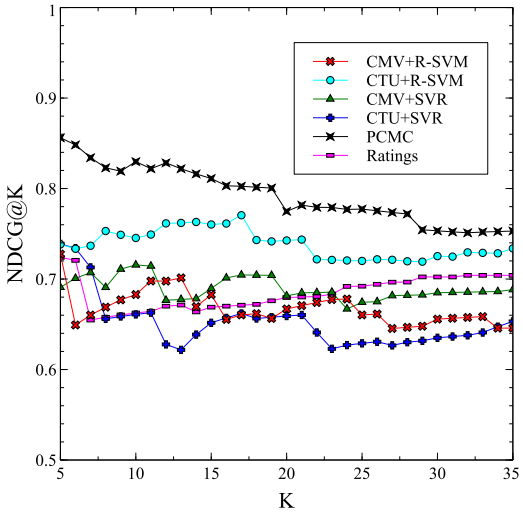
4.5.5 Results & Discussion

Table 4.2 show DCG and nDCG values at position $5, 10, 15$ and 30 of the 5 methods in the two datasets. We make several important observations from the above experiment results.

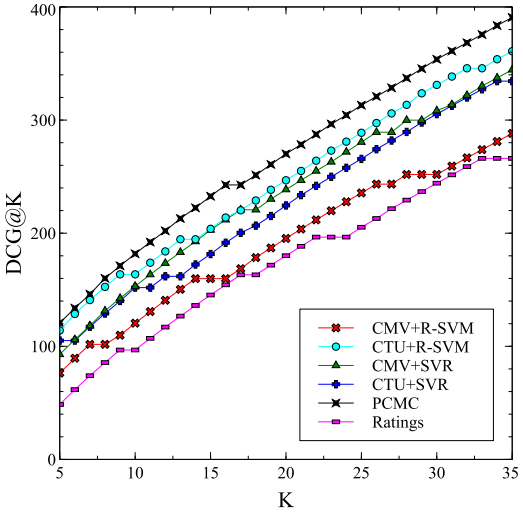
[6] https://code.google.com/p/androguard/



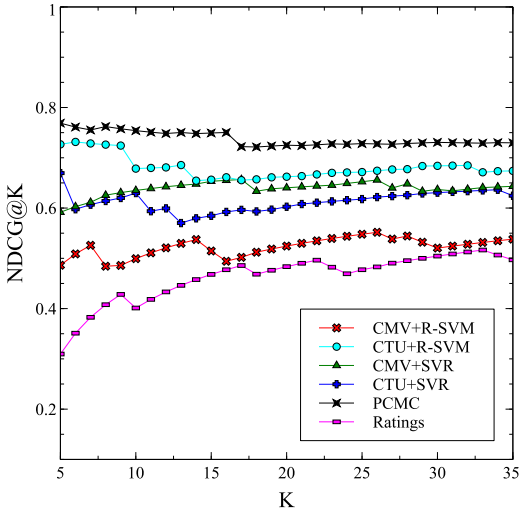
(a) Performance evaluation in D1 dataset with DCG.



(b) Performance evaluation in D1 dataset with nDCG.



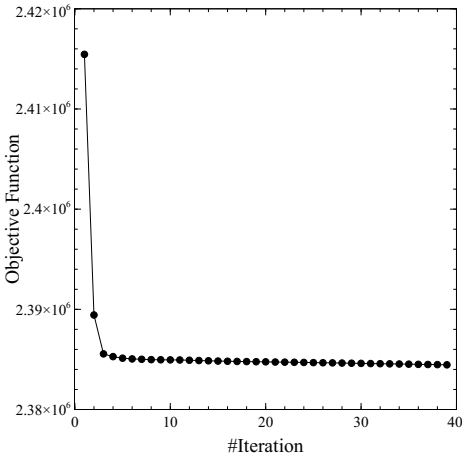
(c) Performance evaluation in D2 dataset with DCG.



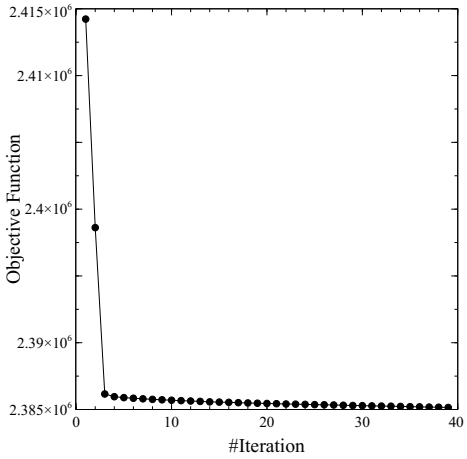
(d) Performance evaluation in D2 dataset with nDCG.

Figure 4.2.: Methods comparison based on DCG and nDCG.

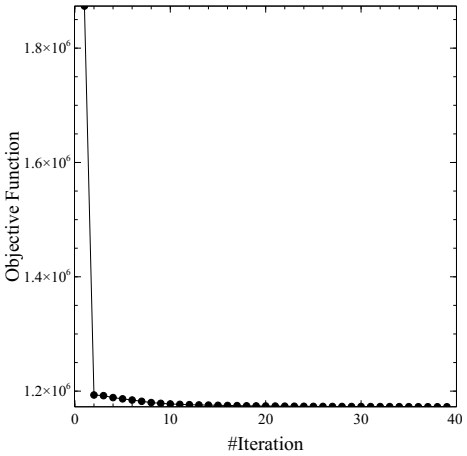
On D1 datasets, PCMC outperformed the other rivals 7.02% (9.46% based on the second best) in terms nDCG, and 17.06% (7.71% based on the second best) in terms of DCG by making an average of all cases.



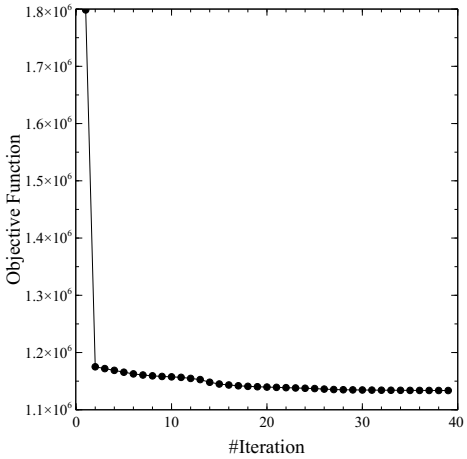
(a) $\log(C) = -10, \log(C') = 9$, D1 dataset



(b) $\log(C) = -11, \log(C') = 9$, D1 dataset



(c) $\log(C) = -14, \log(C') = 20$, D2 dataset



(d) $\log(C) = -13, \log(C') = 20$, D2 dataset

Figure 4.3.: Convergence curve in D1 and D2 dataset.

On D2 datasets, PCMC outperformed the other rivals 6.83% (10.03% based on the second best) in terms nDCG, and 19.07% (9.39% based on the second best) in terms of DCG by making an average of all cases.

CTU+R-SVM performs the second best. The major improvement of of the proposed-with their corresponding user comments PCMC over CTU+R-SVM comes from jointly training of CTU and R-SVM together. When CTU and R-SVM are trained separately, there is no guidance for CTU to distinguish different labels (latent features), and the latent

features are generated independently. On the other side, by jointly training the two models, the ranking model makes feedback through Eq.(4.18) to $\mathbf{P}$, and $\mathbf{P}$ to the CTU model through Eq.(4.23). Notice that, Eq.(4.18) involves all latent feature dimensions jointly with respect to $\mathbf{w}$, which is the weight of each dimension in ranking model. Therefore, by utilizing $\mathbf{P}$ as prior, the CTU model is not estimating the user expertise (α, β) for each label independently any more, hence the training of CTU model is guided indirectly by the pair-wise constraint of $\mathbf{B}$ in Eq.(4.4), and is able to provide a better ranking model.

For a finer comparison of the five methods in the two datasets, Figure 4.2(a), 4.2(b), 4.2(c) and 4.2(d) illustrate how DCG and nDCG change with the evaluation position K. PCMC is observed to maintain an obvious advantage over its rivals in the selected range of K. This advantage will decrease to 0 as K increases and the value of nDCG converges to 1.

Convergence of PCMC The convergence of PCMC is basically guaranteed by the alternating optimization algorithm. Figure 4.3(a), 4.3(b) and 4.3(c),4.3(d) show the convergence curve on both datasets with different parameter settings. Notice that, the big different between the suitable value of C and C' comes from the difference of the regularization terms in Eq.(4.4) and Eq.(4.18). $\|\mathbf{w}\|_2^2$ is the sum of squares of $r = 66$ elements, while the $\|\mathbf{P} - \mathbf{Y}_\ell\|_F^2$ is the sum of squares of nr elements. Hence the proper C' is much larger in order to balance between the hinge loss function and $\|\mathbf{P} - \mathbf{Y}_\ell\|_F^2$ term.

4.6 Limitations

As a method that targets security issues, it would not be appropriated not to mentions the limitations. This will provide a clearer view of the potential of the proposed method and provide interesting future directions.

Security topic extraction There is still room for improvement for extracting security topic from user comments. We are using bag of word model which ignores sentence structure and certain natural language precessing technique would be helpful to get more accurate extraction.

Security risk score Although we can evaluate and compare the methods based on security risk score, it is not an objective measurement which can be agreed on from different perspectives. Some novel metric may be needed for the measurement of mobile app security risk.

Comment spamming The proposed method estimates user credibility based on the assumption that the overall opinion from all user comments should provide relative reliable opinions to an app. However, in the situation of comment spamming, besides the credibility of users' comments, the integrity of the users themselves are questionable, the proposed method will not be able to estimate use credibility as designed, since it is possible that a large portion of the comments for an app are bias. One possible direction for solving this would be taking the meta data of the comments and the users into consideration in order to find out the patterns of spamming. For example, whether large number of comments with similar contents are posted on exactly the same time, or by users with similar user names, or by users register (or downloaded the app) at the same time, etc. This will require support from the market since much of the meta data is not publicly available. Previous works for email anti-spam and Tweet anti-spam would provide good examples for further exploration in this direction.

4.7 Conclusion

In this paper, we propose to utilize user comments to assess security risk of mobile apps. This risk assessment problem is formalized as a two-step task with crowdsourcing to learn the latent true security labels of apps, and learning to rank to provide risk scores based on the true security labels. This is to the best of our knowledge the first work to make mobile app security risk assessment in crowdsourcing and ranking perspective. By jointly training both crowdsourcing and ranking model, the crowdsourcing model is guided by the pair-wise constraint of ranking model and is able to estimate user expertise with the awareness of importance and correlation of multi-label, therefore providing better performance.

Experiment on two real-world datasets both with over $6k$ apps show substantial advantage of the propose PCMC method over state-of-the-art methods. Moreover, by treating user comments as crowd labels, this framework can be easily extended to other assessment task that involves user comments or feedbacks.

Table 4.2.: DCG & nDCG comparisons on D1 & D2 datasets.

Dataset	Measurement	Ratings	CMV + R-SVM	CTU + R-SVM	CMV + SVR	CTU + SVR	PCMC (our method)
D1	DCG@5	113.46	114.04	115.72	108.29	115.80	**134.19**
	DCG@10	159.99	164.91	180.01	172.81	159.60	**200.38**
	DCG@15	208.25	212.52	236.70	214.72	202.88	**252.52**
	DCG@30	340.89	318.26	352.05	332.42	308.25	**365.62**
	nDCG@5	0.72388	0.7276	0.7383	0.6909	0.7388	**0.8561**
	nDCG@10	0.66247	0.6829	0.7454	0.7156	0.6609	**0.8298**
	nDCG@15	0.66889	0.6826	0.7603	0.6897	0.6517	**0.8111**
	nDCG@30	0.7023	0.6557	0.7253	0.6849	0.6351	**0.7533**
D2	DCG@5	61.716	76.30	114.03	92.84	105.05	**120.65**
	DCG@10	96.74	120.43	163.62	153.15	151.89	**181.83**
	DCG@15	145.33	159.80	203.83	202.84	181.54	**232.66**
	DCG@30	244.2	251.93	331.10	308.34	305.29	**353.71**
	nDCG@5	0.3096	0.4862	0.7267	0.5916	0.6695	**0.7689**
	nDCG@10	0.40114	0.4994	0.6785	0.6351	0.6298	**0.7540**
	nDCG@15	0.46803	0.5146	0.6564	0.6532	0.5846	**0.7493**
	nDCG@30	0.50445	0.5244	0.6624	0.6403	0.6026	**0.7250**

5 AUTORBF: AUTOMATICALLY UNDERSTANDING THE REVIEW-TO-BEHAVIOR FIDELITY FOR ANDROID APPS

5.1 Motivation

In previous chapters, we discussed how we can extract security related topics from user comments (Chapter 3) and how to use the topic labels to evaluate app security risks (Chapter 4). In this chapter, we focus on four common security issues and evaluate how our proposed method performs and more importantly, compare the results with code analysis methods to find out the gap between them. We will revisit both the CDCE method and PCMC method as we describe them as one integrated system as AutoRBF.

5.2 Introduction and Related Works

5.2.1 Introduction

Nowadays, people spent more time on using mobile apps on smart phones and tablets because of the convenience they bring to people's daily life. One of the reasons is that users are able to enhance the mobile devices' functionality via rich-featured third-party mobile apps, which can be easily obtained from Google Play store and App Store. The number of available mobile apps has increased dramatically in the past years. For example, in June 2014, App store had 1.2 million apps and a total number of 75 billion downloads [65].

Personalized service (such as targeted advertising) is possible on mobile devices when users' personal information such as contacts and user locations, is accessible by mobile apps. However, disclosing personal information to mobile apps could lead to serious privacy concerns. Unfortunately most users are innocent to the sensitive information (*e.g.*, device hardware information, user data, *etc*) that permission requests ask to access.

Mobile app have exposed severe security risks, and they require the users' private and personal information. The mobile app can perform security-sensitive operations on the mobile devices. The available metadata such as user reviews are helpful for analysis of security related behaviors. From user perspective, it reflects the users' perceptions and expectations to mobile apps.

On Google Play, user reviews are public to all users (*e.g.,* Fig.5.1). The user reviews describe how the users think about the app, and give an idea of the security related issues/behaviors during the running of apps, which we call "*review-to-behavior fidelity*". In this paper, the "behavior" we consider only refers to the security/privacy related behaviors, which may cause severe security and privacy concerns for end users. In more detail, we focus on four categories of security-related behaviors shown in Table. 5.1, which accounts for more than 90% of all users' complains. The reason why we view the user reviewers as an important source to understand mobile apps' behaviors is that whether an app has exposed severe security risk or not actually depends on how the users think about it.

With the belief that the user reviews and behaviors should be consistent, we present AUTORBF, a system that automatically identifies the reviews that reflect the security-related behaviors both at review-level and app-level. AUTORBF can be applied in the following scenarios.

- Users can automatically infer whether the app has security-related behaviors from other users' experiences and expectations.
- Mobile app developers are alert of users' complain by taking the feedback from users, and are aware of the security-related behaviors that the mobile app has displayed.
- The security-related behavior analysis can be helpful for risk assessment for mobile apps, which can be used to improve the credibility of app rating scores in app markets.

However, in practice, in order to build a system which can automatically solve the review-to-behavior fidelity problem, we need to solve the following challenges (C1-C3):

C1: Review semantics We note in user reviews, most of user reviews are short, properly with wrongly spelled words and even made-up words. Moreover, different words and expressions can be used for the same purpose. Therefore, it requires to gather enough semantics from users' review to understand the security-related behaviors.

C2: Security Concern Not all reviews reflect the security/privacy related concerns (Fig. 3.1). Recent study from the distribution of user ratings [35] shows that users tend to give positive feedback. Some reviews include vague and pure emotional comments, and some reviews may complain about the quality, the price of app, which has few relations with security and privacy concerns. Therefore, how to infer the security-related behaviors from the crowded user reviews is still very challenging.

C3: Credibility of users The user reviews can be noisy and diverse. User reviewers are coming from different users. Some users may not be responsible for their reviewers. Some reviewers are not likely to report the security problem. Although users' account are associated with Google account in Goolge Play, still lots of reviews are non-informative. How to distinguish different types of users, and make the app-level review results more reliable?

To the best of our knowledge, none of the previous works have systematically solve the above problems in the context of review-to-behavior fidelity. In this work, we design and implement a system called AUTORBF, which automatically infers the security related behaviors by considering the semantics of user reviews and aggregating the security related behaviors from comment-level to app-level via crowdsourcing. In our system, the state-of-the-art machine learning techniques are used to predict the security-related behaviors, and bridge the gap between the user reviews and behaviors.

The contribution of this paper is summarized as follows.

- To address the challenge of *Review semantics* (C1), at user review level, we extract the semantics from user reviews, and make a feature augmentation approach by expansion of features via exploring the "relevant feedback", in which the correlations and co-occurrence among different reviews are taken into account. In this way, the

Table 5.1.: Security/privacy-related behaviors

index	Category	Behavior Descriptions
1.	Spamming	Ads in notification bar, ads via email, ads via SMS, Pop-up ads, Fishing, *etc*
2.	Financial	Paid for In-App-Purchase (IAP), but do not get the item, free to premium, *etc*
3.	Over-claimed Permission	Request too much permissions than users' expectations
4.	Data leakage	Access privacy data without user acknowledgement, *e.g.*, user account, contact, location, *etc*

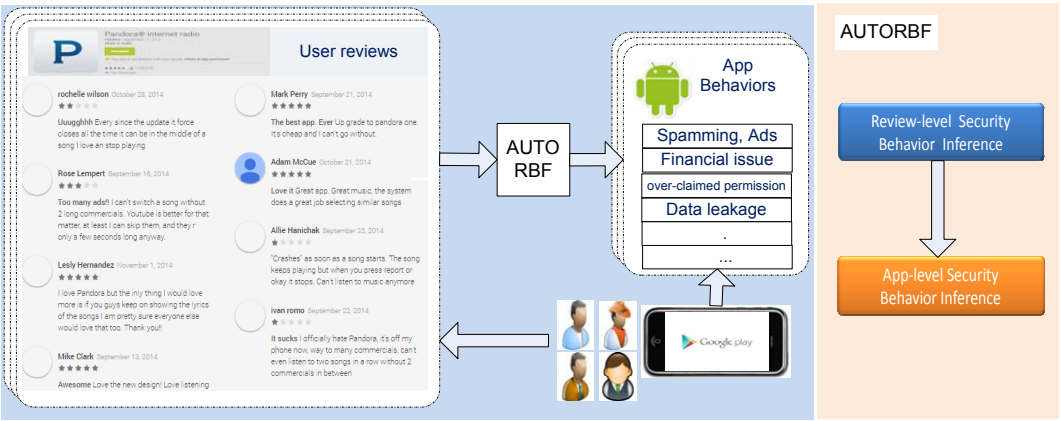


Figure 5.1.: Infer the security-related behaviors from users' reviews. Overview of the framework of AUTORBF : (a) Engine 1: Review-level security behavior inference engine; (b) Engine 2: app-level security behavior inference engine.

security-related behaviors are captured even if they are described in different phrases and expressions.

- To address the challenge of *Security concerns* (C2), we build a classifier which can automatically learn the four categories of security-related behaviors (Table 5.1) based

on the state-of-the-art sparse machine learning technique. As long as enough training user reviewers with labeled security-related behaviors are given, our system can automatically infer the behavior of mobile apps related to security concerns with accuracy as high as 94.05% at user-review level.

- To address the challenge of *user credibility* (C3), based on the extracted review-level semantic features, we aggregate the security behavior annotation result from review-level to app-level based on the expertise of different users. We do not put full confidence on all the users, and the wisdom of crowds are automatically learned via state-of-the-art crowdsourcing techniques.

We believe that AUTORBF system provides a generic and universal framework for analysis of user reviewers with enhanced semantic understanding of security-related behaviors. Also, this framework could be easily extended to analyze the security/privacy issues on the reviews appeared in other webistes, such as `Yelp, Amazon`, and `ebay`. Since the label system we use to infer the security-related behaviors is arbitrary, one may use the same user review data with different label system to evaluate the security/privay aspect of the other product.

We further have the following interesting discoveries:

- *Measurement* Our evaluation on $19,413$ reviews, $3,174$ apps demonstrates that AUTORBF can accurately predict the review-level security related behaviors with accuracy as high as 94.04%. As compared to the other approaches, our work excels a large margin with 51.36% in accuracy, as opposed to a baseline using key-word based approach, which further validates the effectiveness of system design and algorithm deployment.

- *Evaluation* We also make a case study to analyze the difference between behaviors inferred from AUTORBF and those from code analysis. Our findings include: on 67.5% apps, code analysis detected same spamming behavior result as that of AUTORBF; on 58.5% apps, code analysis discovered over-claimed permission and AUTORBF also found the over-claimed permissions than users' expectation; on 73.5%

apps, code analysis detected the "data leakage" issue while AUTORBF agreed. However, for "financial-related" issue, almost no code analysis can find the same security issue as AUTORBF. This suggests that user reviewer analysis from meta-data complements the works about the security-related behaviors analysis from code analysis, which provides an alternative way to look at the mobile app security concerns from user perspective. We believe that the combination of them in an appropriate way will definitely help to understand the app security issues better. Our empirical result about users' credibility suggest users have different preference in reporting the security issues.

The remainder of paper is organized as follows. Section 5.3 presents the problem statement. We introduce the detailed design of AUTORBF system in Section 5.4, followed by the design of review-level security behavior inference engine in Section 5.5. Section 5.6 presents the design of app-level security behavior inference engine. Section 5.7 presents the experimental evaluation of our system, followed by discussions of difference against code analysis in Section 5.8. Section 5.2.2 discusses the related work and finally Section 5.9 concludes the paper.

5.2.2 Related Works

Since the initial intrusion detection work done by Lee *et al.* [66], machine learning/data mining has been widely in a number of efforts to solve security problems, such as network failure analysis [67], worm signature generation [68], malware classification [11], software plagiarism detection [69], *etc*. Recently, machine learning has been used to understand the mobile app permissions and behaviors through analysis on the meta-data like description [44] [70], permission usage patterns of mobile apps [46], and contextual API dependency graphs [71], *etc*. In contrast to these works, our study focuses on analysis of mobile app behaviors from the users' reviews, a new angle of meta-data crawled from google play store, which aims to automatically infer the behaviors of mobile app at both review-level and app-level. Even if we treat the user review understanding problem as a

machine learning problem, the approach in this work is unique: we consider semantics of users' reviews when automatically infer the security behavior of mobile apps; moreover, rather than simply aggregating all the review-level security issues to the app-level, we use crowsourcing as means to capture different users' expectations for security resources from review-level to app-level. Experiment result tells us that not only we achieve high accuracy at review-level security behavior inference, but also we get clues for credibility of different users.

Most (if not all) existing works on analysis of mobile app risks from meta data focus on descriptions of mobile apps [44] [70], which are provided by developers. For example, in order to analyze the app's business aspects, Harman *et al.* [72] use a light-weight text analytic technique to mine the description, pricing of apps, *etc*. WHYPER [44] predicts the risk assessment of mobile apps based on analysis of the descriptions of apps from natural language processing perspective using first-order logic. AUTOCog [70] uses natural language technique and machine learning based approach to access the description-to-permission fidelity of mobile apps. Compared to description of mobile apps provided by developers, the user reviews studied in AUTORBF provide the real needs and expectations of users.

Recently, Lin et al. [73] introduce a new model for privacy by capturing users' expectations via crowdsoucing, where the users' expectations of mobile apps are captured from user study rather than the user reviews published at Google play store by millions of users across the world as in AUTORBF. We believe the user reviews automatically analyzed by AUTORBF can be used as good evidence for quantitative analysis of users' security behaviors. AR-Miner [34] is one of the few works that analyzes the user views from Andriod market, the goal of which, however, is to discover the most "informativee" user reviews rather than understanding the security behavior as in AUTORBF. WisCom [35] analyzes the inconsistency in user reviews but not for solving security issues as in AUTORBF. The other work about the user review analysis [60] can be viewed as an intermediate step in our comprehensive framework of AUTORBF, the capability of which has been greatly enhanced.

Notice that in the field of mobile app analysis, there exist many works on permission analysis [4] [15] [1] [74], code analysis [14] [7] [75] [76] and run-time behavior analysis [77] [78] [79] [80] with the purpose of detecting the app behaviors especially for mobile app malware analysis. For example, Enck *et al* [79] use dynamic taint analysis to analyze how users' private information is misused. While permission analysis, static analysis and dynamic analysis enable the detection of misused permissions, unauthorized actions, potential data leak, *etc*, these approaches, however, analyze the objective behavior of apps without giving enough consideration for the users' real expectations for different mobile apps. They did not bridge the gap between what the code really does and what the users' really need. In contrast, AUTORBF automatically analyzes the users' review and understand what the users' concerns for security issues, which can be used as a complementary tool for the other code-based approaches for improved user experiences and interaction with mobile apps.

5.3 Problem Statement

Android is the most popular smartphone operation system, and the availability of apps in Google play store surpassed 1 million apps in July 2013[1]. User reviewers are provided by different users after they use or install the apps, which offers the valuable feedbacks for new users and developers. From the security point of view, user reviewers indicate the users' expectations and concerns for different behaviors. User are known to be very concerned about the security-related behaviors on mobile apps [81]. As is sharply observed in Felt *et al.* [82], automatically understanding the security behaviors from meta data is not an easy job for most of the users.

Our goal in this paper is to predict the mobile app security-related behaviors from different users' reviews, which we call "*review-to-behavior*" fidelity. Fig.5.1 illustrates the overview framework to evaluate the security-related behaviors from user reviews.

[1] `http://en.wikipedia.org/wiki/Google_Play`

User reviews provide users' perception for mobile app behaviors, which we aim to use and identify the most pertinent information that correlates the security concerns. Different from the mobile app permission or behavior analysis through static or dynamic analysis ([79], [80], [83], *etc*), infer the security behaviors from the users' expectations is still lacking. The AUTORBF tool developed by us can be deployed as an individual app or deployed at Google play markets, which can automatically summarize and alert the users about the security-related behaviors based on users' experiences and expectations. The tool is also expected to help to improve the trustiness of the overall Android app market and make the Android echo-system better.

Several examples of user reviewers on Google play store are shown below.

- **Review 1**: *"Good App but keeps granting itself super user permissions randomly when its suppose to always have it."*
  User: A Google user

- ***Review 2***: *"Seemed ok, but the morning after signing in my account spammed everyone on my contact list with junk mail from me...could be a coincidence..."*
  User: J.S [1]

- ***Review 3***: *"Why does this need to send sms messages and make calls? Updates not that timely. Uninstalling."*
  User: J.D

After carefully examining the user reviews, we found that most of the user reviews are not complaining about the security-issue, not to mention security-related behaviors. The scope of this paper is to infer the "security-related" behaviors from the user reviews. More specifically, we focus on the four categories of security-related behaviors listed in Table. 5.1. The listed four categories of behaviors cover more than 90% of security issues appeared in mobile app markets, and viewed as critical security issues. For example, the *security behaviors* inferred from the above three reviews are shown below.

- **Review 1**: *Over-claimed permission*

[1] Due to privacy reason, we list user name using the first letters of family name and first name.

- ***Review 2***: Data leakage, background spamming
- **Review 3**: Data leakage

Note the defined four categories of security-behaviors reflect the users' understanding and perception on different mobile apps, in a very coarse-grained manner. Considering that most of the users are not professional developers, the user reviewers generally indicate the high-level understanding of privacy concerns from very vague, emotional reviews instead of very professional communication approach as in fine-grained permission control. Thus in the security-related behavior definition, we use a very coarse-grained behavior to cover each category of security issue, which corresponds to a group of fine-grained security-sensitive behaviors illustrated in Table 5.1.

As illustrated in Section 5.2.1, we need to address the three challenges in the design of our approach, which includes *review semantics, security concern*, and *user credibility*. We design and implement the system AUTORBF , which addresses the above challenges. The **design goal** of AUTORBF is to answer the following question: *given user reviews from different users on apps, can* AUTORBF *automatically infer the security-related behaviors of apps? And how accurate is it?* AUTORBF can help users to be aware of security related issues, and discover the correlations between user reviews and security issues. With the help of state-of-the-art sparse machine learning techniques, AUTORBF can achieve the accurate prediction of app behavior both at review level and app level. Besides, the training process works directly on user reviews, and the whole process can be automatically done without human intervention given enough training user reviews and security-related behaviors.

5.4 Overview of System Design

Fig. 5.1 illustrates the overview of system AUTORBF design, which includes two key engines: (a) review-level security behavior inference engine (RLI), and (b) app-level secu-

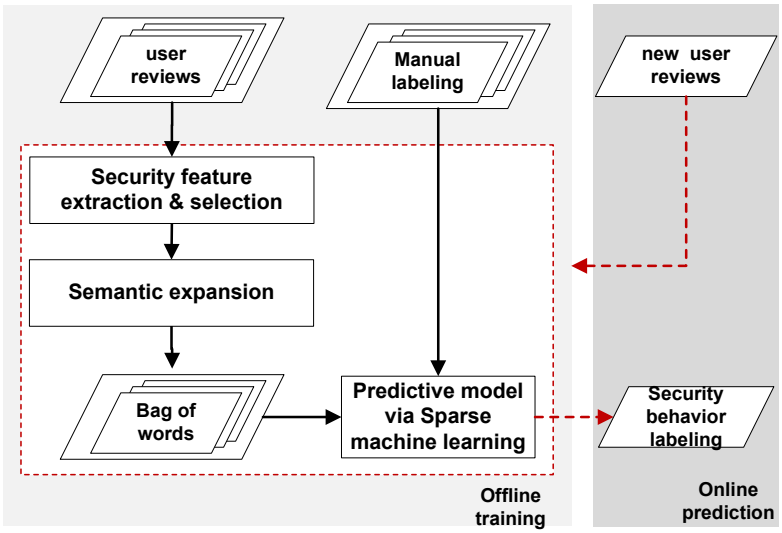


Figure 5.2.: The framework of review-level security behavior inference engine (solid lines are used for the training process, and dashed line for the process of annotating new user reviews).

rity behavior inference engine (ALI). Fig. 5.2 and Fig. 5.3 give the detailed introduction of each engine, respectively.

Given the users' reviews, by taking the advantage of the state-of-the-art machine learning techniques, the output of the review-level behavior inference engine, is review-level security behavior labeling results (*i.e.*, four categories of security behavior labeling shown in Table 5.1) corresponding to each review.

Then the review-level security behavior labeling are fed into the app-level behavior inference engine, with the help of the state-of-the-art crowdsoucing techniques, the output of the app-level security behavior inference engine is the app-level security behavior labeling results in consideration of users' credibility. Although it appears similar to the review-level security behavior labeling and also denoted as four categories of security issues shown in Table 5.1, it actually labels the behaviors of an app by aggregating all different reviewers' opinions via utilizing the wisdom of crowds. We will illustrate the detailed design of each engine in Section 5.5 and 5.6, respectively.

5.5 Review-level Security Behavior Inference Engine (RLI)

The goal of the review-level security behavior inference engine is to infer the review-level security behavior labels from the descriptions of reviews. Given enough training user reviews, the review-level inference engine (RLI) can automatically label the user review to a specific security behavior category.

The overview of RLI engine is depicted in Fig. 5.2, which consists of two key phases: training phases (denoted in solid line in Fig. 5.2) and testing phases (denoted in dash line in Fig.5.2). In training phase, state-of-the-art machine learning techniques are used to learn a model/classifier which can annotate the user reviews automatically, while in testing phase, new user reviews are fed into the classifier, and the security related behaviors are automatically labeled.

5.5.1 Training Phase vs. Testing Phase

The **training phase** includes the following three key steps.

Step 1: Security-related feature extraction and selection. To infer the security related behaviors, we first extract the features (including words and phrases) that have close relations with four categories of security concerns. This is to *address the challenge C2*, because it has been observed from the user review dataset that the complains about the functionality, quality and the attractiveness for mobile apps account for the majority of user review concerns. We use a key-word based approach to select the security related words and phrases appeared in user reviews. After Step 1, only security related features are preserved and selected, and will be used to build a classifier for labeling user reviews.

Step 2: Semantic Expansion. To *address the challenge C1*, this step concerns how to understand the user reviews described in different words and phrases but for the similar or the same meaning. By taking advantage of "relevance feedback" technique [42] in information retrieval, we find the relevant words/phrases related to security-related features, and expand the original review features by adding new "relevant" words and phrases. This process is iterated until all the "relevant" features are added to the review feature sets. This

process is also known as “feature augmentation”. We aim to capture the semantics of user reviews as much as possible by utilizing the relations among different user reviews. After this step, each user review is abstracted into a feature vector denoted as a bag of word (BOW).

Step 3: Training classifier using sparse machine learning. To *address the challenge C1*, once we have obtained the BOW features after semantic expansion, we train the classifier for prediction of each category of security-related behavior. Our framework is open to any classifier that, in order to classify a new user review, requires the BOW feature as the training samples. Such classifiers include the kNN classifier, SVM classifier, *etc*. In our approach, we use the state-of-the-art machine learning classifier (*i.e.*, sparse SVM) by exploring the structural sparsity of feature space, which has shown the state-of-the-art performance.

Testing phases The output of the training phase is an automated user review classifier. Given new user reviews, we first extract and select the word/phrase features related to security concerns, and then generate BOW features. Next, we feed the features to the trained sparse machine learning classifier and automatically determine the security behavior category.

Our framework does not require any human intervention. In a dynamic environment where we have new user reviews, we repeat the process by retraining the sparse machine learning classifier. In the next few sections, we shall elaborate on the technical details of each of the above steps.

5.5.2 Security-related Feature Extraction and Selection

In this section, we focus on the extraction of the security-related features. Proper perprocessing are applied to user reviews before feature extraction, including removing stop words and stemming. Since many user reviews are not related to security concerns of mobile apps, a necessary step we need to take is to narrow down the huge number of user reviews to a more feasible set for further analysis and annotation. To achieve this purpose,

a coarse-grained filtering is firstly applied to create a subset of suspect user reviews. We adopt a key-word based approach to preserve a set of the suspect user reviews, *i.e.*, any user review that contains at least one of the predefined key words will be put into the suspect set.

The key words are manually picked in an iterative way. The initial set of the key words are set to {*security*, *privacy*}. Then the new key words are selected from those that have high co-occurrence with the current key words. The co-occurrence of key words in user reviews are computed based on the user review corpus L (see Section 5.7.1) we collect on Google Play. If the co-occurrence of the word-pairs exceeds a large threshold and one of the word-pair in the suspect set, we add the other word into the suspect set. The rational behind this approach is that these key words reflect users' concerns for the security and privacy issues. This process is iterated until no more new key words are added into the suspect set.

To avoid mismatch, not only the synonyms are considered, other forms of the key-words (e.g., antonyms, mismatch words, etc) are also taken into account. For example, *insure* and *unsure* are added for key-word "*security*", *stole* and *stolen* are added for key-word "*steal*", *etc*. We call these words as "security-related" features which captures the semantics of the security behaviors for mobile apps. Note in this step, instead of using the traditional *black-box* feature selection methods such as F-statistics [84] mutual information [85], *etc*, that rely too much on the statistical property of the data, our method can be viewed as a *semantics-aware* feature selection approach, which identifies the security-related key words by considering the semantic meaning and correlations among different key-words in a white-box way.

5.5.3 Semantic Expansion

User reviews have some properties which have significant difference from the properly edited documents. User reviews are normally short, probably have wrongly spelled words (as mentioned in Section 5.5.2), or even made-up words (such as "*privasy, nonsecure*").

Moreover, different words or expressions may be used for the same meaning. These properties may harm the performance of classifier if the extracted key-words from Section 5.5.2 are directly used for features.

It seems very challenging to solve this problem. We get our idea from information retrieval domain. In information retrieval, the query submitted to the search engine could be short, misspelled and vary in the choice of the words, which is very similar to our situation. For the same motivation, a traditional technique called query expansion with pseudo relevant feedback [42] in information retrieval is borrowed to make review expansion. Originally, this technique uses the top document in the retrieval ranking list with respect to the original query as "*relevant*" documents, and these documents are further used to expand the original query, generating a new query resembling the "*relevant*" documents. This query expansion is reported to always have a positive effect on the retrieval performance [42].

A similar process is adopted for review semantics expansion as follows. The process consists of two steps: **(1)** find the "*relevant*" reviews; **(2)** augment the "*original*" reviews with "*relevant*" reviews. The relevance between the user reviews is evaluated using the similarity in the standard information retrieval model, *i.e.*, cosine similarity between the tf-idf features of the reviews.

An interesting question would be the scope of the retrieval. Should the candidate "*relevant*" reviews picked from all other reviews? Or just the reviews from the same app or the same category of apps? Besides, the "*relevant*" reviews should only be those posted before the reviews for expansion, so we are not using the "future" reviews to extend the current reviews. A sufficiently long ranking list from the retrieval engine will be returned and the scopes and restrictions will be applied to this list to pick the "*relevant*" reviews afterwards.

With a set of "*relevant*" reviews, the review expansion can be conducted by making a sum of the original reviews and the mean of the set of "*relevant*" reviews on feature level as follows: $\mathbf{f}_{\text{new}} = (1-\alpha)\mathbf{f}_{\text{old}} + \alpha\frac{1}{|R|}\sum_{\mathbf{f}_{\text{expand}}\in R}\mathbf{f}_{\text{expand}}$, where $\mathbf{f}_{\text{old}}$ is the Bag of Words (BOW) feature of the original reviews, $\mathbf{f}_{\text{new}}$ is the feature vector after the expansion of user reviews, $\mathbf{f}_{\text{expand}}$ is feature vector for the expanded user reviews, and R is the set of the "*relevant*" reviews with respect to the reviews for expansion. α serves as a tunable parameter for the

degree of effect of the expansion, and can be tuned for the best performance, and so is the size of R.

In summary, review expansion is an efficient way to explore the relationship among reviews, and incorporate more *semantics* into the review understanding process. The review expansion also relies on the retrieval engine, the cost the which is normally constant. Hence, the total cost of review semantic expansion is linear to the number of user reviews, which makes it more applicable in solving practical problems.

5.5.4 Sparse Machine Learning Classifier

After obtaining the BOW feature from the above steps, now we are ready to present the classifier which is designed to label the user review as certain security-related behavior defined in Table. 5.1. More formally, given the BOW feature and the annotated four categories of labels, the suspect user review datasets can be represented as $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_n]$, where $\mathbf{x}_i \in \Re^p$ denotes the BOW feature vector for each user review, n is the number of user reviews. Let $\mathbf{Y} = [\mathbf{Y}_{ik}](1 \leq i \leq n, 1 \leq k \leq K)$ be the label vectors for user reviews and $Y_{ik} \in \{0, 1\}$ with $Y_{ik} = 1$ indicating the presence of the j-th label for user comment i. In our problem setting, K = 4 denotes the number of security-related behavior labels defined in Table. 5.1.

In our problem setting, more than one security behavior categories can be simultaneously assigned to one user review (as shown in Section 5.3). This is known as multi-label learning in machine learning community. To solve this problem, we use state-of-the-art machine learning methods to solve this problem. Meanwhile, we hope our approach can work well on large-scale user review dataset. To satisfy this goal, we use the linear classifier, where the output classification decision is simply a weight vector that separates the data points in the high-dimensional space.

Since support vector machine (SVM) has shown the state-of-the-art performance in solving many real-world text mining classification problem, we use SVM with linear kernel as the base classifier for solving the multi-label learning problem. More specially, we

use the multi-class SVM with linear kernel to solve the behavior categorization problem. We use linear SVM also due to its simplicity, scalability and interpretability. For the exact same reason, SVM classifier has been widely used in large-scale malicious web-site detection [86] and drive-by-download attack detection [87], *etc*.

As in traditional SVM classifier, we use hing loss as the loss function, and enforce the structural sparsity on feature space with LASSO regularization [88]. Meanwhile we consider the label correlations among different behavior categories (*e.g.,* data leakage category, over-claimed permissions, *etc*). Given a set of n labeled data points $\{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^n$, the goal of training spare SVM is to find the projection matrix $\mathbf{W} \in \Re^{p\times K}$, that minimizes

$$\min_{\mathbf{W}} \sum_{i=1}^{n} (1 - \mathbf{w}_{y_i}^T \mathbf{x}_i + \max_{m \neq y_i} \mathbf{w}_m^T \mathbf{x}_i)_+ + \alpha\Omega_1(\mathbf{W}) + \alpha\Omega_2(\mathbf{W}), \tag{5.1}$$

and $\Omega_1(\mathbf{W}) = \sum_{m=1}^{k} \|\mathbf{w}_m\|_1$, $\Omega_2(\mathbf{W}) = \sum_{ij} (\mathbf{w}^i)^T (\mathbf{D} - \mathbf{S})_{ij} \mathbf{w}^j = \text{Tr}(\mathbf{W}^T \mathbf{L} \mathbf{W})$, where the first term is the standard hinge loss function, the second term $\Omega_1(\mathbf{W})$ is the structural sparsity term which enforces the structural sparsity, and for $i = 1, 2, \cdots, n$, $m = 1, 2, \cdots, k$, $\mathbf{W} = [\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_k]$, where $(x)_+ = \max\{x, 0\}$, and the third term $\Omega_2(\mathbf{W})$ is added to eliminate the label correlations for different categories.

The first term measures how data is fit into the model given the separating surface $\mathbf{W}$, which is directly minimizing the training error. The second term regularization is a penalty for projection $\mathbf{W}$ using $\ell_{1,1}$-norm especially when $\mathbf{W}$ is large. Similar to ℓ_1 regularization term, $\ell_{1,1}$ regularization tends to yield sparse weight for projection matrix $\mathbf{W}$, *i.e.,* a relatively few feature weights are set to zeros. Parameter α controls the trade-off between the loss function term and regularization terms (i.e., second and third terms): a large α value will give the second and third term of Eq.(5.1) more weight relative to the first term of Eq.(5.1), and hence yield a more sparse solution of $\mathbf{W}$. The third term $\Omega_2(\mathbf{W})$ is to eliminate the label correlations for different categories. One key observation is that some categories of security-related behaviors co-occurred frequently, thus the $\Omega_2(\mathbf{W})$ regularization term is used to capture the relatedness between different labels. More details are given in Appendix.

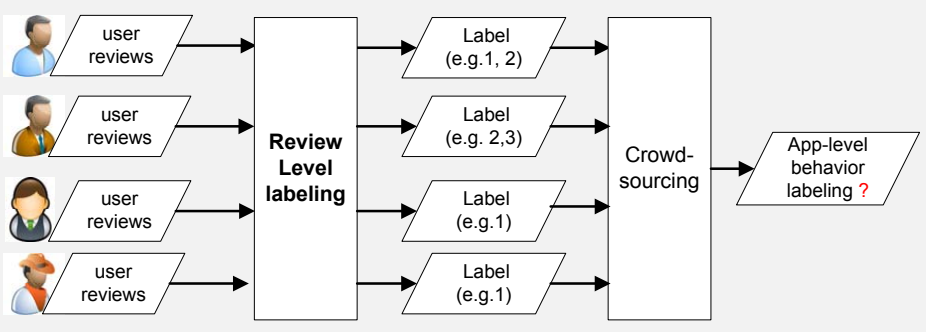


Figure 5.3.: The framework of app-level security behavior inference engine: infer the app-level labeling via crowdsourcing.

As solving the standard lasso problem, the iterative shrinkage method and L-BFGS quasi-Newton method [27] are applied for solving the optimization problem of Eq.(5.1). The convergence of the algorithm can be rigorously proved as that in [27].

5.6 App-Level Security Behavior Inference Engine (ALI)

The review-level security behavior inference model has annotated each review as 4 categories of security behaviors shown in Table. 5.1, which map the raw user review of each app to a list of security labels. For example, take the mobile app "Pandora" as an example,

user **R. L.**[1] : *ads.* ; user **N. O.**: *ads.* ; user **B. B**: *data leakage* ...

Different users have labeled the same apps in different labels. A natural question that follows is: *what the final label at app-level in consideration of all users?* We suggest using the crowdsourcing techniques to aggregate the security-label from the user review level to app-level. Crowdsourcing [49] is a technique to infer the true labels of a given item from the annotations of multiple annotations/works, where the annotations are assumed to be low quality and may contradict with each other. Although there may be substantial disagreement among different annotators, we aim to learn from "crowds" to annotate the app by considering different opinions. Note our task is not a traditional crowdsourcing prob-

[1]For privacy reason, we use the first letters of the first name and family name of the author.

lem, in which we model the mapping from user-reviews to app-level latent true labels as a crowdsourcing problem, treating each user review labels as labels from works/annotators.

More formally, solving the app-level annotation problem by different users can be described as follows in the context of crowdsourcing scenario. For each user u with respect to app i, its reviews $\mathbf{d}_{u,i} \in \{0,1\}^r$ is denoted as a r-dimensional set (r=4 in our case) corresponding to 4 categories of security-related behaviors shown in Table. 5.1, i.e., $\mathbf{d}_{u,i} = \{d^1_{u,i}, \cdots, d^r_{u,i}\}$, where r is the size of label set, and $d^\ell_{u,i}$ is the binary auto annotation result of the ℓth classifier for the review given by user u to app i, i.e.,

$$d^\ell_{u,i} = \begin{cases} 1; \text{if user } u \text{ labels app } i \text{ as label } \ell \\ 0; \text{otherwise} \end{cases} \tag{5.2}$$

Given all the above labels $d_{u,i}$ generated from different reviewers for each app i, we need to learn a mapping which automatically projects the review-level labels $\mathbf{d}_{u,i} = \{d^1_{u,i}, \cdots, d^r_{u,i}\}, (1 \leq u \leq m)$ to app-level security labels $\mathbf{y}_i = [y^1_i, y^2_i, \cdots, y^r_i]$, i.e., $\{d_{u,i}\} \rightarrow \mathbf{y}_i$, where each y^ℓ_i corresponds to one security behavior ℓ described in Table. 5.1 for app i.

5.6.1 Why Not Majority Voting?

One naive way is to treat all the users equally, *i.e.,* simply trust all the users and combine all the reviews' comments no matter they are trustful or not. For example, if most of the users say this app has "data-leakage" issue, then we say this app has "data-leakage" issue. This is known as majority voting [49] in crowdsourcing.

To treat every user equally by averaging over the label of each user, the final label for each app i will be: $y^\ell_i = \frac{1}{m}\sum_{j=1}^{m} d^\ell_{j,i}$ computed from users $\{1, 2, \cdots, j, \cdots, m\}$.

By thresholding y^ℓ_i, this method provides binary crowdsourcing result. One major problem with this method is that it treats each user equally hence the contribution of experts would be overwhelmed by crowds' less valuable opinions. In practice, some users are more

trustworthy while others may not be very responsible for their reviews, or even fraudulent or deceptive.

5.6.2 Crowdsourcing by Giving More Credit to Trustworthy Users

In this paper, we use a two-coin [49] model to annotate the apps from user-review level to app-level, which pays more credit on trustworthy users. More specifically, the probability that a worker labels an app correctly is assumed to follow Bernoulli distribution, one for the true positive label, and the other for negative. The major advantage of this approach is that we can give a more accurate prediction of the security behavior for each app by taking into account the credibility of difference users.

There are two cases. For a specific app i, we use α_u^ℓ to denote *sensitivity*, *i.e.,* a user u would label this security behavior ℓ in an app under the condition that the security-behavior ℓ really exists; and we use β_u^ℓ to denote *specificity*, *i.e.,* a user u would give negative label with respect to security behavior ℓ in an app under the condition that the security-behavior does not exist. Using mathematical formulation,

$$\alpha_u^\ell = \Pr(d_{u,i}^\ell = 1 | y_i^\ell = 1), \tag{5.3}$$

$$\beta_u^\ell = \Pr(d_{u,i}^\ell = 0 | y_i^\ell = 0), \tag{5.4}$$

where $i(1 \leq i \leq n)$ refers to each app, $u(1 \leq u \leq m)$ refers to each user, and $\ell(1 \leq \ell \leq r)$ refers to each label. Parameter $\alpha = [\alpha_u^\ell], \beta = [\beta_u^\ell]$ can be learned from training data using EM algorithm according to Maximum Likelihood Estimation (MLE) and y_i^ℓ is then computed via Bayesian rules.

According to the maximum likelihood principle, this approach maximizes the following objective function: $J(\theta, \mathbf{y}) = \max_{\theta,\mathbf{y}} \Big[\ln \Pr([d_{u,i}]|\theta, \mathbf{y}) + \ln \Pr(\theta) \Big]$, where $\theta = \{\alpha, \beta\}$ is the parameters as in Eqs.(5.3, 5.4), where α denotes the probability that a user will assign security label to a user review under the condition that the label is positive, and β denotes the probability that a user will not assign security label to a user review under the condition that the label is negative.

After a few steps of derivation via EM algorithm (see details in Appendix), finally, app-level behavior y_i^ℓ is expressed as:

$$y_i^l = \frac{a_i^\ell \Pr(y_i^\ell = 1|\theta)}{a_i^\ell \Pr(y_i^\ell = 1|\theta) + b_i^\ell \Pr(y_i^\ell = 0|\theta)}, \tag{5.5}$$

where a_i^ℓ is the likelihood of app i getting label ℓ, b_i^ℓ is the likelihood of app i not getting label ℓ, *i.e.,*

$$a_i^\ell = \prod_{u=1}^{m} (\alpha_u^\ell)^{d_{u,i}^\ell} (1 - \alpha_u^\ell)^{1 - d_{u,i}^\ell} \tag{5.6}$$

$$b_i^\ell = \prod_{u=1}^{m} (\beta_u^\ell)^{1 - d_{u,i}^\ell} (1 - \beta_u^\ell)^{d_{u,i}^\ell}, \tag{5.7}$$

and $\Pr(y_i^\ell = 1|\theta)$ is the prior probability for app i with label ℓ, and $\Pr(y_i^\ell = 1|\theta) = 1 - \Pr(y_i^\ell = 0|\theta)$.

5.6.3 Determination App-level Behavior via y_i^ℓ

However, the probability value y_i^ℓ is not directly useful for determination of app-level behavior. Notice that, although we utilize crowdsourcing algorithm to aggregate review-level labels to app-level labels, the review level labels are not really user annotation, but from auto annotation with a trained classifier (in Section 5.2). The review given by users are not originally annotations for our security issues. Therefore, the prior probability of finding a user review mentioning one of the four security issues (in Table 5.1) is quite low, even given the fact that the security issues exist for the app. This is easy to understand since the user may not actually encounter the security issue when using the app, and may not give a review after encountering the issue, or the review level classifier failed to recognize the *semantic* meaning in the review. As a result, the probability value of y_i^ℓ may not be taken as the probability of the app having the security issues, but more as a security-risk ranking score for comparing the security risk of the app having the issues against others.

In order to get a clear output for whether the app has the security behavior or not, a threshold need to be tuned for each label to provide the binary prediction. This threshold can be determined through active manual annotation, which is discussed in Section 5.7.

5.7 Experiment

In this section, we present the experiment result computed using AUTORBF. AUTORBF aims to bridge the gap between users' understanding of apps and what the apps' behavior really is. To evaluate the effectiveness of AUTORBF, we compare the behavior inferred at both review-level and app-level against the app behavior labeled by human being. we make a quantitative study on the performance of the system. More specifically, we design our experiment and answer the following two questions:

- **RQ1:** What is prediction performance using review-level security behavior inference model?
- **RQ2:** Can we get the app-level security behavior annotation result via crowdsourcing? What is the credibility of different users? Will users report the security problems?

Towards this goal, we first crawled the user-review dataset from Google Play via reverse engineering the service protocol. Then, using the crawled dataset, we evaluate the performance of our approach on review-level and app-level[1].

5.7.1 Data Collection

We collected our dataset from Google Play. On Google Play, a user's reviews about apps that he/she used are publicly available. Once we obtain the Google ID of a user, we can locate all apps the user has reviewed. Therefore, we can obtain a list of Google user IDs and write a crawler to retrieve all rated apps of these users. Moreover, for each retrieved app, we crawled its reviews from Google Play. The crawler was written in python.

[1]To encourage this line of research, we plan to make our dataset publicly available soon.

We evaluate the performance of review-level app behavior inference problem on one dataset collected during November 2013, containing $19,413$ user reviews from $3,174$ apps from Google play. This dataset is annotated manually by two graduate professionals in two months. Each user review is given a label by two annotators. If the two annotators reach a consensus, the user review is labeled as a specific label. Otherwise, the two annotators will discuss and reach a consensus on the controversial user reviews. Each user review was labeled with either one (or several) of the four labels described in Table. 5.1, or none of the above four labels. The annotation was conducted with expectation to respect the meaning of the reviews themselves rather than the actual app behaviors. We refer this dataset as L since it is manually labeled by human being. The statistics of dataset L is listed in Table. 5.2

Table 5.2.: L Dataset details to validate the review-level security behaviors. The Mean, Max and Min are statistics for the number of reviews for per app. The mean number of review per app is small because we already filtered out those review with over 3 ratings, which are the majority of them.

Dataset	#app	#Review	Mean	Max	Min
L	3,174	19,143	6	4,500	1

Moreover, we collect another data to validate the effectiveness of our approach on app-level security behavior inference problem. This dataset is collected through December 2013 to May 2014, containing $12,783$ apps with $13,129,783$ reviews from $2,614,186$ users. However, this dataset shares no intersection with the apps in dataset L. We refer to this dataset as dataset D. However, for this dataset, we did not hire enough labor to manually label this dataset, and get the ground truth of security behaviors with respect to each app. The good thing is that we can directly use this dataset to compare the differences between the results of code analysis and review analysis. Table 5.3 summarizes the basic statistics of this dataset. In addition, Fig. 3.1(a) shows the distribution of number of apps over the number of reviews we get and the distribution of apps over the 5 rating values from the reviews. Note that, the peak at 4000 is artificial, due to the fact that our crawler is set to only crawl the first $4,000$ reviews for each app. It is clear that most users tend to give high

ratings to the apps once they decide to give the reviews. This, however, also means that most of the reviews are not valuable for our purpose of detecting security issues.

Table 5.3.: D Dataset details to validate the app-level security behaviors. The Mean, Max and Min are statistics for the number of reviews for per app. The max number of reviews $(4,000)$ is artificial, because our web crawler is set to crawl only the first 4000 reviews for each app.

Dataset	#app	#Comment	#User	Mean	Max	Min
D	12,783	13,129,783	2,614,186	1,027	4,000	1

We next describe our evaluation results to demonstrate the effectiveness of AUTORBF in identifying review-level and app-level security behaviors.

5.7.2 RQ1: Review-level Security-behavior Inference

To answer RQ1, we quantify the effectiveness of AUTORBF on review-level security behavior inference.

Experiment setting

The evaluation of the proposed review-level classification is conducted on annotated dataset L. As a supervised method, a training set is required for training the model. The whole dataset set is randomly split in a $50\%/50\%$ manner into a training set with $10,893$ reviews and a testing set with $8,520$ reviews. This splitting is based on app-level, so the reviews for the same app can only be in training or testing set altogether and the number of reviews in the two sets are not even.

In solving the optimization problem for security behavior inference, a five-fold cross validation method is adopted for finding the best αs in training set. For semantic expansion, *TF-IDF* features with cosine similarity is adopted for the retrieval model and Lemur[2] is used as the actual tool for the word retrieval. Time constraint is enforced to prevent review expansion with "future relevant" reviews. The indexes of retrieval model are built for the two sets separately so that the model parameter like document number and IDF values will not interfere between sets. The mixture ratio α and the size of "relevant" document set R

[2] `http://www.lemurproject.org/`

for the expansion are fixed by using five-fold cross validation in training set for each label along with the scope. The size of R is selected from $\{1, 3, 5, 10\}$.

Evaluation Measurement The metric used for evaluation is precision, recall and F1 value. Let the number of true positives, false positives, true negatives, and false negatives be TP, FP, TN and FN, respectively w.r.t. a classifier. TP indicates that AUTORBF correctly labels the review as a security issue if the review is. FP indicates that AUTORBF incorrectly labels a review as a security issue if the review is not. TN indicates that AUTORBF correctly labels a review as a non-security issue if the review is not. FN indicates that AUTORBF incorrectly labels a review as a non-security issue if the review is.

The precision metric is defined as Precision $= \frac{TP}{TP+FP}$, and the recall metric is Recall $= \frac{TP}{TP+FN}$. The F_1 measure is the harmonic mean of precision and recall, i.e., $F_1 = \frac{2TP}{2TP+FP+FN}$, and accuracy is, Accuracy $= \frac{TP+TN}{TP+FP+TN+FN}$. Larger values of these metrics suggest better classification results. Thus the values of these metrics indicate how well the security behavior inference result matches with the human annotated result.

Experiment result analysis

Table. 5.4 shows the performance of our method in identifying different categories of security issues. The results indicate that, out of 8,520 reviews, AUTORBF effectively labels the security-related behaviors with the average precision, recall, F1 score, accuracy of 80.10%, 82.46%, 81.26%, 94.05%. After carefully looking at different categories of security-related behaviors, we find that AUTORBF can identify "spamming", "over-claimed permission" and "data leakage" issues at the average precision of 83.84%, 78.25%, 77.97%, respectively, and at the average accuracy of 91.96%, 95.99%, 93.46%, respectively. This confirms the effectiveness of using semantic expansion approach and advanced machine learning classifier in AUTORBF.

An exceptional case is the detection of security issues related to "finance", the precision of which is not as high as the above three categories. When we check the users' reviews, we find that the false positives and false negatives are higher than those computed from other security categories. We find that many users actually do not complain the "financial issue", but their reviews are labeled as "financial issues". Also, some users complain the

"financial issues" without using any words related to "*buy, purchase, pay, paid, etc*", which misleads AUTORBF in making a wrong decision.

Comparisons against the key-words based approach

For comparison purpose, we also apply key-words based approach as a baseline to show the necessity of using the proposed semantic expansion techniques and advanced machine learning method used in AUTORBF. A number of key-words are manually selected for each security label as sampled in Table 5.6, and this approach predicts all reviews that contain at least one of the key-words to be positive for the underlining label. Note that, these keywords are stemmed [3] in pre-processing, so that the words from same origin may be matched to each other. Follow the same evaluation measurement used in AUTORBF, we compute TP, FP, FN, TN, Precision, Recall, F1, Accuracy, and summarize the results in Table 5.5.

We then compute the performance improvement in using AUTORBF against the key-word based approach. Let ∇Precision, ∇recall, ∇F1, ∇ACC be performance difference between our approach and key-word based approach. Table 5.7 shows the comparisons of difference in identifying security issues between AUTORBF and key-word based approach. For all four categories and overall performance, AUTORBF gains significant performance improvement in terms of precision, F1 and Accuracy. We observed that the recall of our approach is generally lower than key-word based approach because our approach has higher false negatives. However, the key-word approach has extremely high false positives since many user reviews that include sensitive words do not necessarily cause security-related issues. Considering the tradeoff between the precision and recall, our approach outperforms key-word based approach significantly, *i.e.,* an average of 46.48% performance improvement in terms of F1, and an average of 51.36% performance gain in terms of accuracy.

[3] `http://en.wikipedia.org/wiki/Stemming`

5.7.3 RQ2: app-level Security Behavior Inference

Now we are ready to present our analysis result in terms of different security issues on app-level.

Experiment setting

Based on the trained classifier from the review level experiment, *e.g.,* the classifiers evaluated in Table 5.4, all reviews in dataset D are automatically annotated with the four labels. We apply the crowdsourcing algorithm introduced in Sec 5.6 to generate app level result $\mathbf{Y}$ for each app in D, while evaluating the user credibility of the users in D.

Experiment Results

In order to determine the thresholds for the four label, we labeled about 50 apps for each label and tuned the thresholds to best classify those labeled apps. These apps are randomly selected from those apps that have (y_i^ℓ) at the range of $[\text{mean}(y_i^\ell) - \text{std}(y_i^\ell), \text{mean}(y_i^\ell) + \text{std}(y_i^\ell)]$, where $\text{mean}(y_i^\ell)$ is the average values of y_i^ℓ, and $\text{std}(y_i^\ell)$ is the standard deviation of y_i^ℓ.

In this way, we get all the security labels for each app at app-level. For example, `CallToPark` is a web app that aims to make personal payment and account management faster. y_i^ℓ values computed from crowsourcing approach (Sec 5.6) is:

$$[0; \quad 0.997; \quad 0.002; \quad 0.003],$$

corresponding to four categories of security issues of *spamming*, *financial*, *over-claimed permission* and *data leakage*. The results indicate it has 3 security issues (*i.e.*, *financial, over-claimed permission and data leakage*) out of four. The serious degrees of security issues are reflected by these numbers. The greater numbers indicate the more serious security issues from user perspective, such as more complains and dissatisfaction. Due to space limit, we did not show more quantitative results with respect to each app.

Analysis of user credibility

A by-product of crowdsourcing in AUTORBF is the learned parameters for each user, indicating how reliable the user's reviews are in our system. Figs. 5.4, 5.5 show the distribution of the number of users based on the trained parameters.

The distribution of number of users over the learned α values shows three peaks (Fig. 5.4).

The middle peak is around 0.5 which is the manually set prior value. In fact, this peak is the expectation of the distribution, indicating the behavior of majority users.

The lower peak shows the users that are less likely to report the issues in reviews, hence are given less credibility, therefore contribute less in the crowdsourcing process. These users may be more focusing on the functionality and attractiveness over the security issues of the app, or they simply do not want to pay the effort of reporting the issues.

The higher peak shows the users that are much more likely to report a security issue. They are considered as experts and their reviews are considered with high weights in the crowdsourcing process. These users are sensitive to those four security issues and are more willing to report them once they found them.

The distribution of user over the learned β values (Fig. 5.5) shows not much information, compared to the one for α. Although α and β looks symmetric in Eqs.(5.3, 5.4), they are in fact quite different in data. Users are not obligated to report security issues in reviews, and their default behavior is to report nothing. Therefore, β, as the value of the probability of report nothing when the app do not have the issues, has a very high expectation with lower variance. As a result, the distribution for β has a clear tendency towards 1, and the three peaks are less obvious than α (note that the number of users is in log scale, the peak near 1 is actually much more significant).

It is worth noting that, the parameters are learned based mainly on how well the users' opinions match the majority. It does not reveal the exact reason why the credibility is low for some user. So we are not considering those users with low credibility as fraud users, it is just that their opinions carry less value for these four underlining security problems and it is perfectly fine for users not reporting any of the issues in reviews.

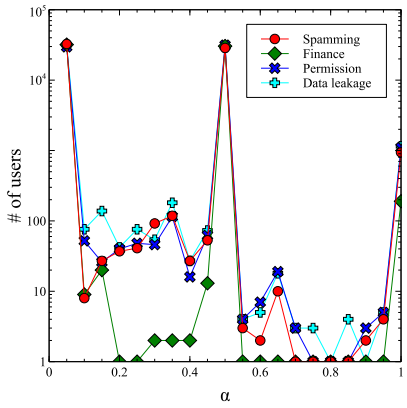


Figure 5.4.: Distribution of learned α parameters for number of users in log scale.

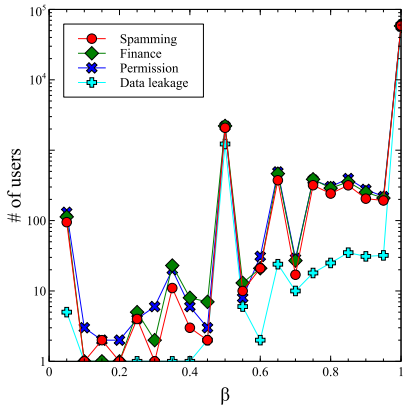


Figure 5.5.: Distribution of learned β parameters for number of users in log scale.

This also shows the reason why we use the so-called two-coin crowdsoucing model instead of the simple majority voting in crowdsourcing precess. With the distinguishing of different types of users, the app level result would be much more reliable.

5.8 Behavior Gap between AUTORBF and Code Analysis

The analysis results from user reviews reflect users' understanding and perception of security issues. Different users have exhibited different security concerns depending on personal preference. In the app-level crowdsoucing results, we can get the crowdsourcing results for each app. For each app, based on the statistics from different users, one can see how many users have complained one security concern, and how many users never mind or

simply do not notice the security concern. And it is hard to say which user is right or which user is wrong. We can only say different users have different preference to report different security concerns.

The results from both static analysis (*e.g.*, [14] [7] [75] [76]) and dynamic analysis (*e.g.,* [77] [78] [79] [80]) can reflect the code inherent structures and behaviors objectively, although sometimes incomplete, due to the inherent limitations of static analysis and dynamic analysis (*e.g.*, undecidability of static analysis, non- coverage characteristic of dynamic analysis, etc). However, it is not enough to reflect how users' real security concerns during using the app. Generally, users have different psychological expectations for different apps, and do not necessarily worry about the same security issue [73]. Our study provides the quantitative study result to understand the security concerns of different users, and also the credibility of users. The security issue inferred at app-level via crowdsourcing reflects the psychological needs of users, and is really something that users are worried about.

Thus, in our evaluation, we first make a comparative analysis of the "subjective" results from user behavior inference via crowdsourcing, against the objective results from code analysis. After that we analyze why these differences exist using case study. In the following, we will analyze four categories of security behaviors one by one.

5.8.1 Spamming

Spamming here refers to both foreground and background ads and spamming behaviors, including Ads in notification bar, ads via email, ads via SMS, Pop-up ads, Fishing, *etc*. Ads libraries in Android ecosystem offer developers solutions for monetizing the apps by displaying ads to users. Over half of the apps in Google play include an ad or ads library [14]. The ads libraries communicate with the ads network to get and show proper ads, based on the app context and personalized user information (such as location, context, *etc*). Many ads libraries collect users' personal information, by which they do target ads, which results in serious security concerns.

Most (if not all) of the users dislike advertising practices, although some users may not realize that ads cause security risks. In AUTORBF, we infer the ads spamming behavior from the users' reviews.

From code analysis view, different mobile apps embed different ads libraries. We use the methods proposed in [89] as a guideline to implement our own analysis tool to extract the ads-library used in a mobile app. We first manually investigate the usages of the most popular 100 ads libraries [89] to identify the usage signatures of each ads library in a mobile app, and then we automatically extract the list of ads libraries used by a mobile app by searching the decompiled code of the app using signature matching. Through this, we can get a list of ads library used by each app.

We compare the results of "spamming analysis" from running ads library analysis tool and AUTORBF on apps in dataset D, while on 67.5% apps, ads analysis indicates that the app uses ads and AUTORBF agrees, on 32.5% apps, ads analysis indicates that the app uses ads but AUTORBF disagrees. This implies that *the set of the app that has "spamming " security issues* AUTORBF *detected is only a* subset *of that of ads library analysis.*

Further, we make a case study on two specific apps.

⊙ **Example 1: `Pandora internet radio`**

Pandora is a free, personalized radio that allows users to play music that he/she loves, and helps users to discover new music and enjoy old favorite songs, artist, composer, etc. In this example, ads analysis indicates the app `Pandora` uses ads, and the AUTORBF agree. Many reviews have claimed there are too many ads! From the review-level and app-level analysis results, we can easily verify it. For example, one reviewer's review[1]

"Too many annoying ads I may not have noticed the ads as much if they didn't play the same 3 irritating ads every 3-5 songs. Switching to iHeartRadio's much much better ad-to-song ratio."

⊙ **Example 2: `Yelp`**

The free `Yelp` mobile app is one of the most popular apps that search for business (*e.g.*, restaurant, bars, coffee, teas, *etc*) around you. When we do ads analysis, the result indicates that `Yelp` uses the ads library `com.google.ads`. However, AUTORBF disagrees.

[1]Due to privacy reason, we omit the name of the review.

Most of the users give very high-recommendation for this app, and it has an overall score of 4.3. An example of review is shown below.

"Yelp is the best Before going ANYwhere, I first check it out on yelp. Nail salons, restaurants, coffee shops, liquor stores, anything! I trust my fellow yelpers to give me the honest truth, and that's why yelp is the best. Plus the app makes it incredibly easy to get directions, write a review, and find out what's good in your area all from your phone!"

Very interestingly, almost none users complain the "spamming" issue especially ads for `Yelp`. `Yelp` itself can be viewed as a "spmming ads", which always helps you to discover great local businesses, and search for nearby restaurants, shops, services, *etc*. It is natural that no users will take it as an ad.

★ **Lessons Learned** Given the ads library used by apps, why user complains about some apps but not others? The ads detection task defined in ads analysis from code analysis perspective may not comply with what users really feel about the ads.

5.8.2 Financial Issue

There are external issues that bother users' experience and cause security concerns that are beyond the running app in users' device. These issues mainly involve the web services used by the running apps to facilitate their functionalities. In this paper, we focus on one type of external issues that may cause financial problems to users. We define the financial issues as those complains about In-App-Purchase (IAP), free app to premium update and others that cause the users actual money lost in external context other than the running apps. For example, user may pay for IAP or upgrade but get nothing as described, a bank app may transfer money for user but only deduction happened without deposit. There issues may caused by program bugs but also by intentional fraud. Therefore it would be quite important that we can find a way to detect them.

From code analysis perceptive, there is almost no way to clearly detect the financial issue even if your account information is sent out of the phone. We are aware that flow

and dynamic analysis (*e.g.*, taintdroid [79], flowdroid [83], droidbox[1], *etc*) can potentially detect the users' banking account information going out of the phone through taint analysis and running the app in a virtual controlled environment. However, even if users' account information is leaked, we cannot say that financial issue really happens. In contrast, based on user review analysis, we can get evidence/clues of how financial problems occurred. Examples related to *IAP* and *free to premium* are shown below.

⊙ **Example 1: Cases for IAP**

"I baught $2.00 in papaya's and I never got them. In really mad about it. I've spent a lot of money on this game. If I get what I paid for I will retract my statement but until then no"

"I paid for the year subscription and a week later I was getting notifications that my subscription was almost up. And a few days the app stopped working. So I emailed the developer to fix this and he never got back to me. So now I wasted my money on this app I can't even use. This is staying a one star until it's fixed."

"My weapons that I purchased have dissapeared. I payed real money! "

"I use to love this game ...having been playing it for years but recently I started purchasing coins with my credit card and never received them but I call and check my balance and the money has been deducted from my account....thieves "

⊙ **Example 2: Cases for free to premium**

"So angry!! Paid for this app and it hardly worked for the 1st day then the second day it stopped working completely!! The developer wont reply to my emails so basically my money has been stolen by this developer!! Btw the free version of this works fine as soon as you buy it boom it stops working!!! Think the developer is trying to take the Mick of people who buy this!!"

"Paid for the premium one but was unable to download videos. Very upset as money wasted."

★ **Lessons Learned** Running app is only part of the user experience, which may be partially or fully investigated by code analysis (including static flow analysis and dynamic analysis). However, there exits environmental or external factors that matters too. For those factors, user review serves as a better and more complete source for analysis. For the security behavior related to app environment, it cannot be inferred from code analysis. We

[1] `https://code.google.com/p/droidbox/`

believe there are also other external issues that are only detectable from user reviews but not code analysis. Due to space limit, we did not show more examples here.

5.8.3 Over-claimed Permission

In Android system, there are more than 170 permission which are request by users to access the phone hardware, settings, user data like location, contact, photos, account, etc. *More permission than expectation* here refers to the app request more permission than users' expectation. It has subtle difference from the "over-claimed permission". When we say "over-claimed" permission, it indicates that the app requests more permission as compared to the permissions that are actually used during the running of the apps, which is from app developers' perspective. What we mean, however, is from users' expectation perspective. As long as users do not think that the app requests more permissions than his/her expectation, we believe that this app does not request too much permission.

From code analysis view, we can get the truly called permissions by an app through static analysis. Specifically, we firstly decompile the apk files of a mobile app and obtain the function call graph (FCG) of the app. Then we traverse the function call graph (FCG) to trace any functional call that invokes permission check. By doing so, we are able to obtain the real permission that is used in an app. More details about how to identify used permission in a mobile app can be found in [4].

We compare the results of "more permission than users' expectation" from running AUTORBF on apps in dataset D, against the "over-claimed" permission by running static analysis tools on mobile apps, while on 58.5% apps, code analysis indicates the app has over-claimed permission and AUTORBF agrees the app requests "more permissions than users' expectation"; and on 41.5% apps, code analysis indicates the app has over-claimed permission but AUTORBF disagrees that the app request "more permission than users' expectation". The numerical value further implies that the two goals are not aligned.

⊙ **Example 1: `Angry Birds`**

The free `Angry Birds` mobile app is a video game franchise created by Finnish computer game developer Rovio Entertainment, and is called "*one of the most mainstream games out right now*"[1]. When we do code analysis, the result indicates that the permission `location` is not necessarily needed in function calls. However, in AUTORBF result, almost no users complain that it requests too more permission than expectation. An example of review is shown below.

"Excellent Game I enjoy playing this awesome game as it's cool with good graphics and background sound but ads videos anoying me. Also sometimes I feel boring whenever I have to replay again and again to cross the levels. Overall, it is exciting game....."

⊙ **Example 2: `dictionary`**

The free `dictionary.com` mobile app is the top 1 fee English dictionary app since 2010 with over 2,000,000 definitions and synonyms. When we do code analysis, the result indicates that the permission `location` is really accessed in FCG. However, from user's perspective, there is no need to access location for an online/offline dictionary. In AUTORBF result, some users complain it requests users' `location` and `media file` resources. An example of review is shown below.

"Too much information Why do you need my location and media files? Access to the internet I understand but this is ridiculous! My pictures and my mic are an invasion of privacy, especially since this is a DICTIONARY!!"

★ **Lessons Learned** Whether the permission is over-claimed depends highly on what the expectation the users have on the app. Code analysis (including static analysis and dynamic analysis) may not be able to grab the expectation, from the users' mental needs.

5.8.4 Data leakage

"*Data leakage*" refers to access sensitive information without user acknowledgement. For example, an app may access users' account, location, contact, media file, phone number

[1] `http://www.digitaltrends.com/gaming/israeli-angry-birds-satire-goes-viral/`

without asking for users' consent, such that users' personal information are already sent out of the phone.

From code analysis view, we can get the truly accessed resources by each app from static analysis on the flow of app (*e.g.*, flowdroid [83]) or dynamic analysis (*e.g.*, taintdroid [79], *etc*) to capture the running time behaviors. Specifically, we run the dynamic analysis tool to analyze the dynamic behaviors of mobile apps, which emulates the actions of user interaction, incoming calls, SMS messages, I/O file read/write, network operations, telephone records (such as IMEI,IMSI, MSISDN, *etc*). By doing so, we are able to obtain the resources that the app accessed during running.

We compare the results of "data leakage" from running AUTORBF on apps in dataset D, against "access sensitive personal data via network traffic. file writes, and SMS texts" by running the dynamic analysis tool; while on 73.5% apps, code analysis indicates the app has accessed sensitive personal data and AUTORBF agrees that the app requests "access sensitive data" and on 26.5% apps, code analysis indicates the app accesses sensitive data but AUTORBF do not.

⊙ **Example 1: `Accuweather`**

The free `Accuweather` mobile app is one of the leading minute-by-minute weather forecast, and the weather report can be localized to your exact street address. When we do code analysis, the result indicates that it accesses photos/media/file, and has the potential of file read/write activity on devices, resulting in "data leakage" behaviors. However, in AUTORBF result, most (if not all) of the users have very high recommendation for this app, and does not classify it as "data leakage". From users' perspective, most of the users trust it. An example of review is shown below.

"Reliable 100% and accurate 85%, Why? (read comment below) Therefore I rate this app 100% to anyone, one tiny problem it cannot find my location with is ST FRANFIS BAY Eastern Cape South Africa! It can find cape st Francis but that about 10/15min away! "

⊙ **Example 2: `Aroundme`**

The free `Aroundme` mobile app is an app which consistently shows you a complete list of all the businesses such as nearest bank, bar, gas station, hospital, hotel, movie theatre,

restaurant, *etc*. When we do code analysis, the result indicates that it accesses many personal data, such as location, photos/media/files, and these information are probably sent out of the phone. In AUTORBF result, it labels `aroundme` as "data leakage", because from user's perspective, it consistently requests some resources such as location. An example of review is shown below.

" Always search for current location ... Though GPS on this app prompts for setting. I hate this nonsense bull sheet app. 2ndly When change the units still it shows yards ... Works crazy with Android... "

★ **Lessons Learned** We can find whether the app accessed the users' personal data, *e.g.*, location, account, contact, *etc* through code analysis. However, we do not know whether the users feel comfortable if their personal information are accessed, and even sent to the third-party. From users' perspective, AUTORBF can capture users' expectation and psychological needs.

5.8.5 Summary & Insight

Although deep analysis from code perspective can be applied before the apps are put into the app store, user reviews are more like an update of the previous security analysis on the app. It is an important complementary analysis to previous analysis of app such as static and dynamic analysis due to the fact that:

- *The user reviews reveal how the users feel about the experience, but not what really happened. Users' expectation plays a big role on how much the users can tolerate the apps' behavior, as show in the case of different apps likes* `aroundme`, `angrybird`, `dictionary`, `yelp`, *etc. Similar app behaviors may receive different responses from users.*

- *External factors of the app-using behavior are much easier revealed by user reviews.*

- *Privacy issues are relative and personal. The borderlines between privacy-intruding and tolerable misbehavior are fuzzy and depend highly on users' subjective expecta-*

tions. Hence user review may provide an important complementary source for determining the threshold based on the objective app behavior revealed by code analysis.

5.9 Conclusion

In this paper, we propose the system AUTORBF that understands the review-to-behavior fidelity in Android apps, *i.e.*, it can infer the mobile app security related behaviors from the apps' reviews from different users via crowdsourcing. Our novel learning-based algorithm and advanced crowdsourcing techniques are able to mine the relationships between user reviews and app security behaviors. To our knowledge, AUTORBF is the first work that has the capability to accurately detect the apps' security behaviors from user reviews. In inferring four categories of security related behaviors from user reviews, our system achieves the average accuracy as high as 94.05%. We also get credibility for different users at app-level behavior prediction. Our study provides valuable insights and quantitative analysis in understanding the app behaviors from the users' view.

Table 5.4.: Evaluation on different metrics in AUTORBF. # size denotes the number of positive samples with respect to the label, and ACC denotes accuracy.

Label	# size	TP	FP	FN	TN	Precision	Recall	F1	ACC
spamming	2,788	2,574	496	214	5,236	83.84%	92.32%	87.88%	91.66%
Financial	578	337	179	241	7,763	65.31%	58.30%	61.61%	95.07%
Over-claimed Permission	711	511	142	200	7,667	78.25%	71.87%	74.93%	95.99%
Data leakage	1,258	977	276	281	6,986	77.97%	77.63%	77.82%	93.46%
Sum	5,335	4,399	1,093	936	27,652	N/A	N/A	N/A	N/A
Average	N/A	N/A	N/A	N/A	N/A	80.10%	82.46%	81.26%	94.05%

Table 5.5.: Evaluation on different metrics using key-word based approach. # size denotes the number of positive samples with respect to the label, and ACC denotes accuracy.

Label	# size	TP	FP	FN	TN	Precision	Recall	F1	ACC
spamming	2,788	2,780	5,619	8	113	33.10%	99.71%	49.70%	33.96%
Financial	578	549	3,282	29	4,660	14.33%	94.98%	24.90%	61.14%
Over-claimed Permission	711	706	6,012	5	1,797	10.51%	99.30%	19.00%	29.38%
Data leakage	1,258	1,172	4,490	86	2,772	20.70%	93.16%	33.87%	46.29%
Sum	5,335	4,399	1,093	936	27,652	N/A	N/A	N/A	N/A
Average	N/A	N/A	N/A	N/A	N/A	21.16%	97.60%	34.78%	42.69%

Table 5.6.: Sampled key-words used in "Key-word based approach".

Label	key-words
spamming	ad, spam, notif, advertis, advert, spammi, add, money, secur, push, etc
Financial	text, deduct, sm, bought, paid, took, privat, txt, taken, charg, purchas, etc
Over-claimed Permission	permiss, access, money, read, info, privaci, regist, camera, need, want, contact, requir, ask, locat, data, internet, request, credit, email, call, necessari, etc
Data leakage	permiss, privaci, info, hack, access, contact, money, fool, lie, id, ground, inform, lockscreen, steal, wit, wall, camera, data, requir, phish, internet, licenc, locat, etc

Table 5.7.: Performance difference between our approach and key-word based approach. ∇ indicates the performance difference in terms of different metrics.

Label	∇Precision	∇ Recall	∇F1	∇ACC
spamming	50.74%	−7.39%	38.18%	57.70%
Financial	50.98%	−36.68%	36.71%	33.93%
Over-claimed Permission	67.74%	−27.43%	55.93%	66.61%
Data leakage	57.27%	−15.53%	43.95%	47.17%
Average	68.94%	−15.14%	46.48%	51.36%

6 SUMMARY

The works of this dissertation is motivated to explore new ways in managing the security and privacy issues in the thriving Android platform. With machine learning as our weapon, we conducted a series of data-driven researches on different Android app related security issues. These researches approach from two directions: direct analysis of the Android app and indirect analysis from user comments.

6.1 Direct Analysis

Our direct analysis approach focus on malicious app detection and propose to use probabilistic discriminative model to improve the performance and decompiled source code of the app as information source. This results in a over 95% F1 score and is a huge improvement from previous work.

The limitation of this work comes from several aspect.

- Data-driven means the performance relies on the training data. Although this approach use function name occurrence as feature and would be more robust on malicious code variants than exact code piece signature based methods, it will not guarantee to response to new threat until retraining.

- This is a static analysis, so it will not defend against runtime change of code logic by the Reflection feature in Java and the alike. This technique is best used for screening of apps before more complicated code analysis including expensive dynamic analysis.

6.2 Indirect Analysis

Our indirect analysis approach makes use of the user comments to extract relevant topics, then evaluate the app based on that. This is a new line of work that has not been explored yet. We collected user comment dataset and did annotation with a designed label set with consideration of both the scenario and the nature of the security issues. A topic extraction method is provided to extract security topics regarding the labels from use comments. Then the topics of each comments are used in evaluating the apps' security risk. By treating these extracted topics as user annotation to the app, we applied crowdsourcing technique to accumulate topics from user comment level to app level while estimate user credibility in commenting apps with respect to different security topics. Finally, we evaluate the apps' security risk by a learning to rank process, leveraging app level security topics and features. Moreover, in order to improve the performance, we developed a joint optimization method to jointly train both the crowdsourcing and learning to rank model and get about 7% improvement in terms nDCG metric.

Further effort are done to compare the user comment based indirect analysis to traditional code analysis based direct analysis on multiple security issues of Android apps. We confirmed that the new user comment based indirect analysis would be a necessary complimentary means in solving Android app security issues due to the following observations:

- User comment based method may cover some issues that direct code analysis may not or are hard to deal with. For example the IAP issue that is heavily related to server side behavior other than apps on the smart phone.

- User expectation plays a important role in mobile app privacy issues. Whether an app is using Ads abusively or whether a permission should be granted, these questions are not just technical issues but also very much depend on user expectations of the underlining app. And user comment based analysis may shed some light on what user expect from the apps.

- Objectives misalignment between code analysis and user comment based analysis. For example, in the case of over-claimed permission issue, it is natural for code

analysis to set task to find mismatch between the permissions claimed and the permissions in use. But from the users' perspective, one may argue that claim an unnecessary permission is bad, actually using it is worse. The "over-claimed" may mean the mismatch between the claimed permission and the user expectation of the app.

As a new direction of work, limitations are inevitable but also point out future directions.

- Comment spamming remains a problem for user comment based analysis. Unreliable user comments render all analysis built upon them untrustworthy. Therefore it would be necessary to explore user credibility from user intention perspective (anti-spam) instead of user ability. Making use of meta-data of the comment posting activity may be greatly helpful in identifying spam user and separate their comments from the evaluation of the app. Based on experience from other anti-spam works (email spam, Twitter spam, etc.), this seems to be a start.

- Our current auto labelling method has not been using the most state-of-the-art Natural Language Processing (NLP) technique for identifying user opinion. By incorporating some well studied NLP method, the performance of user comment labelling could be improved, hence improving the evaluation of the apps' security.

Related Publications

The material in this dissertation is mainly based on the papers listed below, which have already been published to conference or workshop proceedings.

- Deguang Kong, **Lei, Cen**, Hongxia Jin. AUTOREB: Automatically understanding the review-to-behavior fidelity in Android applications. CCS, 2015
- **Lei, Cen**, Deguang Kong, Hongxia Jin, Luo Si. Mobile app Security Risk Assessment: A Crowdsourcing Ranking approach from User Comments. SDM, 2015.
- **Lei Cen** Luo Si, Ninghui Li, and Hongxia Jin. User comment analysis for Android apps and CSPI detection with comment expansion. In SIGIR 2014 workshop: Privacy-preserving Information Retrieval, 2014.
- **Lei Cen**, Chris Gates, Luo Si, Ninghui Li. A Probabilistic Discriminative Model for Android Malware Detection with Decompiled Source Code. In TDSC, 2014.

REFERENCES

REFERENCES

[1] Hao Peng, Chris Gates, Bhaskar Sarma, Ninghui Li, Yuan Qi, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Using probabilistic generative models for ranking risks of Android apps. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 241–252, New York, NY, USA, 2012. ACM.

[2] William Enck, Machigar Ongtang, and Patrick 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.

[3] Bhaskar Pratim Sarma, Ninghui Li, Chris Gates, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Android permissions: A perspective combining risks and benefits. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, SACMAT '12, pages 13–22, New York, NY, USA, 2012. ACM.

[4] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 627–638. ACM, 2011.

[5] Alexander Genkin, David D. Lewis, and David Madigan. Large-scale Bayesian logistic regression for text categorization. *Technometrics*, 49(3):291–304, 2007.

[6] Qifan Wang, Luo Si, and Dan Zhang. A discriminative data-dependent mixture-model approach for multiple instance learning in image classification. In *Proceedings of the 12th European Conference on Computer Vision*, ECCV '12, pages 660–673. Springer, 2012.

[7] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. Riskranker: Scalable and accurate zero-day Android malware detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 281–294, New York, NY, USA, 2012. ACM.

[8] A.-D. Schmidt, J.H. Clausen, A. Camtepe, and S. Albayrak. Detecting Symbian OS malware through static function call analysis. In *Proceedings of the 4th International Conference on Malicious and Unwanted Software*, MALWARE '09, pages 15–22, Piscataway, NJ, USA, 2009. IEEE Press.

[9] Haibo He and Edwardo A. Garcia. Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering*, 21(9):1263–1284, September 2009.

[10] J. Zico Kolter and Marcus A. Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 7:2721–2744, December 2006.

[11] Jiyong Jang, David Brumley, and Shobha Venkataraman. Bitshred: Feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 309–320, New York, NY, USA, 2011. ACM.

[12] Anthony Desnos. Android: Static analysis using similarity distance. In *Proceedings of the 45th Hawaii International Conference on System Sciences*, HICSS '12, pages 5394–5403, Washington, DC, USA, 2012. IEEE Computer Society.

[13] Aubrey-Derrick Schmidt, Rainer Bye, Hans-Gunther Schmidt, Jan Clausen, Osman Kiraz, Kamer A. Yüksel, Seyit A. Camtepe, and Sahin Albayrak. Static analysis of executables for collaborative malware detection on Android. In *Proceedings of the 2009 IEEE International Conference on Communications*, ICC '09, pages 1–5, Piscataway, NJ, USA, 2009. IEEE Press.

[14] William Enck, Damien Octeau, Patrick McDaniel, and Swarat 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.

[15] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, February 2012.

[16] Yousra Aafer, Wenliang Du, and Heng Yin. DroidAPIMiner: Mining API-level features for robust malware detection in Android. In *Security and Privacy in Communication Networks*, pages 86–103. Springer, 2013.

[17] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. Mining specifications of malicious behavior. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC-FSE '07, pages 5–14, New York, NY, USA, 2007. ACM.

[18] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and classification of malware behavior. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 108–125, Berlin, Heidelberg, 2008. Springer-Verlag.

[19] Michael Bailey, Jon Oberheide, Jon Andersen, Z. Morley Mao, Farnam Jahanian, and Jose Nazario. Automated classification and analysis of internet malware. In *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection*, RAID '07, pages 178–197, Berlin, Heidelberg, 2007. Springer-Verlag.

[20] Asaf Shabtai and Yuval Elovici. Applying behavioral detection on Android-based devices. In *Mobile Wireless Middleware, Operating Systems, and Applications*, pages 235–249. Springer, 2010.

[21] Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. Paranoid Android: Versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 347–356, New York, NY, USA, 2010. ACM.

[22] Iker Burguera, Urko Zurutuza, and Simin 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*, SPSM '11, pages 15–26, New York, NY, USA, 2011. ACM.

[23] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: Analyzing the Android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 217–228, New York, NY, USA, 2012. ACM.

[24] Adrienne Porter Felt, Kate Greenwood, and David Wagner. The effectiveness of application permissions. In *Proceedings of the 2nd USENIX Conference on Web Application Development*, WebApps '11, pages 7–7, Berkeley, CA, USA, 2011. USENIX Association.

[25] Jesse Davis and Mark Goadrich. The relationship between Precision-Recall and ROC curves. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 233–240, New York, NY, USA, 2006. ACM.

[26] Yiming Yang and Jan O. Pedersen. A comparative study on feature selection in text categorization. In *Proceedings of the 14th International Conference on Machine Learning*, ICML '97, pages 412–420, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[27] Stephen P. Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[28] Brian Quanz and Jun Huan. Aligned graph classification with regularized logistic regression. In *Proceedings of SIAM International Conference on Data Mining*, SDM '09, pages 353–364. SIAM, 2009.

[29] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, Piscataway, NJ, USA, 2012. IEEE Press.

[30] Tom Fawcett. An introduction to ROC analysis. *Pattern Recognition Letter*, 27(8):861–874, June 2006.

[31] Stefan Axelsson. The base-rate fallacy and the difficulty of intrusion detection. *ACM Transactions on Information and System Security*, 3(3):186–205, 2000.

[32] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. DroidChameleon: Evaluating Android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, pages 329–334. ACM, 2013.

[33] Hien Thi Thu Truong, Eemil Lagerspetz, Petteri Nurmi, Adam J. Oliner, Sasu Tarkoma, N. Asokan, and Sourav Bhattacharya. The company you keep: Mobile malware infection rates and inexpensive risk indicators. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 39–50, 2014.

[34] Ning Chen, Jialiu Lin, Steven CH Hoi, Xiaokui Xiao, and Boshen Zhang. AR-Miner: Mining informative reviews for developers from mobile app marketplace. *International Conference on Software Engineering*, 2014.

[35] Bin Fu, Jialiu Lin, Lei Li, Christos Faloutsos, Jason Hong, and Norman Sadeh. Why people hate your app: Making sense of user feedback in a mobile app store. In *Proceedings of the 19th ACM International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 1276–1284. ACM, 2013.

[36] Laura V. Galvis Carreño and Kristina Winbladh. Analysis of user comments: An approach for software requirements evolution. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 582–591. IEEE Press, 2013.

[37] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent Dirichlet allocation. *The Journal of Machine Learning Research*, 3:993–1022, 2003.

[38] Gjorgji Madjarov, Dragi Kocev, Dejan Gjorgjevikj, and Sašo Džeroski. An extensive experimental comparison of methods for multi-label learning. *Pattern Recognition*, 45(9):3084–3104, 2012.

[39] Grigorios Tsoumakas and Ioannis Katakis. Multi-label classification: An overview. *International Journal of Data Warehousing and Mining*, 3(3):1–13, 2007.

[40] Jesse Read, Bernhard Pfahringer, Geoff Holmes, and Eibe Frank. Classifier chains for multi-label classification. *Machine Learning*, 85(3):333–359, 2011.

[41] Christopher M. Bishop. *Pattern Recognition and Machine Learning*, volume 1. Springer, Secaucus, NJ, USA, 2006.

[42] Jinxi Xu and W. Bruce Croft. Query expansion using local and global document analysis. In *Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '96, pages 4–11, New York, NY, USA, 1996.

[43] Kun Liu and Evimaria Terzi. A framework for computing the privacy scores of users in online social networks. *ACM TKDD*, 5(1):6, 2010.

[44] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. WHYPER: Towards automating risk assessment of mobile applications. In *Proceedings of the 22th USENIX Security Symposium*, pages 527–542, 2013.

[45] D. Kong and H. Jin. Towards permission request prediction on mobile apps via structure feature learning. In *Proceedings of SIAM International Conference on Data Mining*, SDM '15.

[46] M. Frank, B. Dong, A. P. Felt, and D. Song. Mining permission request patterns from Android and Facebook applications. In *12th IEEE International Conference on Data Mining, Brussels, Belgium*, pages 870–875, 2012.

[47] B. Liu, D. Kong, L. Cen, N. Gong, H. Jin, and H. Xiong. Personalized mobile app recommendation: Reconciling app functionality and user privacy preference. In *Proceedings of the 8th ACM International Conference on Web Search and Data Mining*, WSDM '15, 2015.

[48] Alexander Philip Dawid and Allan M. Skene. Maximum likelihood estimation of observer error-rates using the EM algorithm. *Applied Statistics*, pages 20–28, 1979.

[49] Vikas C. Raykar, Shipeng Yu, Linda H. Zhao, Gerardo Hermosillo Valadez, Charles Florin, Luca Bogoni, and Linda Moy. Learning from crowds. *The Journal of Machine Learning Research*, 11:1297–1322, August 2010.

[50] Dengyong Zhou, Sumit Basu, Yi Mao, and John C. Platt. Learning from the wisdom of crowds by minimax entropy. In *Advances in Neural Information Processing Systems*, pages 2195–2203, 2012.

[51] Dengyong Zhou, Qiang Liu, John C. Platt, and Christopher Meek. Aggregating ordinal labels from crowds by minimax conditional entropy. In *Proceedings of the 31st International Conference on Machine Learning*, June 2014.

[52] Ioanna Lykourentzou, Dimitrios J. Vergados, Katerina Papadaki, and Yannick Naudet. Guided crowdsourcing for collective work coordination in corporate environments. In *Computational Collective Intelligence, Technologies and Applications*, pages 90–99. Springer, 2013.

[53] Ping Li, Qiang Wu, and Christopher J.C. Burges. McRank: Learning to rank using multiple classification and gradient boosting. In *Advances in Neural Information Processing Systems*, volume 20, pages 897–904, 2007.

[54] Koby Crammer and Yoram Singer. Pranking with ranking. In *Advances in Neural Information Processing Systems*, volume 14, pages 641–647, 2001.

[55] Thorsten Joachims. Optimizing search engines using clickthrough data. In *Proceedings of the 8th ACM SIGKDD*, pages 133–142. ACM, 2002.

[56] Christopher J.C. Burges. From RankNet to LambdaRank to LambdaMART: An overview. *Microsoft Research Technical Report MSR-TR-2010-82*, 11:23–581, 2010.

[57] Yoav Freund, Raj Iyer, Robert E. Schapire, and Yoram Singer. An efficient boosting algorithm for combining preferences. *The Journal of Machine Learning Research*, 4:933–969, 2003.

[58] Michael Taylor, John Guiver, Stephen Robertson, and Tom Minka. SoftRank: Optimizing non-smooth rank metrics. In *Proceedings of the 2008 International Conference on Web Search and Data Mining*, pages 77–86. ACM, 2008.

[59] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. Learning to rank: From pairwise approach to listwise approach. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07.

[60] Lei Cen, Luo Si, Ninghui Li, and Hongxia Jin. User comment analysis for android apps and CSPI detection with comment expansion. In *SIGIR 2014 Workshop: Privacy-preserving Information Retrieval*, 2014.

[61] James C. Bezdek and Richard J. Hathaway. Convergence of alternating optimization. *Neural, Parallel and Scientific Computations*, 11(4):351–368, December 2003.

[62] Alex J. Smola and Bernhard Schölkopf. A tutorial on support vector regression. *Statistics and Computing*, 14(3):199–222, 2004.

[63] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2(3):27:1–27:27, May 2011.

[64] Yining Wang, Wang Liwei, Yuanzhi Li, Di He, Wei Chen, and Tie-Yan Liu. A theoretical analysis of NDCG ranking measures. In *26th Annual Conference on Learning Theory*, 2013.

[65] App Store Statistics. *http://en.wikipedia.org/wiki/App_Store_(iOS)*.

[66] W. Lee and S. J. Stolfo. Data mining approaches for intrusion detection. In *Proceedings of the 7th Conference on USENIX Security Symposium – Volume 7*, SSYM '98, pages 6–6, 1998.

[67] R. Potharaju, N. Jain, and C. Nita-Rotaru. Juggling the jigsaw: Towards automated problem inference from network trouble tickets. In *10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '13, pages 127–141, Lombard, IL, 2013. USENIX.

[68] J. Newsome, B. Karp, and D. Song. Polygraph: Automatic signature generation for polymorphic worms. In *IEEE Symposium on Security and Privacy*, May 2005.

[69] C. Liu, C. Chen, J. Han, and P. S. Yu. GPLAG: Detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 872–881, 2006.

[70] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen. AutoCog: Measuring the description-to-permission fidelity in Android applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 1354–1365, 2014.

[71] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. In *Proceedings of the 21th ACM Conference on Computer and Communications Security*, CCS '14, Scottsdale, AZ, November 2014.

[72] M. Harman, Y. Jia, and Y. Zhang. App store mining and analysis: MSR for app stores. In *Proceedings of the 9th Working Conference on Mining Software Repositories, MSR '12*, Zurich, Switzerland, 2-3 June 2012.

[73] J. Lin, Shahriyar Amini, J. I. Hong, N. Sadeh, J. Lindqvist, and J. Zhang. Expectation and purpose: Understanding users' mental models of mobile app privacy through crowdsourcing. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, pages 501–510, 2012.

[74] A. P. Felt, S. Hanna, E. Chin, H. J. Wang, and E. Moshchuk. Permission re-delegation: Attacks and defenses. In *the 20th Usenix Security Symposium*, 2011.

[75] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. MAST: Triage for market-scale mobile malware analysis. In *Proceedings of the 6th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '13, pages 13–24, 2013.

[76] C. Gibler, J. Crussell, J. Erickson, and H. Chen. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, TRUST '12, pages 291–307, 2012.

[77] M. Neugschwandtner, P. M. Comparetti, G. Jacob, and C. Kruegel. Forecast: Skimming off the malware cream. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 11–20, 2011.

[78] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: Retrofitting Android to protect data from imperious applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 639–652, New York, NY, USA, 2011. ACM.

[79] 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 realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI '10, pages 1–6, 2010.

[80] L. K. Yan and H. Yin. DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security '12, pages 29–29, 2012.

[81] A. P. Felt, S. Egelman, and D. Wagner. I've got 99 problems, but vibration ain't one: A survey of smartphone users' concerns. In *Proceedings of the 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 33–44, 2012.

[82] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the 8th Symposium on Usable Privacy and Security*, SOUPS '12, 2012.

[83] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, 2014.

[84] Richard O. Duda and Peter E. Hart. *Pattern Classification and Scene Analysis*. John Wiley, New York, 1973.

[85] H. Peng, F. Long, and C. Ding. Feature selection based on mutual information: Criteria of max-dependency, max-relevance, and min-redundancy. *IEEE Transactions on Pattern Analysis Machine Intelligence*, 27:1226–1238, 2005.

[86] J. Ma, L. K. Saul, S. Savage, and G. M. Voelker. Beyond blacklists: Learning to detect malicious web sites from suspicious urls. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 1245–1254, 2009.

[87] K. Rieck, T. Krueger, and A. Dewald. Cujo: Efficient detection and prevention of drive-by-download attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 31–39, 2010.

[88] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society, Series B*, 58:267–288, 1994.

[89] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 101–112, 2012.

VITA

VITA

Lei Cen was born in Beihai, a coast city in Guangxi province in China. After high school, he went to Fundan University in Shanghai and studied computer science from 2004 to 2008. After recieving his B.S. degree in computer science, he started as a M.S. student in Fudan University and got his M.S. degree in computer science in 2011. He then came to Purdue University and received his Ph.D. degree in computer science in August 2016.