2018

Smartphone User Privacy Preserving through Crowdsourcing

Bahman Rashidi
Virginia Commonwealth University

SMARTPHONE USER PRIVACY PRESERVING THROUGH CROWDSOURCING

A Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

by

BAHMAN RASHIDI

August 2014 to June 2018

Director: Dr. Carol Fung,

Assistant Professor, Department of Computer Science

Virginia Commonwealth University

Richmond, Virginia

June, 2018

Acknowledgements

I would like to express my special appreciation and thanks to my advisor Dr. Carol Fung, you have been a tremendous mentor for me. I would like to thank you for encouraging my research and for allowing me to grow as a research scientist. Your advice on both research as well as on my career have been invaluable. I would also like to thank my committee members, Dr. Vivian Motti, Dr. Wei Cheng, Eyuphan Bulut and Dr. Kun Sun for serving as my committee members even at hardship. I would like to thank to my family and my beloved wife, Zhuo. Thank you for supporting me for everything, and especially I can't thank you enough for encouraging me throughout this experience.

TABLE OF CONTENTS

Chapter Page

List of Algorithms

LIST OF TABLES

LIST OF FIGURES

Abstract

SMARTPHONE USER PRIVACY PRESERVING THROUGH CROWDSOURCING

By Bahman Rashidi

A Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

Virginia Commonwealth University, 2018.

Director: Dr. Carol Fung,

Assistant Professor, Department of Computer Science

In current Android architecture, users have to decide whether an app is safe to use or not. Expert users can make savvy decisions to avoid unnecessary private data breach. However, the majority of regular users are not technically capable or do not care to consider privacy implications to make safe decisions. To assist the technically incapable crowd, we propose a permission control framework based on crowdsourcing. At its core, our framework runs new apps under probation mode without granting their permission requests upfront. It provides recommendations on whether to accept or not the permission requests based on decisions from peer expert users. To seek expert users, we propose an expertise rating algorithm using a transitional Bayesian inference model. The recommendation is based on aggregated expert responses and their confidence level. As a complete framework design of the system, this thesis also includes a solution for Android app risks estimation based on behaviour analysis. To eliminate the negative impact from dishonest app owners, we also proposed a bot user detection to make it harder to utilize false recommendations through bot users to impact the overall recommendations. This work also covers a multi-view permission notification design to customize the app safety notification interface based

on users' need and an app recommendation method to suggest safe and usable alternative apps to users.

CHAPTER 1

INTRODUCTION

Mobile apps have brought tremendous impact to businesses, social, and lifestyle in recent years. Various app markets offer a wide range of apps from entertainment, business, health care and social life. Android app markets, which share the largest user base, have gained a tremendous momentum since its first launch in 2008. According to the report by Android Google Play Store, the number of apps in the store has reached 2.2 million in June 2016, surpassing its major competitor Apple App Store [75]. The rise of Android phones brought the proliferation of Android apps, resulting in an ever-growing application ecosystem [6].

As users rely more on mobile devices and apps, the privacy and security concerns become prominent. Malicious third-party apps not only steal private information, such as contact list, text messages, online accounts, and location from their users, but also cause financial loss to users by making secretive premium-rate phone calls and text messages [68]. At the same time, the rapid growth in the number of apps makes it impractical for app market places, such as Google App Store, to thoroughly verify if an app is malicious or not. As a result, mobile users are left to decide whether an app is safe to use or not. This approach leaves little obstacle for malicious apps to be installed by users. In addition, unlike iOS, Android device owners do not have to root or "jailbreak" their devices to install apps from “unknown sources”. This gives Android users broad capability to install pirated, corrupted or banned apps from Google Play simply by changing a systems setting. This provides further incentive for the users to install third-party applications from various (potentially untrusted) app markets [3, 4, 2], but exposes their privacy to significant security risks [5].

More specifically, in the current Android architecture, users have to decide what re-

sources are given to an app at installation time. Unauthorized communications among apps are prohibited. However, such permission control mechanism has been proven to be ineffective in protecting users from malicious apps. A study shows that more than 70% of smartphone apps request to collect data irrelevant to the main function of the app [1]. Among the 1.4 million apps in Google Play, a significant percentage of them have permissions going beyond the apps' intended use. The situation is even worse in the third-party markets which are also available to Android users. In addition, such study shows that only a very small portion (3%) of users pay attention and make correct responses to requests for resource permission at installation, since they tend to rush through to get to use the application. The current Android permission warnings do not help most users make correct security decisions [32].

1.1 Android Privacy Preserving

Realizing these shortcomings in the current Android architecture, several efforts have been made to address the problems. Many resource management systems are proposed such as in [53, 58, 47]. Going down to the system level, L4Android [43] isolates smartphone OS for different usage environments in different virtual machines (VMs). QUIRE [26] provides a set of extensions addressing a form of attack, called *resource confused deputy* attacks, in Android. However, such approaches are not efficient since users are either not paying attention to permissions being requested or not aware of the permissions' implications. Hence, no mechanism that assumes users to have high technical and security knowledge will be usable for a wide audience.

As pointed out in [32, 31], the reasons for the ineffectiveness of the current permission control system include: (1) inexperienced users do not realize resource requests are irrelevant and could compromise their privacy, (2) users have the urge to use the app and may be have to give up their privacy in order to use the app. To address this problem, we propose

DroidNet, a framework to assist mobile users in controlling their resource usage and privacy through crowdsourcing. First, the framework allows users to use apps without having to grant all permissions. Second, DroidNet allows one to receive help from expert users when permission requests appear. Specifically, DroidNet allows users to install untrusted apps under a *"probation"* mode, while the trusted ones are installed in normal *"trusted"* mode. In probation mode, users make real-time resource granting decisions when apps are running. The framework facilitates a user-help-user environment, where expert users are identified and their decisions are recommended to inexperienced users.

Exploring user perceptions of privacy on smartphones using crowdsourcing has already been investigated. Agarwal et al. propose PMP [8] which collects users' privacy protection decisions and analyses them to recommend them to other iOS users. However, their recommendations are based on simple *majority voting* which results in high false recommendation rates.

Liu et al. investigated people's privacy preferences by capturing apps logs and analyzing them to identify a small number of profiles that simplify decision makings for mobile users [46]. Profiles were mined from logs by using SVM techniques. However, they do not include users' expertise in their study and this might cause false recommendation.

Lin et al. investigated the feasibility of identifying a small set of privacy profiles to help users manage their privacy profiles [45]. Instead of relying on smartphone users' decisions on permission requests, they identified the privacy profiles using Androguard, a static code analysis tool. They analyzed the purpose for which an app requests a permission and identified the permissions that satisfy the least privilege policy. Thus, they can find a set of necessary permissions for apps.

Liu et al. proposed PriWe in which they crowdsource users' decisions on permission requests and identify users' expectations [49]. In their work, they focus on finding users with similar responses to permission requests. After finding similar users and applying

a recommendation algorithm they identify some privacy profiles and recommend them to those who have similar strategy for responding to permission requests.

Ismail et al. propose a crowdsourcing solution to find a minimal set of permissions that will preserve the usability of the app for diverse users [39]. Their approach has a few shortcomings. Repackaging apps for all possible permission combinations is not practical. Also their inability to differentiate between inexperienced and malicious users makes their recommendations of limited quality. Yang et al. [84] propose a system to allow users to share their permission reviews with each other. Users leave comments on permissions and the system ranks reviews and recommends top quality reviews to users. Shahriyar proposes Gort [11], an analysis technique that analyzes app behavior while taking into account the context and semantics of the app. Gort uses a three-phase crowd analysis approach, in which crowd workers are asked whether it makes sense for the application to use its requested resources and tasks. App Ops [10], a feature in Android v4.3, allows users to selectively disable permissions for apps on their phones. However, Google removed this feature in their next update, reporting that it was experimental and could cause apps to behave in unexpected ways.

The common feature of those approaches is that they do not consider users' expertise in privacy profiling or permission recommendations. In contrast, considering the fact that most users are inexperienced, we proposed an expertise ranking algorithm to evaluate the expertise level of users for higher quality recommendations.

1.2 Risk Analysis and Malware Detection

In this section of the thesis, we propose two malware detection and app risk assessment models. The proposed models help the crowdsourcing framework to assess the maliciousness of apps in cases when the crowdsourcing model is unable to generate recommendations for users. This happens when there is not a sufficient amount of information about

apps to make recommendations. Therefore, the proposed models in this section, will enable the framework to help users for those apps.

1.2.1 Risk Assessment

Depending on the technology used, malware detection techniques can be divided into static and dynamic methods, where the former focuses on static code analysis of apps' and the latter investigate apps' maliciousness at runtime [30, 29, 74]. One major drawback of static analysis is that it does not detect vulnerabilities introduced at runtime [37, 28]. Dynamic analysis identifies vulnerabilities at runtime and supports the analysis of applications without actual code. It also identifies vulnerabilities that might have been false negatives in static code analysis [15, 29]. In this project, we proposed an Android app risk assessment approach to measure the risk of those apps that are newly published and the recommendation system does not cover. We analyze Android apps' behaviors as they run on the device, and propose XDroid, a dynamic analysis method based on Hidden Markov Models (HMM) [42]. A HMM engine can be used to model the runtime behaviors of an app, including malicious and normal ones.

In our approach, we consider other inputs such as API calls, time, ads libraries, and sensitive permission requests to build a comprehensive HMM. We discovered that the introduction of the time feature significantly improves the detection accuracy of the model. In our approach, we first log apps' behaviors through an instrumentation tool that we developed, by our research lab, called *DroidCat*. Then a filtering and parsing method is applied to syntheze and organize the captured behaviors. We train and test the HMM model using a dataset of known malicious apps and normal apps. Our experimental results demonstrate that our proposed model achieves high accuracy in detecting malicious apps.

Several research efforts have focused on the principles and practices of managing resource usage and Android security [53, 58, 25, 12] and privacy protection. Stochastic mod-

els are a powerful method to model security problems or defend against attacks. Solutions based on those models have been widely discussed in literature. Stochastic processes and specifically Markov chains have been extensively used to model security attacks. Although HMMs have been explored as technique for detecting malware for personal computers, to the best of our knowledge only few approaches [23, 76, 82, 21] have been proposed that use HHMs to address the mobile security issues, such as users' privacy preserving, malicious apps detection, and targeted malware.

Chen et al. [23] proposed a hidden Markov model to detect Android malicious apps at runtime. This approach is the closest to our model. Their approach is based on application intents passing through the Android binder only. However, only relying on the apps' intents as observations is not sufficient in order to decide whether the app is malicious or not. For example, they do not take into account the malicious API calls (Ads libraries), time, sensitive permission requests etc. This can be the reason why their detection accuracy is 67%. In contrast, we take all these features into account and achieve better accuracy. Additionally, the malware dataset we used to evaluate our approach is large (5560).

Xie et al. [82] propose a behavior-based malware detection system, called pBMDS, which adopts a probabilistic approach through correlating user inputs with system calls to detect anomalous activities in mobile phones. They mainly focus on recognizing non-human behavior instead on known behavior of malicious apps in order to detect applications that misuse the SMS/MMS and email services. In order to detect behavior anomalies, they analyze a series of GUI interactions, such as keypad interactions, between the user and the device. To evaluate their approach, they do not apply the technique to Android platform and use Linux-based smartphones instead. They only use three ad-hoc malicious apps for evaluating the technique, while we used a dataset with 5560 malwares. Additionally, they do not evaluate the proposed technique with goodware to understand how many benign apps are mistakenly recognized as malware, while we evaluated our model also on benign

apps.

Canfora et al. [21] propose a static analysis technique based on opcodes sequences to detect malware. The main difference with our model is that they use a static model, which consists of discrete wavelet transform (DWT) to segment the files and a distance function (edit distance) to compute the similarity of apps.

Suarez-Tangil et al. [76] proposed a stochastic model to address targeted malware. The context where a malicious behavior takes place plays a key role in their approach. In order to capture how the users interact with an app or a set of apps, they rely on a discrete-time first-order Markov process.

The major difference between our model and the existing ones is that they do not use a comprehensive app behavior analysis in their model training. The definition and discovery of proper observations, such as apps' intents, API calls, and time-stamp, make our model a unique solution in terms of Android malware detection. In addition, our model updates the model's parameters dynamically based on users' preferences through a self-train strategy. To the best of our knowledge, this the first model to help users through risk alerts generated by a well-trained HMM model and gets updated in a real-time manner.

1.2.2 Malware Detection

In general, malware detection techniques include *static analysis* and *dynamic analysis*. Analyzing Android apps' codes and the *Control Flow Graph* (CFG) is the key part of every static-based malware analysis. Using the CFG analysis, the model is able to find the malicious API calls and put a set of predefined restrictions on them as well as system calls [29, 63]. Since static-based malware analysis only focuses on the apps' code including API and syscalls, it is not able to detection malicious behaviors happening at the runtime [28]. In contrast, dynamic analysis captures all the runtime activities of apps and run a deep analysis on them to detect the malicious behaviors and activities during the

runtime [29]. In this paper, we study Android apps' behaviors as they run on the device, and propose an active learning method using Support Vector Machines (SVM). SVM can be used to classify apps in order to distinguish between malicious and benign ones. The active learning method empowers the model to retrain itself at a low cost.

In our approach, we not only consider apps' activities such as API cals and syscalls, but also the time that the activities occur. This can help to build a comprehensive set of features to be used as our training set. By introducing the time as a feature (timestamp of activities), we notice that this can significantly enhance the malicious app detection accuracy of the proposed model. The first step of our model is to log apps' activities using our own developed instrumentation tool called *DroidCat*. After capturing the logs a filtering and parsing mechanism is applied to syntheze and clean the captured activity logs. We train the SVM model and test it using a dataset of known malicious and benign apps. Our experimental results demonstrate that our proposed model achieves high accuracy in detecting malicious apps.

Several research efforts have focused on the principles and practices of managing resource usage and Android security [53, 25] and privacy protection [8, 66, 64]. Machine learning has been shown to be a powerful method to model security attacks or defend against attacks. The models are designed to detect and classify malicious activities. These machine learning solutions have been widely discussed in literature. As a machine learning model SVMs have been extensively used to model security attacks. In the last two years, a few approaches [52, 70, 80, 55, 56] have been proposed using SVM to address the mobile security problems such as assuring users' privacy, and detecting malicious apps, and targeted malware.

Nissim et al. [56] proposed a machine learning model, called ALDROID, to detect Android malicious apps at runtime. The proposed model is the closest one to our model. The model is based on active learning and updates the machine learning model periodically.

ALDROID uses active learning to reduce the labeling efforts of security experts, and enables a frequent process for enhancing the framework's detection model. ALDROID uses exploitation as query strategy. The model is trained by features that are based on the application's static code analysis. To construct the training dataset, they extracted requested permissions from `AndroidManifest.xml` as features. They claim that their model is behavioral-based. However static knowledge does not represent the actual apps' behaviors. In contrast, our model captures actual apps' behaviors and thus has higher accuracy and robustness. In addition, we conducted a set of comprehensive experiments to evaluate the stability and reliability of our model.

Narayanan et al. [55] proposed an adaptive Android malware detection solution, called DroidOL, based on online learning. The model aims at updating the learning model to be able to address the *malware population drift*. The training dataset was collected manually from official and unofficial app markets such as Google Play, FDroid, and SlideMe. To build the ground truth the collected apps were labeled using *VirusTotal*. They assumed that the model receives labeled data, which is not inter-procedural control flow sub-graph (static analysis). Also their model is based on a shakey ground, which means they assume that the model receives labeled data periodically and retrains the model. These all can be the reasons that the accuracy of the model is low. In contrast, our model uses active learning to handle the new unlabeled data.

Sahs et al.[70] proposed a machine learning based system for the detection of malware on Android devices. The training set of their model is limited to only the permissions (built-in and non-standard), and control flow graphs (CFGs) of the input applications. They evaluated the model using 91 malware apps, which is a much small set of apps. They did mention the ground truth for their dataset, but relying on static information such as CFGs and permissions list cannot be a strong metric.

The major difference between our proposed model and the existing ones is that they

do not use a comprehensive app behavior analysis in their model training. The definition and discovery of proper observations, such as apps' intents, API calls, and time-stamp, make our model a unique solution in terms of Android malware detection. In addition, our proposed model updates the model's parameters using active online learning as its query strategy. To the best of our knowledge, ours is the first model to detect malware using a real-human app instrumentation data and active learning.

1.3 Bot User Detection

Providing a user-help-user environment, where expert users' decisions are recommended to inexperienced users is the main goal of our recommendation system. Therefore, the decisions from expert users determine what recommendation the recommendation system will provide to inexperienced users. However, this also opens the door for malicious/dishonest users to misguide the recommendation system's recommendations. For instance, the developer of malicious applications can create/employ many dummy users and gain high level of expertise through responding to other apps. However, those dummy users send dishonest responses to targeted requests from malicious apps to mislead the recommendation from the recommendation system. A Sybil detection function may be able to detect some dummy users for the recommendation system system. However, a sybil detection function may not discover all dummy users and they are still influential when attackers are savvy enough to evade the detection system. Studying what the attacker can do to make maximum profit through malicious user attack and what the recommendation system can do to minimize the damage caused by attack is our focus in this proposed game model.

In order to analyze the attackers' strategies and actions in the interaction with the recommendation framework, we use a game-theoretical model to analyze the behavior and strategies from both users including attackers and the recommendation system. More specifically, a static Bayesian game [38] is used and Nash equilibria of the game are inves-

tigated. In this Bayesian game the defender does not know the type of its opponent(regular or malicious) so it is also an incomplete information game [33]. Through this Bayesian game formulation, we are able to help the recommendation system select best strategies to play against the attacker and minimize the potential damage caused by attackers (malicious users).

Liu et al. [48] proposed a game-theoretical approach to model the interactions between a DDoS attacker and a network administrator (system). They model the network and infer the attackers' intents, objectives, and strategies to observe the importance of modeling and its effects on risk assessment and harm prediction. Jormokka et al. [40] presented a few examples of static game models with complete information (players' information) where each example represents an information warfare scenario. Lye et al. [50] and Alpcan et al. [9] proposed two game-theoretical solutions to analyze the interactions between malicious attackers of system, IDS, and security of a computer network. In both works, they focus on the existing network parameters and the interactions between attacker and defender (system). Zhu, et al. [88, 87] proposed game-theoretical models to incentivize collaboration in intrusion detection networks (IDN). In this work, a game-theoretical model is proposed to analyze the behavior of IDSs in network and incentivize their participation by strategical game policy design. Due to the complexity of attackers' activities, many efforts have been proposed towards the risk assessment, modeling the attackers' activities (behavior), and cybersecurity strategies [22, 81, 85, 34].

In spite of the existing similarities between the our recommendation system and its architecture and related proposed works such as having an attacker in one side and a defender on the other side, there are differences, which seem to be significant and make these proposed models inapplicable to the recommendation framework. For example, interaction between the recommendation framework and users is different and more complicated than these models. Therefore, due to this inconsistency between the proposed models and the

recommendation framework, we need to design a game model, which is more consistent to the recommendation features. The proposed model should be able to model the interactions (request/response) between the game's players.

1.4 Smartphone Permission Notification

Another important problem of the privacy preserver system is how to present the privacy and permission risks to users. The big variation of users' knowledge about apps privacy risks turns the problem into a challenging one. In the case of Android privacy, lack of experience and knowledge raises similar concerns. Therefore, facilitating the process of informing users about the risk of apps is a key challenge. We aim at designing and developing a permission privacy risk notification mechanism that helps users understand privacy risks regardless of their background and experience. The mechanism takes usability, reliability and performance into account.

In order to achieve such model, considering users' knowledge and interface preference, our model generates multiple views of the security and privacy risk related to app permissions and resource usage. Users have the choice to select a view which they prefer the most. After selecting a view, users will see the privacy and security risk related to their app through the interface/view they prefer. We conducted a user study to evaluate the usability of the model. Our study results show that users are significantly interested in using our proposed model. It is worth noting that users' interests are almost equally distributed among the views. This proves that users with different backgrounds and preferences are interested in different views.

1.5 Application Recommendation System

In current Android systems, the application recommendation function is an important feature that users can find a similar and secure application to replace a risky one. The

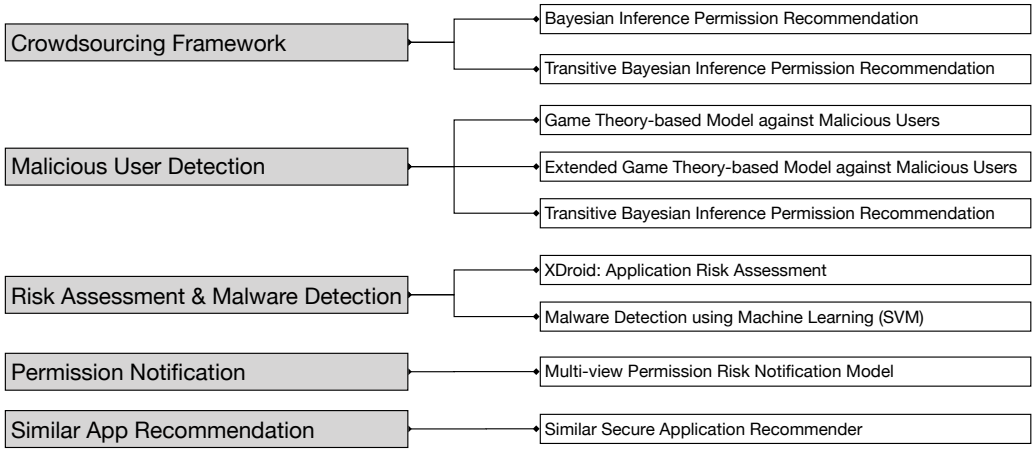


Fig. 1.5.1.: Thesis sections structure.

current recommendation system provided through Google and the Google Play store presumably recommends applications similar to a target application while accounting for the popularity of each application. However, it does not take the security feature of each application or users preferences into consideration. In this project, we aim at proposing an app recommendation system that provides users with fine-grained and customizable application recommendations. The app recommendation system considers other metrics such as popularity, security, and usability. More specifically, the app recommender system provides an interface for users to configure the weight on each metric and a recommendation algorithm that generates a list of recommended applications based on the combined scores.

Figure 1.5.1 shows an overview of the sections of the thesis. For each section of the project, there are at least one proposed solution. For example, as you can see, we proposed three malicious user and bot detection models for section three.

CHAPTER 2

ANDROID CROWDSOURCING-BASED PRIVACY PRESERVING

In this project, we study the security and privacy issues of the Android OS. We review the existing proposed solutions and approaches toward addressing the issues. As the main part of this project, we present a permission control framework that assists inexperienced users to make a low-risk decision as for whether a permission request should be granted or denied. We elaborate the evolution of the project from the preliminary idea to the most recent version of the framework.

2.1 Preliminary Crowdsourcing Model

This project is to provide a framework which allows users to install untrusted apps under a *"probation"* or a "trusted" mode. In the probation mode, users make real-time permission control during an app's running time. The framework also facilitates a user-help-user environment, where expert users' decisions are recommended to inexperienced users. The framework provides the following functionalities:

- Two app installation modes for any new apps: *trusted mode* and *probation mode*. In probation mode, an app requests permission from users to access sensitive resources (e.g. GPS traces, contact information, friend list) when needed. In trusted mode, the app is fully trusted and all permissions are all granted.

- A recommendation system to guide users with permission granting decisions, by serving users with recommendations from expert users on the same apps.

2.1.1 Problem Definition

The rapid growth of smartphone application market raises security and privacy concerns regarding untrusted applications. Studies have shown that most apps in markets request to collect data irrelevant to the main functions of the apps. Traditional permission control design based on one-time decisions on installation has been proven to be not effective to protect user privacy and poorly utilize scarce mobile resources (e.g. battery). In this work, we propose RecDroid, a framework for smartphone users to make permission control in real time and receive recommendations from expert users who use the same apps. This way users can benefit from the expert opinions and make correct permission granting decisions. We describe our vision on realizing our solution on Android and show that our solution is feasible, easy to use, and effective.

2.1.2 System Design

Our general approach is to build RecDroid with four functional processes, of which two are on mobile clients and the remaining two are on remote servers. On the client side, RecDroid allows users to install apps in a permission tracking mode, and provides recommendations to users on resource permission requests from mobile apps. On the server side, collected users responses are processed and mined to extract safe responses. In particular, RecDroid (1) collects users permission-request responses, (2) analyzes the responses to eliminate untruthful and biased responses, (3) suggests other users with low-risk responses to permission requests, and (4) ranks apps based on their security and privacy risk level inferred from users' responses. Figure 2.1.1 shows an overview of RecDroid service, which is composed of a *thin OS patch* allowing mobile clients to automatically report users responses to and receive permission request response suggestions from a *RecDroid* service.

Before going into further details about individual components, we first describe the

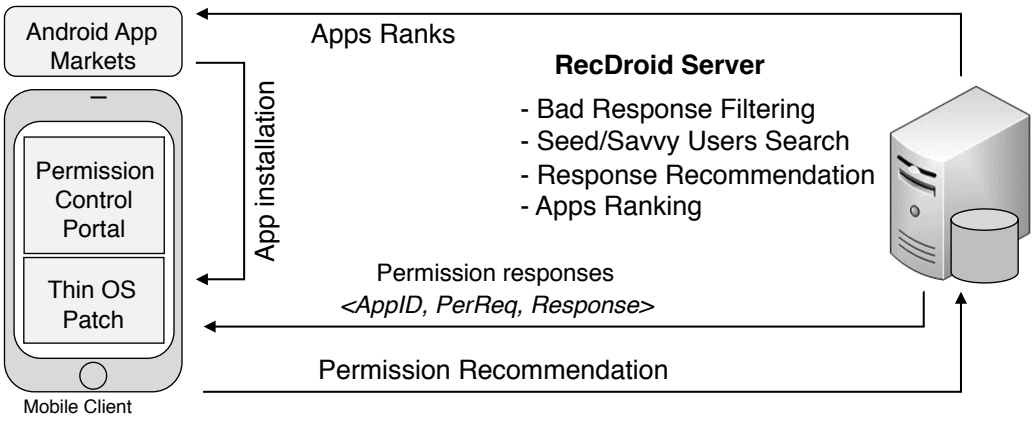


Fig. 2.1.1.: RecDroid Service Overview

permission handling procedure during an installation a mobile app with RecDroid. When a user first downloads and installs an application from app stores, the installer requests permission to access resources on the device. Instead of being sent to the operating system's legacy permission handler (e.g. Package Manager Service on Android) , the requests pass through RecDroid checks, which are installed on the mobile client at OS level. Figure 2.1.2 illustrates the permission checking and granting flow on RecDroid. In the first installation of an app, RecDroid allows the app to be installed on one of the two modes: *Trusted* and *Probation* (tracking mode), as shown in Figure 2.1.3(a). All requested permissions will be permanently granted to the app as specified by the user, if *Trusted* mode was selected. Otherwise, in *Probation* mode, RecDroid will compare the requested permissions with a predefined list of critical permissions that is of concern to users, such as location access, contact access, and messaging functions, etc. Regarding the installation mode selection, RecDroid also recommends an installation mode to the users based on collected data. For example, for new apps and apps that frequently receive rejections on permission requests, a probation installation mode should be recommended to users. With critical permissions, RecDroid client queries the online RecDroid service to get response recommendations for the permissions, specifically for the apps to be installed. Upon receiving the answer from

the recommendation service, RecDroid client pops up a request, combined with the suggested response, to the user, as shown in Figure 2.1.3(c). Based on the suggested response, the user decide to grant or deny permission to access to certain resource. If a user chooses to deny a request, dummy data or *null* will be returned to the application. For example, a denied GPS location request could be responded with a random location. That decision is both recorded in RecDroid client and populated back to RecDroid server for app ranking and analysis. Only then, the request is forwarded to legacy permission handler for book keeping and minimizing RecDroid's unexpected impact on legacy apps. It is important to note that this process only happens once, when the app is first installed. Later, after collecting users responses and preferences; and having a security and privacy ranking of the app, RecDroid server should decide and notify RecDroid client whether to pop up permission requests or automatically answer them based on prior knowledge. Therefore, RecDroid strives to achieve a balance between the fine-grained control and the usability of the system.

In order to realize RecDroid service, four main challenges need to be addressed.

- How do we instrument the operating system to intercept resource requests with the minimum amount of changes to the system such that it does not affect normal and legacy operations of the device? At the same time, how do we make that instrumentation work on both existing legacy apps and coming apps?

- Given that the responses from users are subjective, could be biased, and even malicious, how do we design the recommendation and ranking system that could detect and then filter out untruthful or biased responses?

- How do we bootstrap and expand the recommendation system? Since this is a participatory service, it is important to have a sustainable and scalable approach that could provide valuable recommendations to all applications. It is certainly a challenging

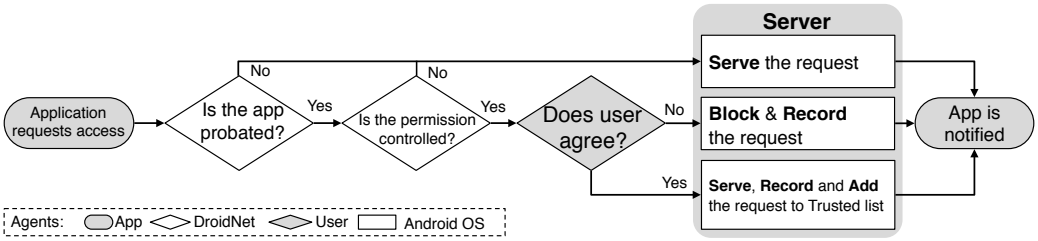
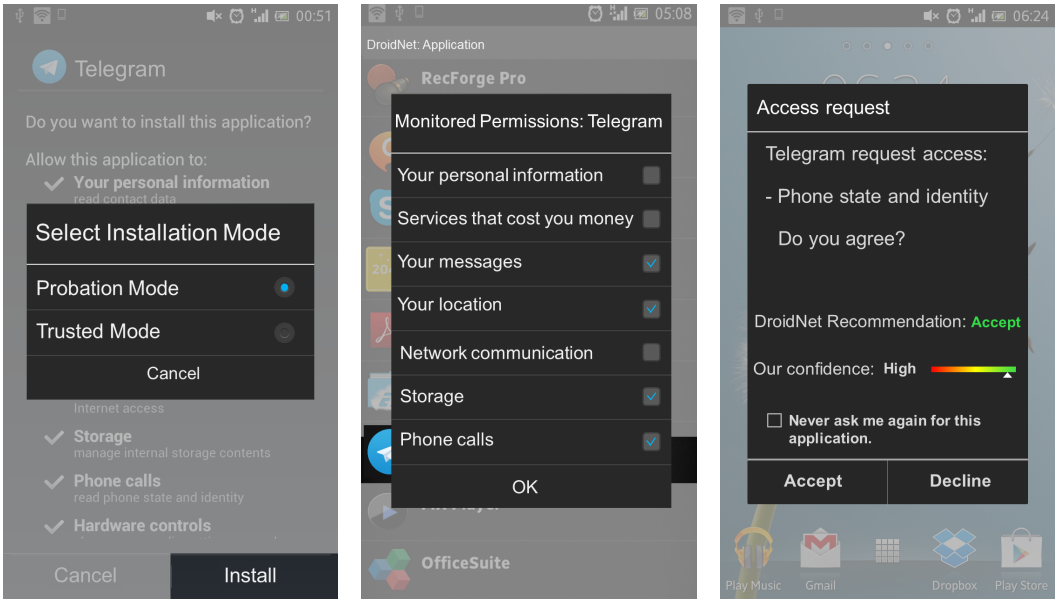


Fig. 2.1.2.: Permission request flow in RecDroid



(a) (b) (c)

Fig. 2.1.3.: An example of DroidNet on Telegram app: (a) probation and trusted installation modes; (b) users pick which critical resources to be monitored; (c) pop-up for permission granting with suggestion from DroidNet and its confidence.

mission given millions of apps out there on different apps market places, and more to come.

Next, we will provide our vision on addressing the above mentioned challenges. At a high level, we propose a participatory framework to collect and analyze users' response to provide recommendations to users and rank apps based on the plausibility of their permission requests. In our proposed system, a *Permission Control Portal* installed on the mobile devices intercepts apps' permission requests, records the requests, and collect users' re-

sponse to the requests.

Intercepting permission requests: Since intercepting permission requests requires OS level access, we create a small software patch to modify client's operating system. We investigated different potential approaches to perform OS modification and designed a solution that causes minimum impact to legacy apps and applicable to a broad range of OS versions, hardware platforms, and permission access models. One might argue that collecting this permission granting behaviors from users could raise privacy concern. However, since the portal does not collect any actual sensitive information, the data it collects doesn't contain private information. In fact, the portal merely communicates three-tuple data in the form of *<AppID, Permission Request, User's responses>*.

Bootstrapping the service: In order to suggest plausible responses to users, RecDroid starts from a set of *seed expert users* and make recommendation based on their responses. However, it is impractical to have our expert users to provide plausible responses to hundreds of thousands of apps available on the market. To address this scalability challenge, we propose a spanning algorithm that searches for *external expert users* based on the similarity of their responses to our set of internal experts, in combination with the user's accumulative reputation. Our recommendation for an app is based on the average of top N expert users in combination with the response that is selected by majority of participants. Having the same nature as the spanning algorithm described in [67], RecDroid spanning algorithm is sketched as follow.

Algorithm 1 describes our external expert users search method. In its simplest form, if the similarity of unguided (without recommendation) responses from a user and the responses from our seed experts to the same app is high, then we recruit that user into the spanned expert users list. This base method could be further improved by understanding the criticality of a given user by looking at their responses to permission requests across multiple apps. For example, a response from a user that always says "Accept" to all per-

mission requests might not be weighted as high as that of a user that has a diverse set of positive and negative responses. It is certainly true that the more responses our seed expert user send to the system, the more spanned expert users we can find.

Algorithm 1 Seek spanned expert users

Notations :
U :the set of all users
E_i :the set of initial seed expert users
E_s :the set of spanned expert users
for each user u in $U \setminus E_i$ **do**
    if The percentage of matching *unguided* responses from u and E_i is higher than a threshold θ and the number of matching samples is higher than τ **then**
        $E_s = E_s \cup u$
    end if
end for

Table 2.1.1 shows the permission decision table for the recommendation system. When a resource access request pops up, our system searches for the response from our seed experts to the same request first. If a match is found then send the response of the expert user as our recommendation and with high confidence. If only the response from a spanned expert user is found, then send the expert response as our recommendation with medium confidence, otherwise no recommendation is made.

Table 2.1.1.: Recommendation Decision Table

Top recommender	Recommendation	Confidence
Seed expert	Response from the expert	High
Spanned expert	Response from the expert	Medium
Others	No recommendation	-

2.1.3 Conclusion

We presented our vision for implementing RecDroid, a smartphone permission control and recommendation system which serves the goal of helping users perform low-risk resource accessing control on untrusted apps to protect their privacy and potentially improve efficiency of resource usages. We propose a framework that allows users to install apps in either trusted mode or probation mode. In the probation mode, users are prompt

with resource accessing requests and make decisions to grant the permissions or not. RecDroid also provides expert recommendations on permission granting decisions to reduce the user's risk of making false decisions. We implemented our system on Android phones and demonstrate that the system is feasible and effective.

2.2 Bayesian Inference-based Android Resource Access Permission Recommendation with RecDroid

In this section, we use a Bayesian Inference model to improve our preliminary mode. The contributions of this project are as follows:

- Two app installation modes for apps that is about to be installed: *trusted mode* and *probation mode*. In probation mode, at run time, an app has to request permission from users to access sensitive resources (e.g. GPS traces, contact information, friend list) when the resource is needed. In trusted mode, the app is fully trusted and all permissions are all granted.
- An architecture to intercept and collect apps' permission requests and responses, from which recommendations are made as for what permission from which apps should and should not be granted.
- A recommendation system to guide users with permission granting decisions, by serving users with recommendations from expert users on the same apps.
- A user-based ranking algorithm to rank security risks of mobile apps.

2.2.1 Problem Definition

With the exponential growth of smartphone apps, it is prohibitive for apps market places, such as Google App Store for example, to thoroughly verify if an app is legitimate or malicious. As a result, mobile users are left to decide for themselves whether an app is

safe to use. Even worse, recent studies have shown that most apps in markets request to collect data irrelevant to the main functions of the apps, which could cause leaking of private information or inefficient use of mobile resources [1]. To assist users to make a right decision as for whether a permission request should be accepted, we propose RecDroid. RecDroid is a crowdsourcing recommendation framework that collects apps' permission requests and users' permission responses, from which a ranking algorithm is used to evaluate the expertise level of users and a voting algorithm is used to compute an appropriate response to the permission request (accept or reject). To bootstrap the recommendation system, RecDroid relies on a small set of seed expert users that could make reliable recommendations for a small set of applications. Our evaluation results show that RecDroid can provide high accuracy and satisfying coverage with careful selection of parameters. The results also show that a small coverage from seed experts is sufficient for RecDroid to cover the majority of the app requests.

2.2.2 Recommendation System Design

In RecDroid, the responses to permission requests from all users are logged by a central server and they can be used to generate recommendation to inexperienced users to help them make right decisions to avoid unnecessary permission granting. For example, if a restaurant finding app is requesting access to the users camera, then the request is suspicious and very likely will be declined by an expert user. The responses of expert users are then aggregated and the system would suggest other users of the same app not to accept the similar requests. However, how to find expert users and how to aggregate the responses from expert users are the focus of this section.

2.2.3 Rank RecDroid Expert Users

In this section, we investigate an algorithm to seek expert RecDroid users. Suppose we have a set of users U and among them set $E \subset U$ is a set of initial seed expert users. For instance, the security experts are employed by RecDroid to monitor the permission requests from apps. The seed experts respond to the permission requests from selected apps based on their professional judgment. Their responses are considered accurate and are used in the system as ground truth. However, due to limited budget and manpower, the number of apps that can be covered by seed experts is small, compared to the entire app base that RecDroid monitors. In this section, we use set A to denote all apps that are monitored by RecDroid.

Suppose initially each of our seed expert users E have responded to a set of apps of their choice. The apps responded by a seed expert user $e \in E$ is denoted by $A(e)$. Since there may be multiple permission requests popped up during an app usage, we use $R(a)$ to denote the set of requests the app $a \in A$ may have. We use R_e to denote all requests that are covered by RecDroid seed expert users, named *labeled requests*. Then we have,

$$R_e = \cup_{\forall e \in E} \cup_{\forall a \in A(e)} R(a) \tag{2.1}$$

How to determine whether a user is expert user or not? In our approach we propose a ranking algorithm to evaluate the expertise level of a user based on the ratio of correctness on his/her responses to app requests. Let p_i denote the probability that user i correctly responds to permission requests. Our mission is to estimate p_i based on the number of correct and incorrect responses that user i has responded in the past. Our approach is to observe all labeled requests that are independently responded (without recommendation) by the user, and compute the ranking of the user based on the number of correct/incorrect responses to those requests. For the convenience of presentation, we drop the subscript i in the rest of the notations.

We use notation α to represent the cumulative number of requests that are responded consistent with seed experts and β requests are responded opposite to the experts' advice (note that the labels from seed experts arrive later than the user's responses). Furthermore, we use a random variable $X \in \{0, 1\}$ to denote a random variable that a user answers the permission requests correctly or not. Where $X = 1$ represents that user responds to a request correctly, vice versa. Therefore, we have $p = \mathbb{P}(X = 1)$. Given a sequence of observations on X, a beta distribution can be used to model the distribution of p.

In Bayesian inference, posterior probabilities of Bernoulli variable given a sequence of observed outcomes of the random event can be represented by a beta distributions. The beta-family of probability density functions is a continuous family of functions indexed by the two parameters α and β, where they represent the accumulative observation of occurrence of outcome 1 and outcome 0, respectively. The beta PDF distribution can be written as:

$$f(p|\alpha, \beta) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} p^{\alpha-1}(1-p)^{\beta-1} \tag{2.2}$$

The above can also be written as,

$$p \sim \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} y^{\alpha-1}(1-y)^{\beta-1} \tag{2.3}$$

In our scenario, the parameters α and β are the accumulated number of observations that the user respond to the permission request correctly and incorrectly, respectively.

Let $x_n \in \{0, 1\}$ be the n_{th} observation in the past, where $n \in \mathbb{N}$. The accumulative observations of both correct and incorrect responses from a user after n observations can be written as,

$$\alpha_n = \sum_{k=1}^{n} q^{n-k} x_k + q^n C_0 \tag{2.4}$$

$$= x_n + q x_{n-1} + ... + q^{n-1} x_1 + q^n C_0$$

$$\beta_n = \sum_{k=1}^{n} q^{n-k} (1 - x_k) + q^n C_0 \tag{2.5}$$

$$= (1 - x_n) + q(1 - x_{n-1}) + ... + q^{n-1}(1 - x_1) + q^n C_0$$

Where C_0 is a constant number denoting the initial believe of observations; $q \in [0, 1]$ is the remembering parameter which is used to discount the influence from past experience and therefore emphasize the importance of more recent observations.

Equation 2.4 and 2.5 can also be written into an iterative form as follows:

$$\alpha_0 = \beta_0 = C_0 \tag{2.6}$$

$$\alpha_n = x_n + q\alpha_{n-1} \tag{2.7}$$

$$\beta_n = (1 - x_n) + q\beta_{n-1} \tag{2.8}$$

Let random variable Y represent the possible value that the true expertise level of a user can be, then we have,

$$\mathbb{E}[Y] = \frac{\alpha}{\alpha + \beta} \tag{2.9}$$

$$\delta^2[Y] = \frac{\alpha\beta}{(\alpha + \beta)^2(\alpha + \beta + 1)} \tag{2.10}$$

We use s to represent the expertise ranking of the user. Then we can compute s using the following formula,

$$
\begin{aligned}
s =& max(0, \mathbb{E}[Y_i] - t\theta\delta[Y_i]) \\
=& max(0, \frac{\alpha}{\alpha+\beta} - t\theta\sqrt{\frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}})
\end{aligned} \tag{2.11}
$$

Where $t \in [0, \infty)$ represents a conservation factor where a higher value means our ranking mechanism is more conservative to less confident estimation. $\theta = \sqrt{\frac{2-q}{1-q}}$ is the normalization factor, which is to decouple the forgetting factor and expertise rating.

We can see that the higher ratio a user responds to permission requests correctly, the higher ranking score it receives through RecDroid system; the more samples the system knows about the user, the higher score it receives.

Given the ranking scores of users, we can use a simple threshold τ to identify expert users from novice users. That is, if a user i has $s_i \geq \tau$, then it is labeled as an expert user. The ranking scores will be used to make recommendations to other app users when permission requests pop up.

In the next subsection, we propose an algorithm to generate recommendation on app permission requests based on existing responses from other users who have used the same app.

2.2.4 Response Aggregation through Weighted Voting

When a user receives a permission request from an app in probation mode, RecDroid system attempts to make a recommendation to the user regarding whether he/she should grant the request. If the app has been investigated by our seed expert users, then the response from the seed experts will be recommended to the user. However, due to the limitation of our seed experts, majority of apps may not be covered by seed experts. In this case, we aggregate the responses from other users and recommend the aggregated response

if confidence level is high enough.

The proposed approach in this model is called weighted voting. The voting process is divided into three steps: qualification, voting, and decision. The algorithm is described in Algorithm 2.

Algorithm 2 Weighted Voting for Recommendation Decision

This algorithm is to decide whether to make recommendation, and what recommendation to make given the response from other regular users.
Notations :
M :the set of users who have responded to the permission question
s_i :the ranking of the i_{th} user
x_i :the response of the i_{th} user
τ_e :the minimum required ranking score to be classified into expert users
τ_d :the recommendation threshold
a, b :the cumulative ballots for reception and rejection decision
D_0 :the initial ballot count for both decisions
//initialize voting parameters
$a = b = D_0$
for each user u in M **do**
    if $s_u > \tau_e$ **then**
        //only qualified users responses are considered into the voting
        if $x_u = 1$ **then**
            $a+ = s_u$
        else
            $b+ = s_u$
        end if
    end if
end for
//decision making based on final ballots result
if $\frac{a}{a+b} > 1 - \tau_d$ **then**
    Recommend to accept the request
else if $\frac{a}{a+b} < \tau_d$ **then**
    Recommend to reject the request
else
    No recommendation
end if

In the qualification step, only responses from qualified users are included into the voting process. Initially the ballot count for reception and rejection decisions are equally initialized to D_0. For each qualified voter, the weight of the cast ballot is the ranking score of the voter. After the voting process finishes, the average ballot score is used to make a final decision. If the average ballot score exceeds a decision threshold, then corresponding recommendations are made. Otherwise, no recommendation is made.

2.2.5 Experiments

To evaluate the performance of RecDroid, we conducted a set of simulation experiments to measure the accuracy, reliability, and effectiveness of the system.

2.2.6 Simulation Setup

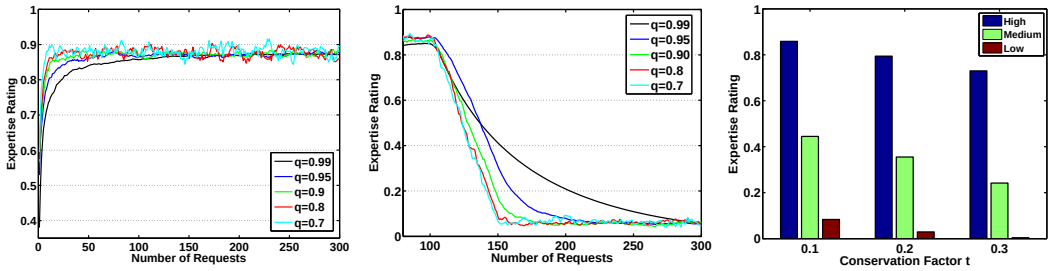
As a proof of concept we set up a RecDroid users profile to be a set of 100 users consisting of three different levels of expertise. Note that the expertise we refer here is the probability that a user answers permission requests correctly (a.k.a. consistent with standard answers). Among the 100 users, 40% are with a high level of expertise (0.9), 30% are with a medium level of expertise (0.5), and the remaining 30% are with a low level of expertise (0.1) . Unless particularly specified in the experiments, we fix the number of requests answered by users to 100.

Our simulation environment is MATLAB 2013 on a Windows machine with 2.5Ghz Intel Core2 Duo and 4G RAM. All experimental results are based on an average of 100 repeated runs with different random seeds.

2.2.7 Expertise Rating and the Impact of Parameters

The remembering parameter q (Equation 2.4) and conservation factor t (Equation 2.11) are two essential parameters that RecDroid uses for user expertise rating. In this experiment we study the impact from the two factors and determine the parameter choices for the rest of the experiments.

In the first experiment, we track the RecDroid expertise rating of a high expertise (0.9) with the number of labeled requests they have answered under different remembering factor settings. In the second experiment, we deliberately configure the user so it immediately turns to be malicious and gives opposite responses after the 100 honest requests.



(a) (b) (c)

Fig. 2.2.1.: Forgetting and Conservative factor: (a) Expertise level of users with different forgetting factor; (b) 100 percent of requests are answered; (c) Expertise level of users for different conservation factors

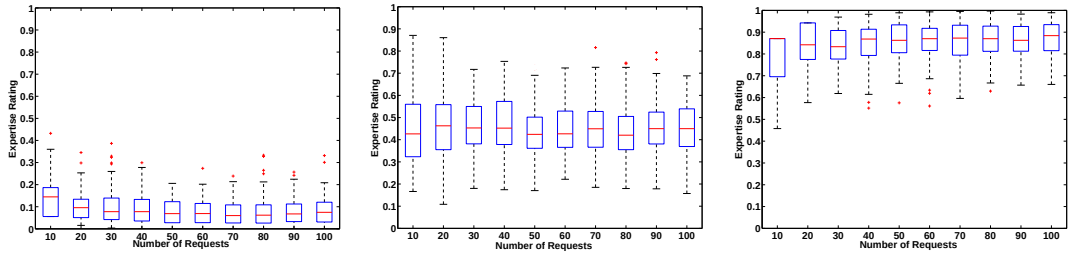
From Figure 2.2.1(a) we can see that with higher q setting, the curves are smoother. This is because a high q means the expertise rating largely depends on past accumulation, which brings stableness to expertise rating. However, from Figure 2.2.1(b) we can see that high q also represents less flexibility to sudden change. To leverage the pros and cons, we decide to fix $q = 0.9$ in the rest of this section.

In the third experiment, we track the expertise rating of high, medium, and low expertise users after 100 labeled requests under different t setting. Figure 2.2.1(c) shows that with higher t setting, the rating of all users are lower. We chose a moderate setting $t = 0.1$ in the rest of this section.

Figure 2.2.2 shows the expertise rating of the three types of users (with expertise $0.1, 0.5$, and 0.9). The blue boxes represents the central 50% of the expertise rating data while the red bars are the medium values of the samples. The vertical whiskers indicate the range of all data except outliers, which are represented by red crosses.

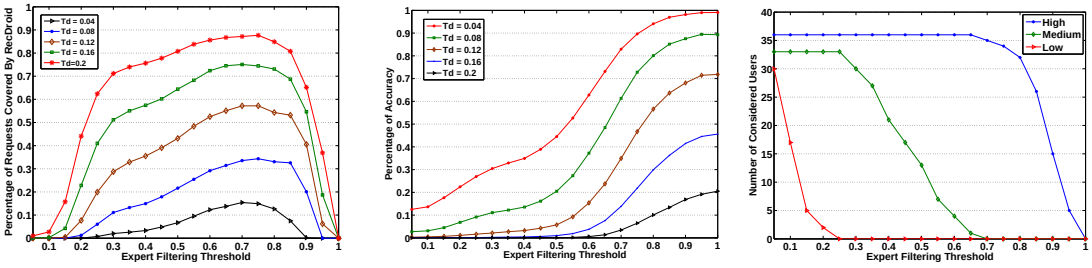
2.2.8 Coverage and Accuracy of RecDroid Recommendation

In this experiment we evaluate the performance of RedDroid recommendation by measuring its recommendation coverage and accuracy. We define *coverage* to be the percentage of the requests that RecDroid decides to give recommendation to users given the existing responses from users participating RecDroid. We define *accuracy* to be the percentage of



(a) (b) (c)

Fig. 2.2.2.: Expertise ratings of: (a) Low expertise nodes; (b) Medium expertise nodes; (c) High expertise nodes



(a) (b) (c)

Fig. 2.2.3.: Coverage and Accuracy: (a) The percentage of requests that RecDroid makes recommendation; (b) The percentage correct recommendations that RecDroid makes; (c) The percentage of users who pass expert filtering and participate into recommendation voting

correct recommendation that RecDroid makes. Note that if a request is covered by a seed expert, then RecDroid always recommend the response from the seed expert.

In the first experiment we investigate the scenario that 100 requests receive responses from all 100 users, except seed experts. RecDroid uses Algorithm 1 to determine whether to make a recommendation to new users or not and what recommendation it should make. Note that we assume all 100 users have received expert rating scores previously. We plot in Figure 2.2.3(a) and Figure 2.2.3(b) the percentage of requests (among 100) that RecDroid decides to make recommendation and percentage correct recommendations that RecDroid makes, under different τ_e and τ_d settings. We can see that with higher τ_d (which means wider acceptance range for the recommending decision, see Algorithm 1), the coverage increases while the accuracy decreases. This is because the more selective RecDroid is regarding the voting score results, the higher accuracy it achieves and less voting results

will be qualified for recommendation. We also notice that the accuracy increases with experts filtering threshold τ_e. However, with very low or very high τ_e, the coverage is low. This is because when all users are included in the decision process, the conflict of responses among users leads to low voting score and therefore RecDroid is less likely to make recommendations. On the other side, high filtering threshold causes few or no users are qualified to voting process, which also leads to no recommendation.

Figure 2.2.3(c) depicts the result on percentage of qualified users of the three types under different τ_e setting. We can see that with higher expert filtering threshold τ_e, less users can be involved in the decision process as described in Algorithm 1. Also the involving rate of low expertise users is lower than high expertise users.

Finally, we simulate a scenario that no users are rating previously and all users have responded to 100 request. As a coordinator of the RedDroid system, we hire a seed expert to respond to some application requests as ground truth, which will then be used to rate the expertise of other users. Regarding the requests not covered by the seed expert, RecDroid may provide recommendation based on the response from other users. We study the coverage rate by seed expert and the percentage of requests that are covered by RecDroid. As shown in Figure 2.2.4, the overall RecDroid recommendation rate increases with coverage rate from the seed expert. The linear line represents the coverage rate from the seed user. The difference between the overall coverage and seed expert coverage is called the *bonus coverage*. Higher bonus coverage represents a higher utilization of Recdroid. From an economic point of view, if we consider the coverage of seed expert brings cost to the RecDroid coordinator (since the seed expert is hired), then the bonus coverage brings saving to the coordinator. The decision makers can choose the optimal seed coverage based on its optimal profit. Note that in the above experiment, the values of simulation parameters are $\tau_e = 0.5$, $\tau_d = 0.2$, q=0.9, and t=0.1.

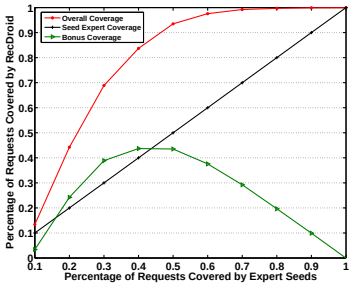


Fig. 2.2.4.: Coverage of overall requests vs. coverage of seed experts

2.2.9 Conclusion

In this section we presented RecDroid, an Android permission control and recommendation system which serves the goal of helping users perform low-risk resource accessing control on untrusted apps to protect their privacy and potentially improve efficiency of resource usages. We propose a framework that allows users to install apps in either trusted mode or probation mode. In the probation mode, users are prompt with resource accessing requests and make decisions to grant the permissions or not. Our RecDroid recommendation algorithm can effectively use crowdsourcing techniques to find expert users in the user base and provide recommendation based on the responses from expertise nodes. Our evaluation results demonstrate that RecDroid recommending system can achieve high accuracy and coverage when parameters are carefully selected. We also show that RecDroid only need a small seed expert coverage to bootstrap the system. We implemented our system on Android phones and demonstrate that the deployment of such system is feasible and effective.

2.3 Android Permission Recommendation Using Transitive Bayesian Inference Model

To support this user-help-user environment, an effective expert user seeking is the major challenge. DroidNet starts from a small set of trusted expert users (seed users) and propagates the expert evaluation using a transitional Bayesian learning model. We evaluate

the effectiveness of the model through simulation and survey data from real users. The major contributions of this project include:

- A comprehensive Android permission control framework to facilitate a user-help-user environment in terms of permission control.
- A novel transitive Bayesian inference model to propagate expertise rating of users in a network through pairwise similarity among users.
- A low-risk recommendation algorithm which can help inexperienced users with permission control decision making.
- A prototype implementation of the system and real user evaluation on the usability of the system.

2.3.1 Problem Definition

In current Android architecture, users have to decide whether an app is safe to use or not. Expert users can make savvy decisions to avoid unnecessary privacy breach. However, the majority of normal users are not technically capable or do not care to consider privacy implications to make safe decisions. To assist the technically incapable crowd, we propose DroidNet, an Android permission control framework based on crowdsourcing. At its core, DroidNet runs new apps under probation mode without granting their permission requests up-front. It provides recommendations on whether to accept or reject the permission requests based on decisions from peer expert users. To seek expert users, we propose an expertise ranking algorithm using a transitional Bayesian inference model. The recommendation is based on the aggregated expert responses and its confidence level. Our simulation and real user experimental results demonstrate that DroidNet provides accurate recommendations and covers the majority of app requests given a small coverage from a small set of initial experts.

2.3.2 Expert Users Seeking

The key challenge of DroidNet is to seek experts from the regular users in the DroidNet. In DroidNet users' responses to permission requests are recorded by a central server and the responses from expert users are used to generate recommendations to help inexperienced users make low-risk decisions.

DroidNet starts from a small set of trusted seed expert users, and propagate the expertise evaluation based on similarity among users using a transitive Bayesian inference model [67]. This section describes the model in more detail.

2.3.2.1 Assumptions and Notations

DroidNet can be seen as a network $\mathcal{G} = \{s \cup \mathcal{U}, \mathcal{E}\}$, which consists of a *seed expert* s, a set of n regular users $\mathcal{U} = \{U_1, U_2, \cdots, U_n\}$, and a set of edges $\mathcal{E} = \{e_{ij} | \mathcal{R}_i \cap \mathcal{R}_j \geqslant \theta, \forall i, j\}$, where $\mathcal{R}_i$ denotes the set of permission requests answered by user $i(j)$. Edge $e_{ij} \in \mathcal{E}$ denotes the set of permission requests to which both users have answered. Users are connected if the number of commonly answered requests exceeds threshold θ.

The *seed expert* (SE) is one or a set of trusted expert users who might be employed by a DroidNet facilitator to provide *correct* responses to permission requests. It is worth noting that the seed experts follow the *principle of least privilege*, where a minimal set of permissions that are necessary for apps' legitimate purposes are defined to determine the correct responses for permission requests. This way, their responses do not depend on user preferences or context. However, due to the high cost of human labor, the seed expert can only cover limited number of applications. Therefore, identifying expert users from regular users can expand the coverage of apps that can benefit from DroidNet recommendations.

Let $\mathcal{R}_s$ denote the set of requests covered by seed experts. Then the common set of requests answered by both the seed user and user i can be written as $\mathcal{R}_{si} = \mathcal{R}_s \cap \mathcal{R}_i$.

Table 2.3.1.: Notations

Notation	Description
$\mathcal{U}$	$\{U_1, \cdots, U_n\}$: Set of n DroidNet users in the system.
s	The seed user.
$\mathcal{R}_i$	The set of requests responded by user i in the past.
p_i	The true expertise level of user i.
R_i, C_i	The expertise rating and rating confidence of user i.
$(\alpha_{ij}, \beta_{ij})$	The similarity tuple between user i and j.
(α_i, β_i)	The expertise level distribution parameters for user i.

Table 3.1.2 lists the notations we use in this section.

2.3.2.2 The Users Expertise Rating Problem

Before getting into the details, we show a general view of the users network. Figure 2.3.1(a) presents a comprehensive view of DroidNet's user network. In this network we can see different types of users and communities. Figure 2.3.1(b1),(b2) show the User-Seed and User-User connections.

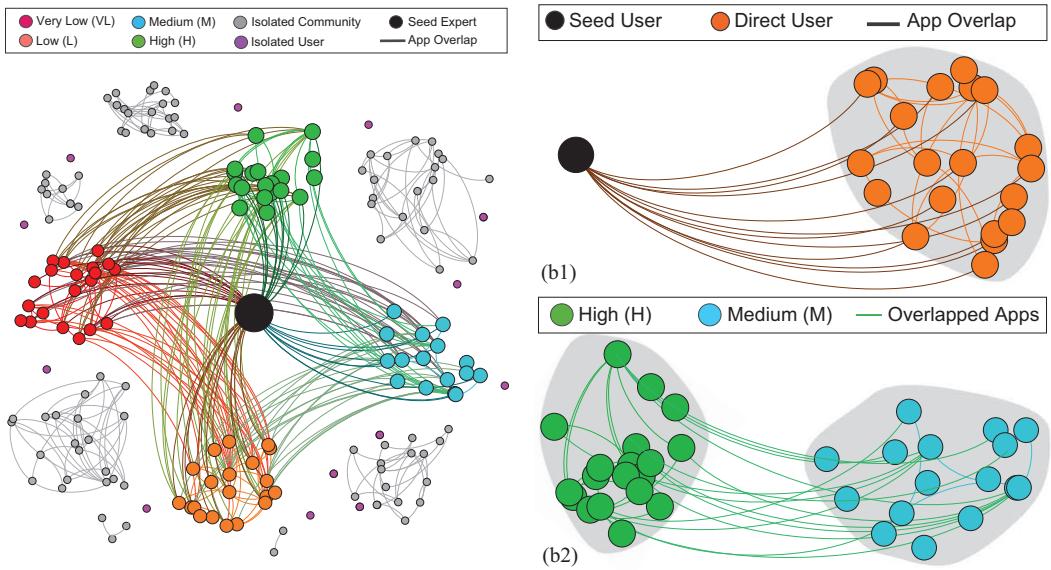


Fig. 2.3.1.: DroidNet user connectivity network: (a) overall view of the network; (b) users' connectivity (User-Seed and User-User)

The *expertise level* of a user i, denoted by $p_i \in [0, 1]$, is the likelihood that the user

makes correct permission granting decisions. Given the set of responses that user i has given to permission requests and their corresponding ground truth, a Bayesian inference model can be used to estimate p_i.

Definition 1. (*Expertise Rating and Rating Confidence*) Assume that the likelihood that a user i makes correct decision (p_i) satisfies a distribution Y_i with pdf $f_i(x)$. Then we define the estimated expertise level of the user to be:

$$R_i = \mathbb{E}[Y_i] = \int_{x=0}^{1} x f_i(x) dx,$$

The confidence level of the estimation is:

$$C_i = 1 - \theta\delta[Y_i] = 1 - \theta\big(\int_{x=0}^{1} (x - R_i)^2 f_i(x) dx\big)\frac{1}{2}$$

where θ is the normalization factor. Therefore, the expertise seeking problem can be described as follows:

Problem 2. (*Expertise Rating Problem*) Given a seed user s, a set of users $\mathcal{U} = \{U_1, ..., U_n\}$, and a DroidNet graph $\mathcal{G} = \{\mathcal{U} \cup s, \mathcal{E}\}$. The expertise rating problem is to find the posterior distributions of all p_i, given their past history of responses to permission requests.

Before presenting the solution, we first define the concept of similarity and then discuss a special case where a user is connected to a seed expert only.

Definition 3. (*Similarity of Two Users*) Let i and j be two users who have responded to a common set of permission requests $\mathcal{R}_i j$, then we define the similarity between i and j as the tuple (α_{ij}, β_{ij}), where α_{ij} and β_{ij} denote the accumulated number of consistent responses and inconsistent responses to those common requests, respectively.

Let $\{x_k \in \{0, 1\} | 1 \leq k \leq n\}$ denote a sequence of n observations in history, where

$x_k = 1$ means that the two users provided consistent responses at the kth overlapped request, and vice versa. The similarity tuple can be computed as follows:

$$\alpha_{ij}^{(n)} = \sum_{k=1}^{n} q^{n-k} x_k + q^n C_0 \tag{2.12}$$

$$= x_n + q x_{n-1} + ... + q^{n-1} x_1 + q^n C_0$$

$$\beta_{ij}^{(n)} = \sum_{k=1}^{n} q^{n-k} (1 - x_k) + q^n C_0 \tag{2.13}$$

$$= (1 - x_n) + q(1 - x_{n-1}) + ... + q^{n-1}(1 - x_1) + q^n C_0$$

Where C_0 is a constant weighting the initial belief; $q \in [0, 1]$ is the remembering factor which is used to discount the influence from past experience and therefore emphasizes the importance of more recent observations.

2.3.2.3 Users Connected to the Seed Expert

We start with the case that a user i who has a common set of responded requests with the seed expert (see Figure 2.3.2(a)). In such case, our approach is to compute the similarity tuple (α_{si}, β_{si}) between the user and the seed, and then the distribution of p_i based on the observations.

We have the following Lemma:

Lemma 4. Let i be a user i that has only one seed expert neighbor in the DroidNet graph. Let (α_{si}, β_{si}) be the similarity tuple of i and the seed expert. Then the rating of the user can be estimated as follows:

$$R_i = \frac{\alpha_{si}}{\alpha_{si} + \beta_{si}} \tag{2.14}$$

$$C_i = 1 - \sqrt{\frac{12 \alpha_{si} \beta_{si}}{(\alpha_{si} + \beta_{si})^2 (\alpha_{si} + \beta_{si} + 1)}} \tag{2.15}$$

Proof. Since the seed expert's advice is assumed correct, α and β are indeed the number of correct and incorrect responses that the user answered in the past. Let a random variable $X \in \{0, 1\}$ denote whether a user answers the permission requests correctly or not. $X = 1$ indicates that user responds to a request correctly, vice versa. Therefore, we have $p = \mathbb{P}(X = 1)$. Given a sequence of observations on X, a beta distribution can be used to model the distribution of p.

In Bayesian inference theory, posterior probabilities of Bernoulli variable given a sequence of observed outcomes of the random event can be represented by a Beta distributions. The Beta-family of probability density functions is a continuous family of functions indexed by the two parameters α and β, where they represent the accumulative observation of occurrence of outcome 1 and outcome 0, respectively. The beta PDF distribution can be written as:

$$f(p|\alpha, \beta) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} p^{\alpha-1}(1-p)^{\beta-1} \tag{2.16}$$

The above can also be written as,

$$p \sim \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} y^{\alpha-1}(1-y)^{\beta-1} \tag{2.17}$$

According to Definition 1, we have Equation (2.14) and (2.15). □

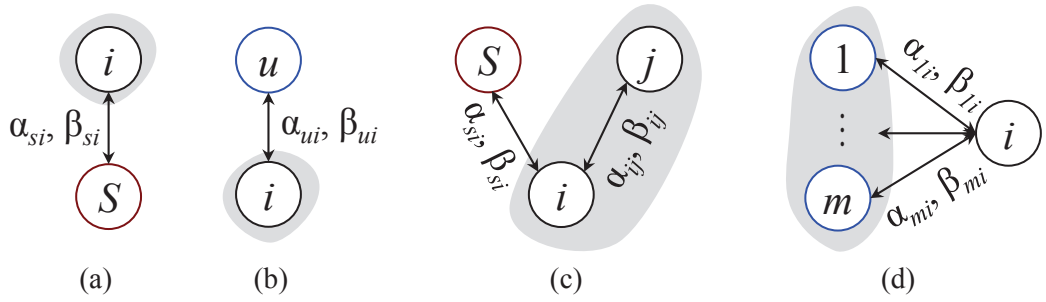


Fig. 2.3.2.: The illustration of four cases of DroidNet graphs. (a) a user is connected directly to a seed user; (b) a user is connected to a non-seed user; (c) a multi-hop rating propagation case; (d) a multi-path rating aggregation case.

2.3.2.4 Users Connected to a Regular User

Due to the limited coverage of the seed user, there may be many users who do not have direct overlap with the seed user (see Figure 2.3.2(b)). To rate users who are connected only to a regular user with known expert rating, we can use the following theorem:

Theorem 5. *Let i be a user connected to a user u with known expertise level $p_u > \frac{1}{2}$ in the DroidNet graph; let $(\alpha_{ui}, \beta_{ui})$ be the similarity tuple of i and u, where $\alpha_{ui} \geq \beta_{ui}$. Then p_i satisfies a Beta distribution: $p_i \sim Beta(\alpha_i, \beta_i)$, where*

$$\alpha_i = \frac{\alpha_{ui} p_u + \beta_{ui}(p_u - 1)}{2p_u - 1} \tag{2.18}$$

$$\beta_i = \frac{\alpha_{ui}(p_u - 1) + \beta_{ui} p_u}{2p_u - 1} \tag{2.19}$$

Proof. Let random variables $X_i \in \{0, 1\}$ and $X_u \in \{0, 1\}$ denote a random event that user i and u respond to permission requests correctly or not. $X_i(X_u)= 1$ means that user $i(u)$ responds to a permission request correctly. Therefore, we have $p_i = \mathbb{P}(X_i = 1)$ and $p_u = \mathbb{P}(X_u = 1)$

Using Bayes theory, the probability that a consistent response being a correct response is formulated as follows:

$$
\begin{aligned}
&\mathbb{P}(X_i = 1 | X_i = X_u) \\
&= \frac{\mathbb{P}(X_i = X_u | X_i = 1)\mathbb{P}(X_i = 1)}{\mathbb{P}(X_i = X_u)} \\
&= \frac{\mathbb{P}(X_u = 1 | X_i = 1)\mathbb{P}(X_i = 1)}{\mathbb{P}(X_i = 1, X_u = 1) + \mathbb{P}(X_i = 0, X_u = 0)} \\
&= \frac{\mathbb{P}(X_u = 1)\mathbb{P}(X_i = 1)}{\mathbb{P}(X_i = 1)\mathbb{P}(X_u = 1) + \mathbb{P}(X_i = 0)\mathbb{P}(X_u = 0)} \\
&= \frac{p_i p_u}{p_i p_u + (1 - p_i)(1 - p_u)}
\end{aligned} \tag{2.20}
$$

Similarly, the probability that an inconsistent response being a correct response is formulated as follows:

$$
\begin{aligned}
&\mathbb{P}(X_i = 1 | X_i \neq X_u) \\
&= \frac{\mathbb{P}(X_i = X_u | X_i \neq 1)\mathbb{P}(X_i = 1)}{\mathbb{P}(X_i \neq X_u)} \\
&= \frac{p_i(1 - p_u)}{p_i(1 - p_u) + (1 - p_i)p_u}
\end{aligned} \tag{2.21}
$$

Note that α_i and β_i denote the cumulative observations that user i responds correctly. Then α_i and β_i can be obtained indirectly from α_{ui} and β_{ui} from the formula below,

$$
\begin{aligned}
\alpha_i =& \alpha_{ui}\mathbb{P}(X_i = 1 | X_i = X_u) + \beta_{ui}\mathbb{P}(X_i = 1 | X_i \neq X_u) \\
\beta_i =& \alpha_{ui}\mathbb{P}(X_i = 0 | X_i = X_u) + \beta_{ui}\mathbb{P}(X_i = 0 | X_i \neq X_u)
\end{aligned}
$$

The above equation set can be transformed into:

$$\alpha_i = \frac{\alpha_{ui} p_i p_u}{p_i p_u + (1-p_i)(1-p_u)} + \frac{\beta_{ui} p_i (1-p_u)}{p_i (1-p_u) + (1-p_i) p_u} \tag{2.22}$$

$$\beta_i = \frac{\alpha_{ui} (1-p_i)(1-p_u)}{p_i p_u + (1-p_i)(1-p_u)} + \frac{\beta_{ui} p_u (1-p_i)}{p_i (1-p_u) + (1-p_i) p_u} \tag{2.23}$$

Note that the estimated expertise level of user i can be written as $R_i = \alpha_i/(\alpha_i + \beta_i)$. However, the actual expertise level p_i of user i is unknown. An iterative method can be used to iteratively update Equations (2.22) and Equation (2.23) starting from $R_i^{(0)} = \frac{1}{2}$ and at each round t replaces p_i with the last round expertise level $R_i^{(t-1)}$. The process stops when $R_i^{(t)}$ converges.

Alternatively we can solve Equation set (2.22) and (2.23) by replacing p_i with $\alpha_i/(\alpha_i + \beta_i)$. Then we get (5) and (6). □

2.3.2.5 Multi-hop User Rating Propagation

Since not all users are connected to the seed user, a rating propagation model is called upon to rate users who are indirectly connected to the seed. As shown in Figure 2.3.2(c), user i has overlap with the seed user, so it can be ranked through our Bayesian ranking algorithm described in Lemma 4. User j only has overlap with user i, so it can be ranked based on its similarity to user i. However, Theorem 5 only works when the expertise of user i is known. Therefore, here we use an iterative method to update the rating of all regular users in DroidNet.

Corollary 5.1. Let i be a regular user directly connected to a set of users $\mathcal{N}_i$. The ratings of the neighbors at round t are (α_i^t, β_i^t), then the rating tuple $(\alpha_i^{(t+1)}, \beta_i^{(t+1)})$ of user i at time $t+1$, can be computed as follows:

$$
\begin{aligned}
\alpha_i^{(0)} =& \beta_i^{(0)} = 1, \forall i, s.t. U_i \in \mathcal{U} \\
\alpha_i^{(t+1)} =& \sum_{k \in \mathcal{N}_i} \Big(\frac{\alpha_{ik} \alpha_i^{(t)} \alpha_k^{(t)}}{\alpha_i^{(t)} \alpha_k^{(t)} + \beta_i^{(t)} \beta_k^{(t)}} + \frac{\beta_{ik} \alpha_i^{(t)} \beta_k^{(t)}}{\alpha_i^{(t)} \beta_k^{(t)} + \alpha_k^{(t)} \beta_i^{(t)}} \Big) \\
\beta_i^{(t+1)} =& \sum_{k \in \mathcal{N}_i} \Big(\frac{\alpha_{ik} \beta_i^{(t)} \beta_k^{(t)}}{\alpha_i^{(t)} \alpha_k^{(t)} + \beta_i^{(t)} \beta_k^{(t)}} + \frac{\beta_{ik} \alpha_k^{(t)} \beta_i^{(t)}}{\alpha_i^{(t)} \beta_k^{(t)} + \alpha_k^{(t)} \beta_i^{(t)}} \Big)
\end{aligned} \tag{2.24}
$$

Proof. From Equation (2.22) and (2.23) we learn that the rating of a node can be computed using the similarity with a source of known rating. We use (α_i^k, β_i^k) denote the transformed observation on user i passed by user k, then we have:

$$
\begin{aligned}
\alpha_i^k =& \frac{\alpha_{ki} p_i p_k}{p_i p_k + (1 - p_i)(1 - p_k)} + \frac{\beta_{ki} p_i (1 - p_k)}{p_i (1 - p_k) + (1 - p_i) p_k} \\
\beta_i^k =& \frac{\alpha_{ki} (1 - p_i)(1 - p_k)}{p_i p_k + (1 - p_i)(1 - p_k)} + \frac{\beta_{ki} p_k (1 - p_i)}{p_i (1 - p_k) + (1 - p_i) p_k}
\end{aligned}
$$

By replacing p_i with $\frac{\alpha_i}{\alpha_i + \beta_i}$ and p_k with $\frac{\alpha_k}{\alpha_k + \beta_k}$, we have:

$$
\begin{aligned}
\alpha_i^k =& \alpha_{ki} \frac{\alpha_k \alpha_i}{\alpha_k \alpha_i + \beta_k \beta_i} + \beta_{ki} \frac{\beta_k \alpha_i}{\beta_k \alpha_i + \alpha_k \beta_i}, \forall k \in \{1, 2, ..., m\} \\
\beta_i^k =& \alpha_{ki} \frac{\beta_k \beta_i}{\alpha_k \alpha_i + \beta_k \beta_i} + \beta_{ki} \frac{\alpha_k \beta_i}{\beta_k \alpha_i + \alpha_k \beta_i}, \forall k \in \{1, 2, ..., m\}
\end{aligned}
$$

In Bayesian inference theory, the observations on one variable can be cumulated through simple summation on all observations, given that they are observed independently. In this case the rating of a user can be represented by the total number of positive and negative observations observed by connected users on different paths. Given that α_i and β_i represent the cumulative positive/negative observations on user i, we have:

$$\alpha_i = \alpha_i^1 + ... + \alpha_i^m = \sum_{k=1}^{m} \alpha_i^k$$
$$\beta_i = \beta_i^1 + ... + \beta_i^m = \sum_{k=1}^{m} \beta_i^k \tag{2.25}$$

□

2.3.2.6 Multi-path User Rating Aggregation

A user may have overlap with multiple other users. As shown in Figure 2.3.2(d), user i is connected to m other users. The overlap with multiple users can be seen as observations from multiple sources and those observations can be aggregated to generate a more accurate ranking of user i.

Corollary 5.2. Let i be a user who has overlap with a set of users $\mathcal{M} = \{U_1, U_2, ..., U_m\}$ with corresponding similarity tuples $\mathcal{S} = \{(\alpha_{1i}, \beta_{1i}), ..., (\alpha_{mi}, \beta_{mi})\}$. Then we have:

$$\alpha_i = \alpha_i^1 + ... + \alpha_i^m = \sum_{k=1}^{m} \alpha_i^k$$
$$\beta_i = \beta_i^1 + ... + \beta_i^m = \sum_{k=1}^{m} \beta_i^k \tag{2.26}$$

where,

$$\alpha_i^k = \alpha_{ki} \frac{\alpha_k \alpha_i}{\alpha_k \alpha_i + \beta_k \beta_i} + \beta_{ki} \frac{\beta_k \alpha_i}{\beta_k \alpha_i + \alpha_k \beta_i}, \forall k \in \{1, 2, ..., m\}$$
$$\beta_i^k = \alpha_{ki} \frac{\beta_k \beta_i}{\alpha_k \alpha_i + \beta_k \beta_i} + \beta_{ki} \frac{\alpha_k \beta_i}{\beta_k \alpha_i + \alpha_k \beta_i}, \forall i \in \{1, 2, ..., m\}$$

Proof. This results are derived from Corollary 5.1 by iteratively computing α_i and β_i on node i in a graph starting from initial setting $\alpha_i^{(0)} = 1$ and $\beta_i^{(0)} = 1$.

□

Algorithm 3 Rate All Regular Users

Compute expertise rating of all regular users in DroidNet
Notations:
$R(\mathcal{U})$: the current rating of all users
$\hat{R}(\mathcal{U})$: the last round rating of all users
s: the seed expert
U_i: the i_{th} user
$\mathcal{G} = (V, E)$: the generated graph of users and overlaps
RU: the set of rated users
QU: the queue of users to be rated
//parameters initialization
set $R(s) = 1$ and $R(U) = 0.5$
while $(Distance(R(\mathcal{U}), \hat{R}(\mathcal{U})) > \epsilon)$ **do**
    $RU \leftarrow s$
    $QU \leftarrow findNeighbors(s)$
    $\hat{R}(\mathcal{U}) \leftarrow R(\mathcal{U})$
    while ($u \leftarrow remove(QU)$ is not null) **do**
        //Users rating using Corollary 5.2
        $R(u) \leftarrow computeRating(u)$
        $RU \leftarrow RU \cup u$
        $\mathcal{N} = findNeighborsNotInRUorQU(u, \mathcal{G})$
        $push(\mathcal{N}, QU)$
    end while
end while

Our approach to determine the order of user rating is to start from the direct neighbours of the seed user, and then we expand the list by looking for the next hop users, and so on. An iterative algorithm is described in Algorithm 3 which rates all regular users in DroidNet. The iteration stops when the difference between two rounds of ratings are sufficiently close.

2.3.2.7 Recommendation Algorithm

After rating users in the network, the next phase is to generate recommendations based on responses from expert users. We propose a weighted voting method to handle the decision making. The voting process is divided into three phases: qualification, voting, and decision. The algorithm is described in Algorithm 4.

In the qualification phase, only responses from qualified users are included into the voting process. Initially the ballot count for reception and rejection decisions are equally

initialized to D_0. For each qualified voter, the weight of the cast ballot is the rating score of the voter. After the voting process finishes, the average ballot score is used for a final decision. If the average ballot score exceeds a decision threshold (τ_d), then corresponding recommendations are made. Otherwise, no recommendation is made.

Algorithm 4 Weighted Voting for Recommendation

Notations :
$R(u), C(u)$:the rating score and rating confidence of user u
$x(u)$:the response to permission request from user u
τ_e, τ_c :the minimum rating score and rating confidence to be considered as an expert user
τ_d :the recommendation threshold
a, b :the cumulative ballots for *yes* or *no* decision
D_0 :the initial ballot count for both decisions
$a = b = D_0$
//Users filtering and ballots casting
for each user u who responded to the request **do**
  if $R(u) > \tau_e$ and $C(u) > \tau_c$ **then**
    if $x(u) = 1$ **then**
      $a+ = R(u)$
    else
      $b+ = R(u)$
    end if
  end if
end for
//decision making based on final ballots count
if $\frac{a}{a+b} > \tau_d$ **then**
  Recommend to *accept* the request with confidence $\frac{a}{a+b} - \tau_d$
else if $\frac{a}{a+b} < 1 - \tau_d$ **then**
  Recommend to *reject* the request with confidence $1 - \frac{a}{a+b} - \tau_d$
else
  No recommendation
end if

2.3.3 Implementation

The goal of DroidNet is to provide a platform for users to grant permissions to apps based on recommendations from expert users. To prove the concept feasibility, we implemented a prototype of DroidNet. More specifically, we modify the permission management component of the Android operating system, and developed an Android application allowing users to monitor and manage resource access permissions at fine-grain level. Figure 2.3.3 shows DroidNet's implementation architecture. DroidNet is installed by applying

a software patch which includes modifications on the Android operating system level and a pre-installed app `DroidNet.apk` on the application level.

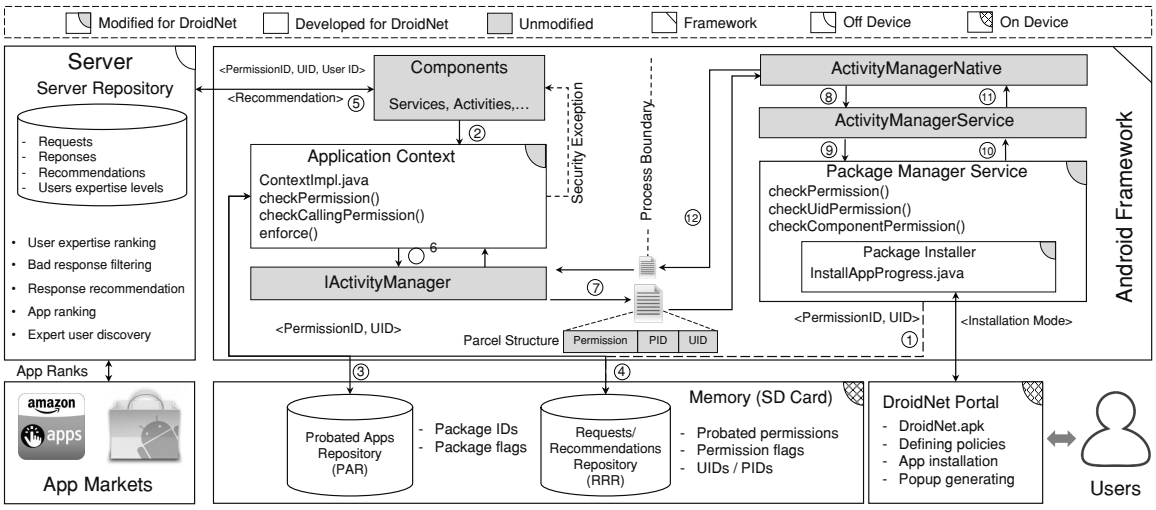


Fig. 2.3.3.: DroidNet implementation architecture overview

2.3.3.1 Permission Control User Interaction

DroidNet users have an option to install apps under a *probation mode*. We use the app "Telegram" (a popular chat application) as an example. The first screenshot (Figure 2.1.3(a)) displays two options when installing the app on the smartphone, e.g. *probation mode* and *trusted mode*. If the user selects the probation mode, then the application will be added to a list of monitored apps on the phone. On the other hand, if the user selects the trusted mode, then the application will be installed with all requested permissions granted.

For each installed app, users can use the pre-installed DroidNet application to view a list of apps which are under the probation mode. If the user clicks on an app in the list, a set of requested resources is displayed (Figure 2.1.3(b)) where checked resources are monitored. By default all sensitive resources are monitored; however users can change this default.

If an app is installed under the probation mode, whenever the app requests to access to a resource under monitoring, the user is informed by a pop-up (Figure 2.1.3(c)). In addition, DroidNet recommends a decision to users with a confidence level. If the user chooses to follow the recommendation, the request of the application will be served; otherwise the request will be blocked.

2.3.3.2 Android Framework Modification

To implement a real-time resource permission control, DroidNet monitors all resource access requests (system calls) at runtime. We modified a few components and methods in Android framework to meet our goal.

App Installation Mode: To allow users to have the option to install under probation or trusted mode, we modified the *Package Manager Service*, which plays the main role in the installation of apps and their requested permissions management. Installation is managed by the *PackageInstaller* activity and when an application installation is completed, a notification is sent to *InstallAppProgress.java*, which is the place we added a post install prompt to ask users if they would like to put application on probation mode.

If a user selects the trusted mode installation then the app would not be managed by DroidNet, and no information will be recorded about the application. If the user selects the probation mode, DroidNet records app's UID and the set of requested permissions by the probated app in the *Probated Apps Repository* (PAR) and *Request/Recommendation Repository* (RRR) repositories. Note that communication is supported by using these repositories that all layers (framework and application) read and write from.

System Calls monitoring and Permission Enforcement: Our implementation is designed to be extensible and generic. While our implementation requires multiple changes in one place, it does not require modifications on every permission request handler, as it was the case on some previous works, such as in MockDroid [18]. The modification is presented

in the form of a framework patch, which can be executed from a user's space, making this technique easier to adopt. In order to design an extensible and central permission enforcing point, we modified methods such as `enforcePermission`, `checkPermission`, and `enforceCallingPermission` of *ContextImpl.java* class of the *context* component of Android. These methods are called whenever an application seeks to use some permissions that are not hardware related. When the methods are called, it is passed an UID and a permission name. We first check to see if the UID is a system call. If yes then we check the repository to see if the UID is present, and if it is, what value the flag associated with the passed in permission has. Algorithm 3 shows the flow of permission enforcement for incoming permission requests.

2.3.3.3 DroidNet recommendation server

Recording the users' responses and providing decision recommendations to users are essential to DroidNet. For this purpose we maintain a remote server to record the responses on an online server and also compute recommendations according to the recorded responses from users. The DroidNet clients request recommendations from the server when needed.

2.3.4 Experiments

For a comprehensive evaluation of the DroidNet system, we use simulation to evaluate the performance of the expert rating and recommendation algorithms.

2.3.4.1 Simulation Setup

As a proof of concept, we created a set of DroidNet users' profiles consisting of four different levels of expertise. The expertise level we refer here is the probability that a user responds to permission requests correctly (a.k.a. consistent with the correct responses). User profiles consist of users with a high (H) expertise (0.9), medium (M) expertise (0.7),

Algorithm 5 Permission Enforcing Flow

```
This algorithm is to decide whether to grant a requested permission to app or deny it
Notations :
PAR :Probated Apps Repository
RRR :Request/Recommendation Repository
flag :denotes that permission is probated
uid :Package identifier
p :Permission name (identifier)
r :user's response for a permission request
//initialize voting parameters
while (there is an incoming permission request) do
    Fetch UID's info from PAR
    if uid ∉ PAR then
        //grant the requested permission
    else
        Fetch apps' probated Ps from RRR
        if p's flag = True then
            //grant the requested permission
        else
            Prompting user through a popup
            if r = True then
                //grant the requested permission and record the user's response
            else
                //deny the request and record the user's response
            end if
        end if
    end if
end while
```

low (L) expertise (0.5), and the remaining are users with a very low (VL) expertise (0.1). Note that VL is considered to be malicious since their responses are misleading most of the time. In order to measure the effectiveness of the DroidNet expertise rating, we use a few study cases of multi-hop and multi-path propagation. In all experiments we set $q = 1\%$.

Our simulation environment is C++ on a Windows machine with 3.6Ghz Intel Core i7 and 16GB RAM. All results are based on an average of 500 repeated runs with different random generator seeds.

2.3.4.2 Expertise Rating and Confidence level

To evaluate the effectiveness of the rating and recommendation model, we start from 4 study cases on a set of nodes with designed configuration. The average number of permission requests per app is set to 5 and the maximum number of requests is 500. Figure 2.3.4

shows the four case studies and their configurations.

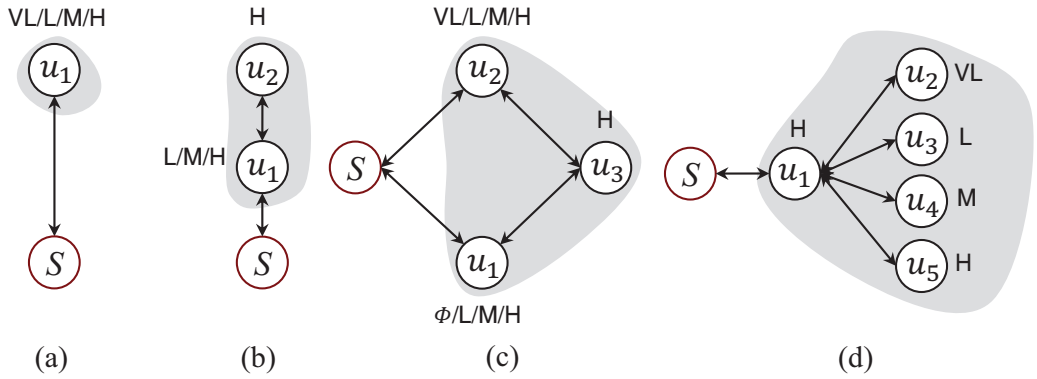


Fig. 2.3.4.: The illustration of four small user profiles designed for our evaluation: (a) the user is connected directly to a seed user; (b) the user is far from seed by distance 1 intermediate user; (c) the multi-path rating propagation case; (d) a multi-path rating case, designed for α and β calculation convergence.

We start from a simple case study in which a user is connected to a seed expert only (Figure 2.3.4(a)), and study the expertise ratings and rating confidence when the user is initialized with H, M, L and VL expertise ratings, respectively. Figure 2.3.5(a) shows the estimated expertise rating for all four types of user's expertise. We can see that when the number of overlapped requests increases, the estimated expertise ratings approach to their true expertise levels. Figure 2.3.5(b) shows the corresponding confidence levels of estimation. The confidence level also increases with the number of overlapped apps. From these results, we can see that DroidNet can have high quality users' expertise rating when the user has sufficient requests overlapping with the seed expert.

In the second case study (Figure 2.3.4(b)), we investigate the influence of intermediate users on the expertise rating propagation. We set user 1 with L, M and H expertise and user 2 with H expertise. Figure 2.3.6(a) shows that the expertise rating of user 2 is influenced by the expertise level of user 1. The higher expertise of the intermediate node, the closer user 2's is rated to its actual expertise level. We call a high expertise node has a *high rating conductivity*. We also conclude that through multi-hop rating propagation, we can find expert users who do not have direct overlap with the seed expert.

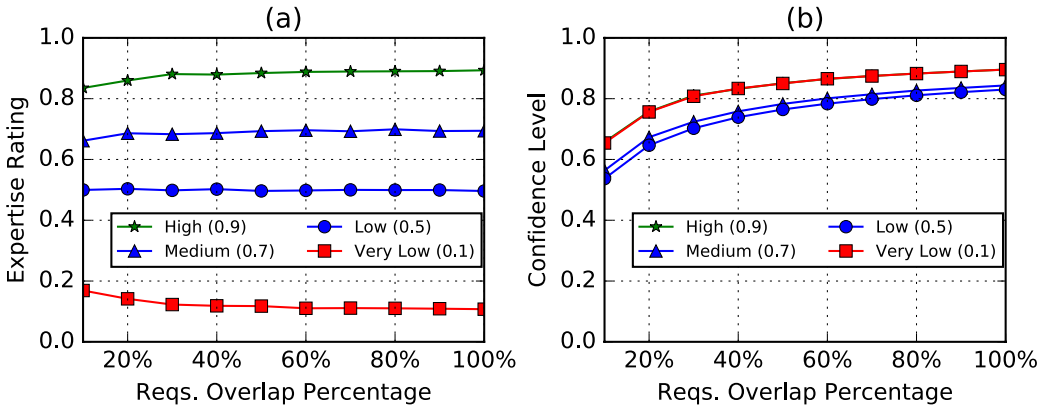


Fig. 2.3.5.: Calculated user expertise and confidence level: (a) expertise level of user with different initial expertise level; (b) computed confidence level of user with different initial expertise level.

In the third case study (Figure 2.3.4(c)), we show the expertise rating of user 3 with two intermediate users between user 3 and the seed expert. We vary the type of user 1 to be void (non-existing), H, M, and L expertise and vary the type of user 2 to be H, M, L, and VL. Figure 2.3.6(b) shows that the conductivity of rating is high if one of the paths has high conductivity node.

Figure 2.3.6(c) shows the expertise rating of five users (with expertise $0.1, 0.5, 0.7$, and 0.9) for the case study shown in Figure 2.3.4(d). As we described in Algorithm 5, we continue updating the expertise rating parameters (α and β) of a user until they converge to a stable value. In this experiment, we show the convergence speed of different types of users through iterating α and β calculation process 10 times start from 1 iteration to 10 iterations. From these results we can see that after 10 iterations all ratings converge to stable values, while the user directly connected to the seed expert achieves stableness after one computation cycle.

In the next experiment we test our technique on a medium size network with 400 users and 250 apps in total (Figure 2.3.1(a)). We set up 100 users for each type (VL, L, M and H). Users choose to install 20 apps out of 250 apps randomly. Figure 2.3.6(d) presents the distribution of the final expertise ratings for all 400 users marked by different colors.

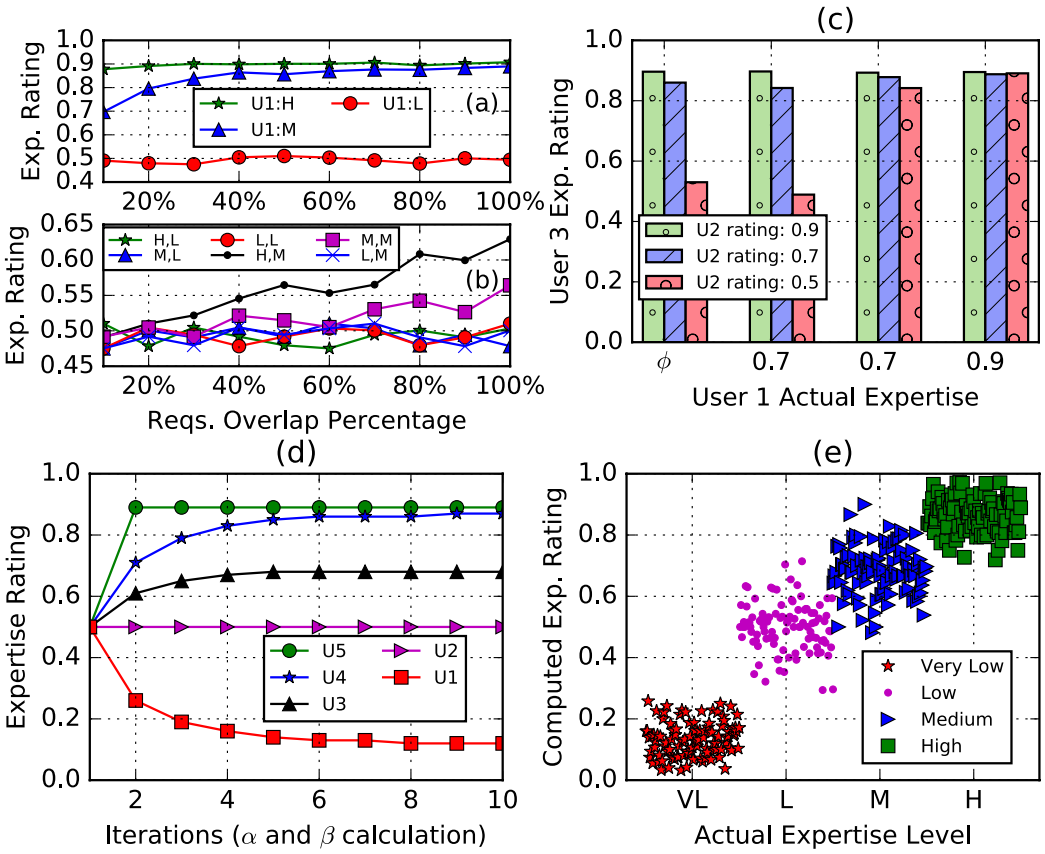


Fig. 2.3.6.: Influence of neighbors on expertise rating: (a)(b) expertise rating of a user with only one user in its locality and different expertise ratings (U1,U2); (c) expertise rating of a user with two users in its locality and different expertise levels; (d) expertise rating of users for different number of α and β calculation iterations; (e) expertise rating distribution after rating users with different actual expertise rating.

We can see that the estimated expertise ratings are clustered around their actual expertise levels; however false positives and false negatives exist.

To evaluate the relationship between the seed expert coverage and false positive on user classification, we repeated the last experiment under 10 different seed expert coverage rates, while using the same configuration. As shown in Figure 2.3.7(a), the number of users assigned to high rating group increases when the seed expert coverage increases.

2.3.4.3 Quality of DroidNet Recommendations

Making accurate permission granting recommendations is the main purpose of DroidNet. We thus evaluate the quality of DroidNet recommendations using two metrics: *cov-*

erage and *accuracy*. Coverage is the percentage of the requests for which DroidNet can offer recommendation to users, while accuracy is the percentage of correct recommendation that DroidNet makes. Note that if a request is covered by a seed expert, DroidNet always recommends the response from the seed expert.

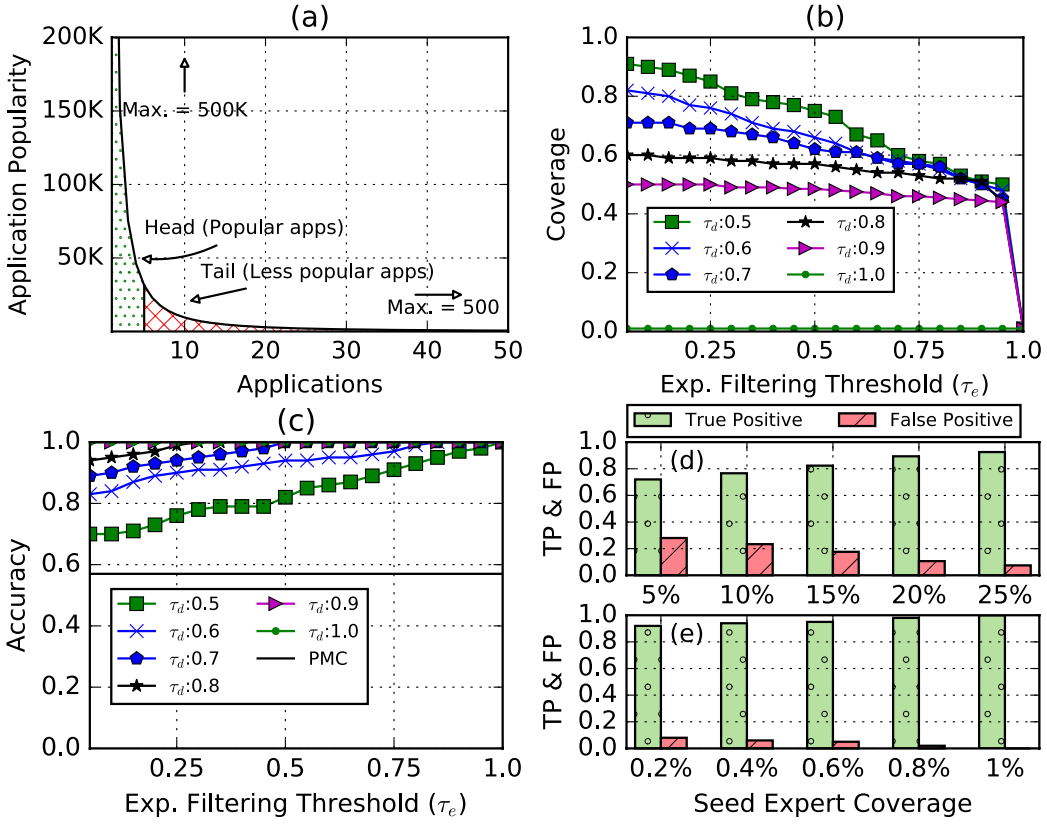


Fig. 2.3.7.: Coverage and accuracy of rating and recommendation: (a) generated dataset based on teh Long-tail distribution; (b) percentage of requests that DroidNet makes recommendation for; (c) percentage correct recommendations that DroidNet makes; (d)(e) accuracy of generated recommendations and seed expert coverage relation.

In order to conduct the simulation close to the real-life scenario as much as possible, we decided to build a network of users and apps proportionally close to the number of Android users ($\simeq$ 2B) and apps ($\simeq$ 2M). Thus, we generated a network with 500K users (125K per each user type) and 500 apps. We set the number of permissions per app and number of apps per user to be 5 and 2 respectively. The number of users per apps follows the *Long-tail* distribution. To generate such dataset of users/apps and assign the apps to

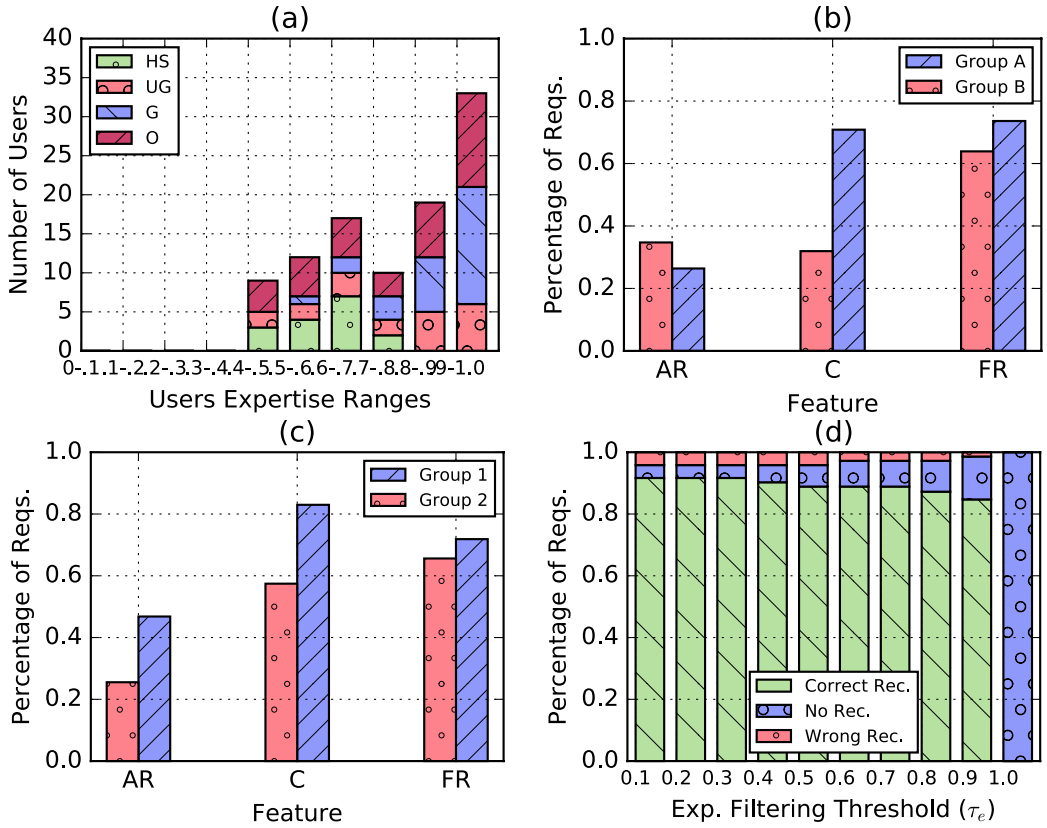


Fig. 2.3.8.: Correctness and recommendation positive response/following rates: (a) expertise ratings of participants; (b) users responses analysis; (c) applications' permission requests analysis; (d) recommendations' accuracy under different normalized τ_e and a fixed τ_d = 0.5.

the users, we used the *power-law* distribution formulation. Power-law distribution formula is $f(cx) = a(cx)^{-k} = c^{-k}f(x) \propto f(x)$, in which x denotes the range of the distribution, a denotes the normalization constant (maximum popularity), k denotes the distribution shape parameter and c is a constant value for scaling the distribution. Fig. 2.3.7(a) shows the distribution of users and apps. For the sake of clarity, only a part of the distribution is illustrated. In this configuration, the number of the most (app 1) and least (app 500) popular apps' users are $\approx$ 500K and $\approx$ 11 respectively.

Figs. 2.3.7(b) and Figs. 2.3.7(c) show the coverage and accuracy of DroidNet under different τ_e and τ_d settings. We can see that with lower values of τ_d (Algorithm 2), the coverage is higher while the accuracy is lower. This shows the trade-off between the coverage

and accuracy. We also notice that the accuracy increases with increasing values of the experts filtering threshold τ_e. However, with very high τ_e, the coverage is low. This is because when all users are included in the decision process, the conflict of responses among users leads to low voting score and therefore DroidNet is less likely to make recommendations. In this experiment, the seed experts coverage is set to 1% of the apps.

To evaluate the relationship between the seed expert coverage and false positive on user classification, we repeated the last experiment under 5 different seed expert coverage rates, while using the same configuration. In this experiment, we also show the difference between two scenarios in terms of covering apps by seed experts. First, when seed experts cover apps randomly selected among top 10% apps and second, when seed experts cover the top 1% apps. It is worth mentioning that the τ_e is set to 0.9 in both scenarios. As shown in Fig. 2.3.7(d)(e), the number of users assigned to high rating group increases when the seed expert coverage increases in both scenarios. The main difference is that using the long-tail distribution helps classifying users more accurately (Fig. 2.3.7(e)).

In the last experiment, we also compare the performance of DroidNet and the PMP system [8]. Fig. 2.3.7(c) shows that the PMP achieves the accuracy of 0.57, whereas DroidNet's accuracy is higher than 0.8.

2.3.4.4 Usability Evaluation

We also evaluated DroidNet through real user experiments. We collected user data to measure the accuracy, reliability, ease-of-use and practicality of the system described as follows.

Participants: We recruited 100 real users to participate in our experiments. To introduce diversity of the participants, we have users from different educational levels (High-School (HS), Undergraduate (UG), Graduate (G) and Others (O)). Table II shows the diversity of the participants in our experiments with respect to educational levels. The par-

ticipants with a higher education degree (undergraduate and graduate) were selected from different majors (engineering and non-engineering). We purposely selected 20 of the grad participants from Computer Science major to see whether participants' majors matters or not. Table III also shows the diversity of our participant in terms of age ranges. We hired participants from different ages (18-50) to conduct a comprehensive usability experiment. It is worth noting that 38% and 62% of our participants are female and male respectively.

Table 2.3.2.: Diversity of Participants (Education level)

Educational Level	High-School	Undergrad	Grad	Others
Number of users	16	20	28	36

Table 2.3.3.: Diversity of Participants (Age)

Age range	$18-19$	$19-25$	$25-35$	$35-50$
Number of users	21	35	37	7

Applications: We selected 12 applications to be evaluated in our experiments. The selected apps include 6 "*trusted*" apps (top ranked) and 6 *aggressive* (not ranked) apps. We define *trusted* apps to be those developed by trusted developers such as *Instagram* (social network), *Weather Channel* (weather category), etc. We define *aggressive* apps to be the apps that request irrelevant resources that they do not need. The apps request various resources including Internet communication, location, camera, storage (photos/media/files), SMS service, and user's contacts. We selected apps from different app categories such as communication, social network, finance, weather, music-audio, card game, and arcade. In each of these categories we downloaded a pair (trusted and aggressive) of apps. There are 72 permission requests in total.

Devices and OS: To prepare for the experiment, we built a customized Android ROM (Android 4.3, Jelly Bean) equipped with DroidNet system. We have used 4 LG Nexus 4 devices running DroidNet system.

Server: DroidNet records all responses from users and stores them on an online server.

The server is implemented on a LAMP stack and uses CentOS, Apache, SQLite3, and the latest version of php.

We asked all participants to respond to the permission requests independently. Among the 72 requests, only 30 requests have DroidNet recommendations available. Those recommendations are created on purpose and may be incorrect to test how likely users will follow those recommendations blindly. We collected the responses from all users for analysis. The ground truth of all permission requests were provided by our seed expert.

2.3.4.5 Data Analysis

Before presenting the results, we show an overall view of the expertise level of users. Fig. 2.3.8(a) shows the expertise rating (unormalized) of all participants based on our collected data and ground truth provided by our seed expert. After applying our expertise rating algorithm on the recorded responses, the expertise rating results of users range from 0.1 (lowest) to 0.92 (highest). Considering the calculated users' expertise ratings, we classify them into four types, users with very low (< 0.1), low (0.1-0.5), medium(0.5-0.7), and high (> 0.7) level of expertise. In this figure, we can see that participants with higher level of education have higher expertise level. As we described in the participants demographic section, 20 of the participants had graduate degrees. 15 of the grad participants have expertise level between $0.9 - 1$. Out of this number 13 of them are Computer Science students. From this result, we can prove that educational background has a direct relation with the expertise level.

To study the correlation between user behavior and their expertise level, we selected two groups of users. Group A are users who are savvy (expertise rating higher than 0.8) and group B are inexperienced (expertise rating below 0.5). Fig. 2.3.8(b) shows the responses from users from the two groups. The *correctness rate (C)* is the percentage of correctly answered requests, and the *following rate (FR)* is the percentage of requests which followed

the DroidNet recommendations. We see a strong correlation between the correctness rate and users groups and a weak correlation between the following rate and user groups. Users in group A achieves much higher correctness rate and behave more conservative in terms of granting permissions. In other words, the *accept rate (AR)* is lower.

We also divided apps into two different groups: trusted group (group 1) and aggressive group (group 2). Fig. 2.3.8(c) shows the responses received for both types of apps. We can see that requests from the apps in group 1 are more likely to be accepted by users, and a higher accuracy rate is also observed for trusted apps. There is no strong correlation between the following rate and app groups.

To evaluate the effectiveness of DroidNet on real data, we run the DroidNet recommendation algorithm on the collected real user data with parameter setting $\tau_d = 0.5$. Note that the recommendations are made only based on the users responses by ignoring the seed experts and normalizing expertise ratings. Fig. 2.3.8(d) shows that the percentage of incorrect recommendations decreases, while the cases of no recommendation increases for increasing values of τ_e. When τ_e is too high, no recommendation will be made.

2.3.4.6 Survey Statistics

Along with the real data collection, we also conducted a survey to measure different factors of DroidNet. In addition to participating in our test, we asked all the 100 users to fill in a questionnaire and answer to some objective multiple-choice questions. Table 2.3.4 shows that the majority percentage (66%) people are concerned with their data privacy on mobile phones, while a large percentage (42%) people believe that the smart phone they use is secure.

We also surveyed the Ease-of-Use and trustworthiness of the DroidNet system. Table 2.3.5 shows that the majority (84%) of the participants believed that DroidNet is easy to use. 72% of the users think that DroidNet's recommendations are reliable.

Table 2.3.4.: Users' opinion on data and device security

Device Security	Secure	Neutral	Not secure	Total
Privacy Concern				
Concerned	28%	23%	15%	66%
Neutral	11%	7%	3%	21%
Not concerned	3%	7%	3%	13%
Total	42%	37%	21%	100%

Table 2.3.5.: DroidNet's Trustworthiness and Ease-of-Use

Level	Low	Medium	High
Metric			
Ease-of-Use	2%	14%	84%
Trustworthiness	8%	20%	72%

2.3.5 Threats and Defenses

Although the purpose of DroidNet is to protect inexperienced smartphone users from being attacked by malicious apps, DroidNet itself may be the target of attacks. In this section, we discuss a few potential threats to DroidNet that we can foresee at this stage. We then show that through integrating strategic defensive design into DroidNet framework, we can detect, deter, or mitigate such threats. We also address the privacy concerns which may rise from DroidNet users and we show that our privacy-aware data collection design can reduce this concern to a minimum.

2.3.5.1 False Recommendations:

One of the main important threats is the injection of false responses to mislead the recommendation system. For example, during the external expert users seeking process, malicious users/attackers behave well in order to be rated as expert users. After being chosen as expert users, they turn around and suggest dishonest recommendations to mislead the recommendation system.

We have investigated this potential threat and developed a multi-agent game theory model to study the gain and loss of malicious user and the DroidNet defense system.

We derived a system configuration to discourage rational attackers to launch such attacks. Through the proposed game model we analyzed the interaction (request/response) between DroidNet users and DroidNet system using a static Bayesian game formulation. In the game, both the DroidNet system and attackers choose the best response strategy to maximize their expected payoff and we studied the Nash Equilibria of the game. We also identified the strategies that DroidNet can use to disincentivize attackers in the system, since they have no gain by attacking the system.

2.3.5.2 Bot Users:

Bot users are fake users which are set up and controlled by attackers to fulfill some specific purpose. For example, the vendor of a malicious app may create many "exper" DroidNet users who will be honest when responding to other applications except to the particular app owned by their "master". Since DroidNet heavily relies on the responses from expert users, many dishonest expert users may misguide DroidNet into providing wrong recommendations if not detected and handled properly. How to detect those bot users and mitigate their impact is an important problem for DroidNet.

In order to address this issue, we have developed a clustering-based method for finding groups of bot users controlled by the same masters, which can be used to detect bot users with high reputation scores. The key part of the proposed method is to map the users into a graph based on their similarity (*features*) and apply a clustering algorithm to group users together. Specifically, we found that malicious users controlled by the same master may:(i) download and respond to the malicious app at as soon as the app is available; (ii) have unusual high overlap on the apps they installed and responded since they are from the same master; (iii) respond to the malicious app differently than benign expert users. We consider the above three features as behaviors of malicious (bot) users. We then apply a customized *weighted* distance function to aggregate them into the similarity between users. After that,

we use a hierarchical clustering method to group users based on their distances. We used *"Agglomerative"* type of hierarchical clustering, which is a *"bottom up"* approach and generate *dendrogram*. As our future work, we plan to apply other clustering methods such as *k-means* or some advanced clustering algorithms. We study this threat and present our solutions in details in Part 4.

2.3.5.3 Application Crashing and DroidNet's Overhead:

In the current implementation of DroidNet, we created an OS patch from all the modifications. Since users have to apply the OS patch to be able to use DroidNet service, it may not be practical for most users. Therefore, as a secondary implementation plan, our framework can be also implemented at the application level in order to make the service accessible to the majority of users by installing an app. This implementation can also avoid app crashing in case of permission denial.

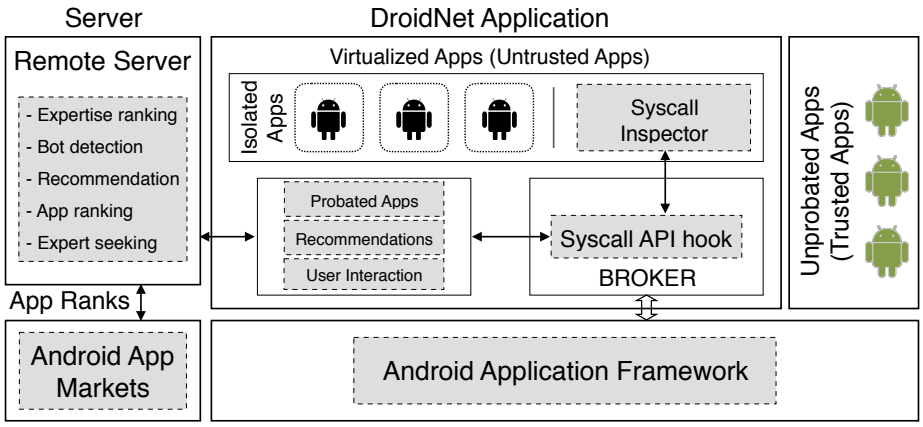


Fig. 2.3.9.: DroidNet implementation architecture overview

As our future plan, we can utilize Android *isolated processes* to be able to run apps at the application level. Figure 2.3.9 shows the application-level implementation architecture. In this implementation isolated processes can be utilized to virtualize apps by loading them into the DroidNet and execute them. This way, system calls and permission requests can

be captured by a component called Broker using Android internal/hidden APIs. The Broker is a system call API hook between the application level and the Android framework. In this implementation users only need to install the *DroidNet app* that can be deployed without firmware modifications or root privileges. One promising example of utilizing Android isolated processes can be Boxify [14]. Boxify is a solution that runs apps at the application level. Boxify has been evaluated and the evaluation results demonstrate its capability to enforce permission control without incurring a significant runtime performance overhead [14]. In addition, using this implementation, apps do not crash upon permission denial, which is an important improvement compared to the current implementation.

Intercepting API calls and syscalls, and enforcing permission control imposes some performance overhead. As one of the main use-cases, Boxify facilitates fine-grained permission control because of it low overhead and runtime robustness. As reported [14], the performance of Boxify is evaluated through a prototype containing both API calls and syscalls. Intercepting API calls to the application framework imposes ans overhead around 1%. For syscalls, a $\approx 100\mu s$ overhead is observed. However, the employed evaluation benchmark depicts the worst case scenario and the overall performance impact on apps is much lower. The measured overall performance overhead by executing several benchmarking apps on top of Boxify, is an acceptable degradation of $1.6\% - 4.8\%$. Therefore, since DroidNet's main activities are calls interception and permission enforcement, we can conclude that Boxify is an effective platform to implement DroidNet on top of it.

2.3.5.4 Privacy Concerns:

DroidNet is a crowdsourcing-based solution and seeking expert users in the network is an important task. DroidNet collects permission responses from all participating users to discover experts. To protect the privacy of users, we design a privacy-aware data collection mechanism that uses *hashing and salting* method (Figure 2.3.10) to protect the true identity

of the users. The salt is randomly generated upon installation. Note that this mechanism provides *double-blind* protection, which means that an attacker who successfully attacked the database will not be able to reverse the function to find out the real phone ID or even verify whether a given phone ID is in the database. Therefore, the identity of the users is well-protected, and our mechanism does not compromise the usability of the collected data.

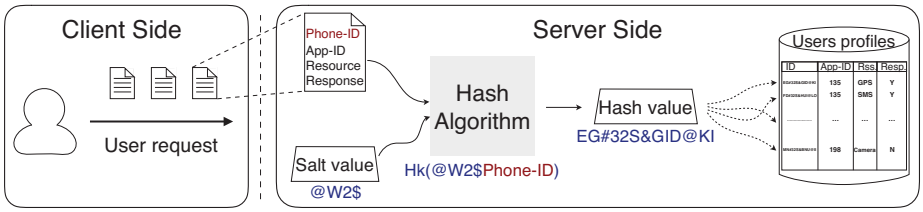


Fig. 2.3.10.: Use one-way ID hashing to protect users' privacy

2.3.5.5 Newly Published Applications (Cold Start)

DroidNet collects resource access requests and user responses and generate recommendations using the presented algorithms in this section. One issue of relying on users for generating recommendations can be lack of sufficient amount data (user responses). This phenomena is called *Cold Start* This happens when apps are newly published in app markets and there is not sufficient amount of data about the apps. To address such issue, we proposed an Android risk assessment model using hidden Markov model (HMM) to assess the risk of newly published apps. The proposed model falls into behavioral-based approaches. The input to the model is apps' activities and the output is the risk of misusing resources. The model is also equipped with an online learning mechanism that helps to tune the model's sensitivity based on users' preferences. We study this issue and present our approach in details in Chapter 3.

2.3.5.6 Platform Dependency:

A well-designed model is a model that is independent of the specific technological platform used to implement it. Such model is called platform-independent. In other words, a platform independent model performs effectively and it is not restricted by the type of environment provided. We believe that DroidNet is a platform-independent model that can be applied and implemented on various mobile operating systems and hardware platforms. In the case of Android OS updates (at any layer), since the DroidNet model itself is platform-independent, so it can be adapted to the updates. In other words, as long as Android is using a permission-based mechanism as one of its security mechanism, DroidNet can be applied to it.

2.3.6 Conclusion

In this section we presented DroidNet, an Android permission control and recommendation system which serves the goal of helping users perform low-risk resource accessing control on untrusted apps to protect their privacy and potentially improve efficiency of resource usages. We propose a framework that allows users to install apps in either trusted mode or probation mode. In the probation mode, users are prompted with resource accessing requests and make decisions on whether to grant the permissions or not. To assist inexperienced users to make low-risk decisions, DroidNet provides recommendations on permission granting based on the responses from expert users in the system. In order to do so, DroidNet uses crowdsourcing techniques to search for expert users using a transitive Bayesian inference model. Our evaluation results demonstrate that DroidNet can effectively locate expert users in the system through a small set of seed experts. The recommending algorithm achieves high accuracy and good coverage when parameters are carefully selected. We implemented our system on Android phones and demonstrate that

the system is feasible and effective through real users experiments.

CHAPTER 3

ANDROID APPLICATION BEHAVIOURAL RISK ANALYSIS

As we previously mentioned, our crowdsourcing-based model is able to generate recommendations for apps which we have a sufficient amount of information about apps and also responses to permission requests from users on those apps. In cases there are not enough information about apps we rely on a set of risk analysis and malware detection models which mainly work in static and dynamic analysis manners. In this section, we elaborate the details of two Android app risk assessment and malware detection frameworks to assess the risk of the apps based on their behaviours on the user's phone. The frameworks utilize hidden Markov models (HMM) and machine learning model to assess the risk of apps and classify them as malicious or benign. The proposed models assess the risk through analyzing app's activity log. In addition, we also design an online learning mechanism to enhance the flexibility of the model based on user's preferences.

3.1 XDroid: An Android permission control using Hidden Markov Models

The major contributions of the work reported in this section include:

- An instrumentation tool that facilitates app behaviour logging in order to generate high quality dataset for analysis.
- A comprehensive time-aware Android app behaviour analysis, which is based on the apps' intents and actions, as well as extra features that further improves detection accuracy.
- A trained hidden Markov model which can decide whether an app is malicious or not

based on its behaviour.

- A dynamic model which can be updated in real time to integrate users' preferences. This is potentially the first time that an HMM online learning model is used on malicious app control in smartphone security.

3.1.1 Problem Definition

Android users are allowed to install third-party applications from various open markets. This raises security and privacy concerns since the third-party applications may be malicious. Unfortunately, the increasing sophistication and diversity of the malicious Android applications render the conventional defenses techniques ineffective, which results in a large number of malicious applications to remain undetected. In this section we present XDroid, an Android application and resource risk assessment framework based on the Hidden Markov Model (HMM). In our approach, we first map the applications' behaviours into an observation set, and we attach timestamps to some observations in the set. We show that our novel use of temporal behaviour tracking can significantly improve the malware detection accuracy, and that the HMM can generate security alerts when suspicious behaviours are detected. Furthermore, we introduce an online learning model to integrate the input from users and provide adaptive risk assessment. We evaluate our model through a set of experiments on the DREBIN benchmark malware dataset. Our evaluation results demonstrate that the proposed model can accurately assess the risk levels of malicious applications and provide adaptive risk assessment based on user input.

3.1.2 Background

In this section we briefly review some background knowledge of *Hidden Markov Models (HMM)* and the problems that we need to address using *HMMs* to be able to compute

the risk-level of Android apps.

3.1.2.1 Hidden Markov Model

A Hidden Markov Model (HMM) is a statistical Markov model which is widely used in science, engineering and many other areas (speech recognition, optical character recognition, machine translation, bioinformatics, computer vision, finance and economics, and in social science) [42]. Markov Models provide powerful abstraction for time series data, but fail to address one of the most common scenarios [42]. For example, how can we reason about a series of states if we cannot observe the states directly, but can only observe some probabilistic function of those states? A HHM can be used to explore this scenario. In this section we model the Android apps' behaviour and compute the risk-level of their activities and device resource usage.

The Hidden Markov Model is a variant of a *finite state machine* which can be represented by a set of hidden states $\mathcal{Q} = \{q_1, q_2, \cdots, q_{|Q|}\}$, a set of observations $\mathcal{O} = \{o_1, o_2, \cdots, o_{|O|}\}$, a set of transition probabilities $\mathcal{A} = \{\alpha_{ij} = P(q_j^{t+1}|q_i^t)\}$, and a series of output (emission) probabilities $\mathcal{B} = \{\beta_{ik} = P(o_k|q_i)\}$. Among the above notations, $P(a|b)$ is the conditional probability of event a given event b; $t = 1, \cdots, T$ is the time; q_i^t is the event that the state is q_i at time t. In other words, α_{ij} is the probability that the next state is q_j given that the current state is q_i; β_{ik} is the probability that the output is o_k given that the current state is q_i. The initial state probabilities are denoted by $\Pi = \{\pi_i = P(q_i^1)|\forall 1 \leq i \leq |Q|\}$, which is the inital probability of all states at time 1. In HMM the current state is not observable. However, each state produces an output with a certain probability (denoted by $\mathcal{B}$). An *HMM* can also be represented using a compact triple ($\lambda = (\mathcal{A}, \mathcal{B}, \Pi)$) [59]. Table 3.1.2 shows the notations and preliminaries.

There are two fundamental problems that we need to address when using HMMs: How to determine unknown parameters and how to find optimal state sequence. We briefly

Table 3.1.1.: Notations

Notation	Description
$\mathcal{Q}$	$\{q_i\}, i = 1 \cdots, N$: Set of n hidden states .
$\mathcal{A}$	$\mathcal{A} = \{\alpha_{ij} = P(q_j^t)\}$ Transition probabilities
$\mathcal{O}$	$\mathcal{O} = \{o_k\}, k = 1, \cdots, M$: Observations (symbols)
$\mathcal{B}$	$\mathcal{B} = \{\beta_{ik} = P(o_k \mid q_i)\}$: Emission probabilities
Π	$\Pi = \{\pi_i = P(q_i^1) \mid \forall 1 \leq i \leq \lvert Q \rvert\}$ Initial state probabilities.
O^t	$O^t \in \mathcal{O}$: Observation at time t.
Q^t	$Q^t \in \mathcal{Q}$: State at time t.

describe the two problems and their solutions in the next sections.

3.1.2.2 Finding the unknown parameters

One of the challenge of making the HMMs applicable to malicious Android apps detection is to define parameters $\mathcal{Q}, \mathcal{A}, \mathcal{O}, \mathcal{B}, \Pi$, and find the most likely set of state transition and output probabilities. To solve this problem, we use Baum-Welch algorithm (a.k.a Forward-Backward algorithm) [60]. We call this step *HMMs Training*. The algorithm has two main steps, Forward and Backward procedures.

Initialization set $\lambda = (\mathcal{A}, \mathcal{B}, \Pi)$ with random initial conditions. The algorithm updates the parameters of λ iteratively until convergence, following the next procedures.

The forward procedure We define $\alpha_i^t = P(O^1, \cdots, O^t, Q^t = q_i \mid \lambda)$, which is the probability of seeing the partial sequence $O^1, \cdots, O^t$ and the ending state Q^t at time t is q_i. We can compute α_i^t recursively as follows:

$$\alpha_i^1 = \pi_i b_i(O^1) \tag{3.1}$$

$$\alpha_j^{t+1} = b_j(O^{t+1}) \sum_{i=1}^{N} \alpha_i^t \alpha_{ij} \tag{3.2}$$

where $b_i(O^j)$ is the probability that observation O^j at time j given state i.

The backward procedure We define β_i^t to be the probability of the ending partial sequence $O^{t+1}, \cdots, O^T$ given that we started at state q_i, at time T. We can efficiently compute β_i^t as:

$$\beta_i^T = 1 \tag{3.3}$$

$$\beta_i^t = \sum_{j=1}^{N} \beta_j^{t+1} \alpha_{ij} b_j(O^{t+1}) \tag{3.4}$$

using α and β, we can compute the following variables:

$$\gamma_i^t \equiv P(Q^t = q_i | O, \lambda) = \frac{\alpha_i^t \beta_i^t}{\sum_{j=1}^{N} \alpha_j^t \beta_j^t} \tag{3.5}$$

$$\xi_{ij}^t \equiv P(Q^t = q_i, Q^{t+1} = q_j | O, \lambda) = \frac{\alpha_i^t \alpha_{ij} \beta_j^{t+1} b_j(O^{t+1})}{\sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i^t \alpha_{ij} \beta_j^{t+1} b_j(O^{t+1})} \tag{3.6}$$

where γ_i^t is the probability that the state at time t is q_i, and ξ_{ij}^t is the probability the state at t is Q_i and the state at $t+1$ is q_j. With γ and ξ , we can define update rules as follows:

$$\pi_i = \gamma_i^1 \tag{3.7}$$

$$\alpha_{ij} = \frac{\sum_{t=1}^{T-1} \xi_{ij}^t}{\sum_{t=1}^{T-1} \gamma_i^t} \tag{3.8}$$

$$\beta_{ik} = \frac{\sum_{t=1}^{T} \delta(O^t, o_k) \gamma_i^t}{\sum_{t=1}^{T} \gamma_i^t} \tag{3.9}$$

3.1.2.3 Finding the optimal state sequence

One of the most common queries of a *HMMs* is to ask what was the most likely series of states given an observed series of outputs. In the other words, we should choose the best state sequence that maximizes the likelihood of the state sequence for the given observation sequence. The solution to this problem is using Viterbi algorithm [78]. The Viterbi algorithm is similar to the forward procedure except that it only tracks the maximum

probability instead of the total probability.

Let δ_i^t be the maximal probability of state sequences of the length t that end in state i and produce the t first observations for the given model. That is,

$$\delta_i^t = max\{P(Q^1, \cdots, Q^{t-1}; O^1, \cdots, O^t | Q^t = q_i)\} \tag{3.10}$$

The two differences between Viterbi algorithm and the Forward algorithm are: (1) it uses maximization in place of summation at the recursion and termination steps, (2) it keeps track of the arguments that maximize δ_i^t for each t and i, storing them in the N by T matrix ψ. This matrix is used to retrieve the optimal state sequence at the backtracking step.

We initial the model as:

$$\delta_i^1 = \pi_i b_i(O^1) \tag{3.11}$$

$$\psi_i^1 = 0, i = 1, \cdots, N \tag{3.12}$$

The recursion steps are:

$$\delta_j^t = max_i[\delta_i^{t-1} a_{ij}] b_j(O^t) \tag{3.13}$$

$$\psi_j^t = argmax_i[\delta_i^{t-1} a_{ij}] \tag{3.14}$$

Finally the most probable sequence's probability $p(T)$ and the most probable last state $q(T)$ given O are:

$$p(T) = max_i[\delta_i^T] \tag{3.15}$$

$$q(T) = argmax_i[\delta_i^T] \tag{3.16}$$

We can have the path (state sequence) through backtracking:

$$q(t) = \psi_{q(t+1)}^{t+1}, t = T-1, T-2, \cdots, 1 \tag{3.17}$$

3.1.3 System Design

The ultimate goal of XDroid is to monitor the behaviour of apps and generate alerts to users when suspicious app behaviours are detected. Figure 3.1.1 shows the architecture design of XDroid. The system contains components on the server side and the mobile device side. Each XDroid device contains an *Interaction Portal* and an *Activity Logger*. The interaction portal provides an interface for users to interact with the device. The activity logger is used to monitor the activities of the apps. The server side components include *Risk Assessment*, *User Profiling*, and *Alert Customization*. In the rest of this section, we describe the key features of the server.

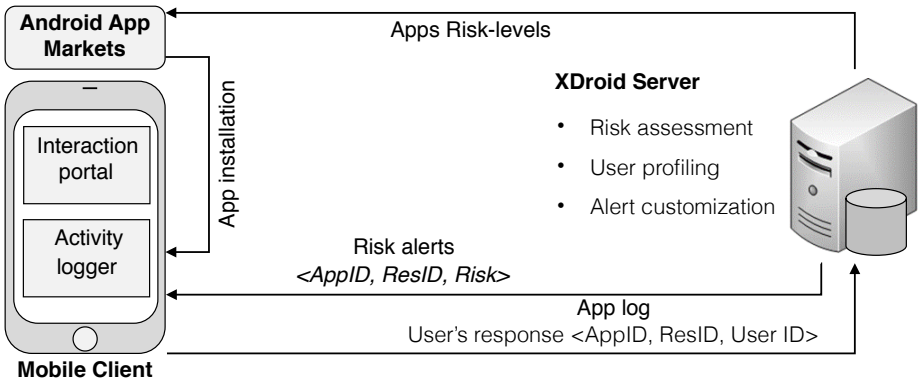


Fig. 3.1.1.: XDroid system overview

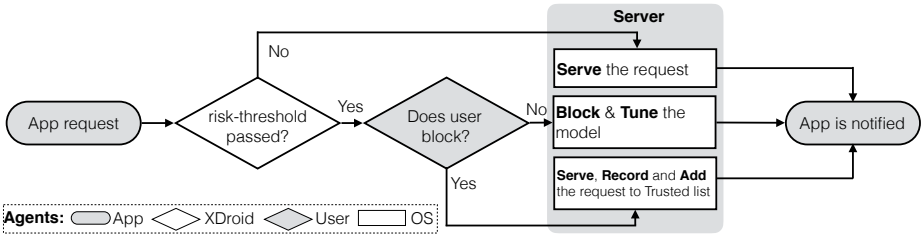


Fig. 3.1.2.: Resource request flow in XDroid

3.1.3.1 Interaction Portal

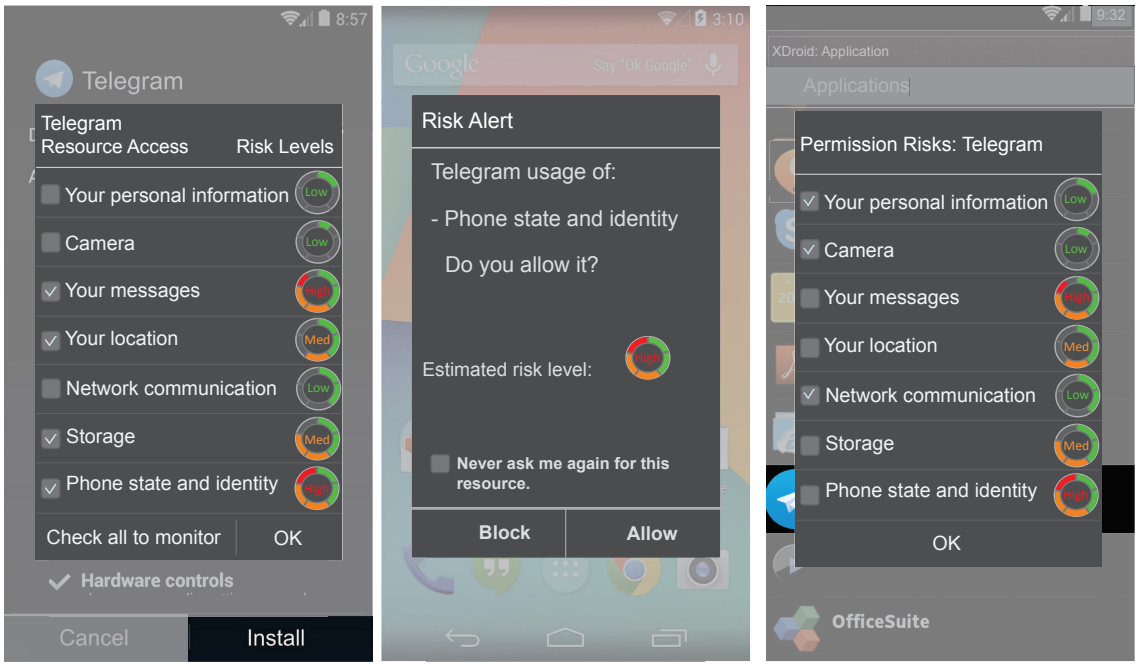
The interaction portal is to facilitate the interaction between users and devices. Instead of sending requests to the Android system's legacy permission handler (e.g. Package Manager Service), the XDroid handles the permission requests through the process illustrated in Figure 3.1.2. For example, when a user installs a popular messaging application and choose to monitor its behaviour using XDroid, the requested resources are displayed along with their estimated risk levels (Figure 3.1.3(a)). The numbers in the screenshots are generated as example and not actual risks. The user can check resources he/she want to monitor. If a resource is monitored and its suspicious activities are detected, the user is informed through a dialog box (Figure 3.1.3(b)). The user can decide whether to block the resource access or allow it based on the estimated risk suggested by XDroid.

For each installed app, the user can use the pre-installed XDroid application to view a list of apps which are under monitoring. If the user clicks on an app in the list, a set of requested resources is displayed (Figure 3.1.3(c)) where checked resources are monitored. By default all sensitive resources are monitored, and can be changed by the user.

3.1.3.2 Risk assessment

The purpose of the risk assessment is to provide a quantitative estimation on how likely a resource access from an app causes damage to users. For example, a SMS access from a puzzle game app may be malicious and XDroid can pop up a reminder for users to block it.

To assess the risk level of resource accesses, the activities of the apps are monitored by activity loggers and the data is sent to the server for analysis. Our risk assessment mechanism uses a Hidden Markov Model (HMM) to analyze the behaviour sequences (Section 3) and provides users with a risk level of involved resource accesses.



(a) (b) (c)

Fig. 3.1.3.: User Interfaces: (a) illustrates the risk computed risk levels for app's requested resources; (b) shows a popup notifying user the risk level of resource at runtime; (c) managing the permission policies after installation.

3.1.3.3 User Profiling

We are aware that users may have different tolerance level on various resources. For example, user A, is very concerned with leaking his/her location to a third party, while user B does not care about it. To provide customized risk estimation, we build user profiles. We assign each new user with an initial tolerance model and update the user profile after receiving the users' permission control decisions. For example, if a user agrees to the GPS access every time it is requested, the tolerance level of the user with respect to GPS access is high.

3.1.3.4 Alert Customization

The purpose of XDroid user profiling is to provide customized risk assessment to users. Upon installing a new app, XDroid provides risk level estimation by integrating the apps' behaviours and past responses from all users who responded to the same app requests. This way, we help users make a decision on wether to monitor the permission requests or not. If users choose to monitor an app's permission, after capturing sufficient behaviour log, XDroid computes the risk level of that permission by considering the new logs and alerts the user. Users' responses (block or allow) to the permission request alerts are integrated to provide customized alerts.

3.1.4 Model

In this work, we use hidden Markov model (HMM) for Android malicious apps risk assessment. We transform the app resource risk computation problem into a HMM problem with two states: *malicious* and *normal*. We map the app's behaviour into the HMM observations. To train the HMM model, we capture the behaviours from both malicious and normal apps and use them to generate an initial trained HMM for risk estimation. In this section we first present our HMM model and then explain how we use it for customized permission risk level estimation.

3.1.4.1 Hidden Markov Model

An HMM is a statistical Markov model widely used in science, engineering and many other areas (speech recognition, optical character recognition, machine translation, bioinformatics, computer vision, finance and economics, and in social science) [42, 35]. Markov models provide powerful abstraction for data expressed as time series, but are unable to support reasoning about a series of states given some observations related to those states.

A HHM can be used to address such shortcoming. In this section we model the Android behaviour sequence into a HMM, and compute the risk levels of given resource usage.

The Hidden Markov Model is a variant of a *finite state machine* which can be represented by a set of hidden states $\mathcal{Q} = \{q_1, q_2, \cdots, q_{|Q|}\}$, a set of observations $\mathcal{O} = \{o_1, o_2, \cdots, o_{|O|}\}$, a set of transition probabilities $\mathcal{A} = \{\alpha_{ij} = P(q_j^{t+1}|q_i^t)\}$, and a series of output (emission) probabilities $\mathcal{B} = \{b_{ik} = P(o_k|q_i)\}$. Among those notations, $P(a|b)$ denotes the conditional probability of event a given event b; $t = 1, \cdots, T$ is the time; q_i^t denotes the event that the state is q_i at time t. In other words, α_{ij} is the probability that the next state is q_j given that the current state is q_i; b_{ik} is the probability that the output is o_k given that the current state is q_i. The initial state probabilities are denoted by $\Pi = \{\pi_i = P(q_i^1)|\forall 1 \leq i \leq |Q|\}$, which is the initial probability of all states at time 1 (initial time). In the HMM the current state is not observable. However, each state produces an output with a certain probability (denoted by $\mathcal{B}$). An HMM can also be represented using a compact triple ($\lambda = (\mathcal{A}, \mathcal{B}, \Pi)$) [59]. Table 3.1.2 summarizes the notations and preliminaries.

Table 3.1.2.: Notations

Notation	Description
$\mathcal{Q}$	$\{q_i\}, i = 1 \cdots, N$: Set of n hidden states .
$\mathcal{A}$	$\mathcal{A} = \{\alpha_{ij} = P(q_j^t)\}$ Transition probabilities
$\mathcal{O}$	$\mathcal{O} = \{o_k\}, k = 1, \cdots, M$: Observations (symbols)
$\mathcal{B}$	$\mathcal{B} = \{b_{ik} = P(o_k\|q_i)\}$: Emission probabilities
Π	$\Pi = \{\pi_i = P(q_i^1)\|\forall 1 \leq i \leq \|Q\|\}$ Initial state probabilities.
O^t	$O^t \in \mathcal{O}$: Observation at time t.
Q^t	$Q^t \in \mathcal{Q}$: State at time t.

Figure 3.1.4 shows the HMM for Android app behaviour modeling. The HMM consists of three states: *Start*, *Normal* (0) and *Malicious* (1). The set of observations are defined using apps' behaviours during runtime (see Section 3.1.4.7). The HMM parameters are: initial state probabilities $\Pi = [\pi_1, \pi_2]$, state transition probabilities $\mathcal{A} = [\alpha_{00}, \alpha_{01}, \alpha_{10}, \alpha_{11}]$, malicious state emission probabilities $\mathcal{B}_M = [b_{11}, b_{12}, \cdots, b_{1N}]$ and normal state emission

probabilities $\mathcal{B}_N = [b_{01}, b_{02}, \cdots, b_{0N}]$.

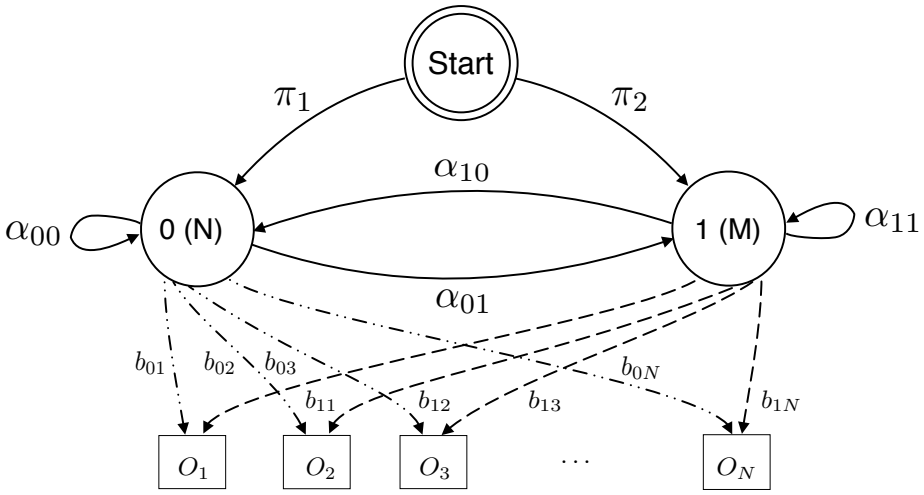


Fig. 3.1.4.: An overview of the proposed HMM model

How to determine the unknown parameters and how to seek optimal state sequence of HMM given a sequence of observations are two major challenges. In the next sections we describe these two problems and their solutions.

3.1.4.2 Compute Unknown Parameters

One of the challenges in utilizing the HMM to model the behaviour of Android apps is to define parameters $\mathcal{Q}, \mathcal{A}, \mathcal{O}, \mathcal{B}, \Pi$, and find the most likely set of state transition and output probabilities. We use two states $\mathcal{Q} = \{0, 1\}$ to denote that the app is behaving normal (0) or malicious (1). The observation set $\mathcal{O}$ is the set of time-aware system calls expressed using a keywords set (see Section 3.1.4.7). To define $\mathcal{A}$ and $\mathcal{B}$, we use the Baum-Welch algorithm (a.k.a Forward-Backward algorithm) [60]. We call it *HMMs Training*. The algorithm has two main steps, Forward and Backward procedures.

3.1.4.3 Initialization set

Let $\lambda = (\mathcal{A}, \mathcal{B}, \Pi)$ with random initial conditions. The algorithm updates the parameters of λ iteratively until convergence, following the next procedures.

3.1.4.4 The forward procedure

We define $\alpha_i^t = P(O^1, \cdots, O^t, Q^t = q_i|\lambda)$, which is the probability of seeing the partial sequence $O^1, \cdots, O^t$ and the ending state Q^t at time t is q_i. We can compute α_i^t recursively as follows:

$$\alpha_i^1 = \pi_i b_i(O^1) \tag{3.18}$$

$$\alpha_j^{t+1} = b_j(O^{t+1}) \sum_{i=1}^{N} \alpha_i^t \alpha_{ij} \tag{3.19}$$

where $b_i(O^j)$ is the probability that observation O^j at time j given state i.

3.1.4.5 The backward procedure

We define β_i^t to be the probability of the ending partial sequence $O^{t+1}, \cdots, O^T$ given that we started at state q_i, at time T. We can efficiently compute β_i^t as:

$$\beta_i^T = 1 \tag{3.20}$$

$$\beta_i^t = \sum_{j=1}^{N} \beta_j^{t+1} \alpha_{ij} b_j(O^{t+1}) \tag{3.21}$$

using α and β, we can compute the following variables:

$$\gamma_i^t \equiv P(Q^t = q_i|O, \lambda) = \frac{\alpha_i^t \beta_i^t}{\sum_{j=1}^{N} \alpha_j^t \beta_j^t} \tag{3.22}$$

$$\xi_{ij}^t \equiv P(Q^t = q_i, Q^{t+1} = q_j|O, \lambda) = \frac{\alpha_i^t \alpha_{ij} \beta_j^{t+1} b_j(O^{t+1})}{\sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i^t \alpha_{ij} \beta_j^{t+1} b_j(O^{t+1})} \tag{3.23}$$

where γ_i^t is the probability that the state at time t is q_i, and ξ_{ij}^t is the probability the

state at t is Q_i and the state at $t+1$ is q_j. Let function $\delta(x, y)$ be 1 if $x = y$ and 0 otherwise. With γ and ξ, we can define update rules as follows:

$$\pi_i = \gamma_i^1 \tag{3.24}$$

$$\alpha_{ij} = \frac{\sum_{t=1}^{T-1} \xi_{ij}^t}{\sum_{t=1}^{T-1} \gamma_i^t} \tag{3.25}$$

$$b_{ik} = \frac{\sum_{t=1}^{T} \delta(O^t, o_k)\gamma_i^t}{\sum_{t=1}^{T} \gamma_i^t} \tag{3.26}$$

3.1.4.6 Finding the Optimal State Sequence

One of the most common queries of a *HMM* is to find the most likely series of states given an observed series of observations. In our case, we can find the state sequence (e.g., NMNNNNMN...) that most likely happens given the observation sequence. This problem can be solved using Viterbi algorithm [78]. The Viterbi algorithm is similar to the forward procedure except that it only tracks the maximum probability instead of the total probability.

Let δ_i^t be the maximal probability of state sequences of the length t that end in state i and produce the t first observations for the given model. That is,

$$\delta_i^t = max\{P(Q^1, \cdots, Q^{t-1}; O^1, \cdots, O^t | Q^t = q_i)\} \tag{3.27}$$

The two difference between Viterbi algorithm and the Forward algorithm are: (1) the Viterbi algorithm uses maximization in place of summation at the recursion and termination steps, (2) the Viterbi algorithm keeps track of the arguments that maximize δ_i^t for each t and i, storing them in the N by T matrix ψ. This matrix is used to retrieve the optimal state sequence at the backtracking step.

We initial the model as:

$$\delta_i^1 = \pi_i b_i(O^1) \tag{3.28}$$

$$\psi_i^1 = 0, i = 1, \cdots, N \tag{3.29}$$

The recursion steps are:

$$\delta_j^t = max_i[\delta_i^{t-1} a_{ij}] b_j(O^t) \tag{3.30}$$

$$\psi_j^t = argmax_i[\delta_i^{t-1} a_{ij}] \tag{3.31}$$

Finally the most probable sequence's probability $p(T)$ and the most probable last state $q(T)$ given O are:

$$p(T) = max_i[\delta_i^T] \tag{3.32}$$

$$q(T) = argmax_i[\delta_i^T] \tag{3.33}$$

We can have the path (state sequence) through backtracking:

$$q(t) = \psi_{q(t+1)}^{t+1}, t = T-1, T-2, \cdots, 1 \tag{3.34}$$

3.1.4.7 Observations

Our vision to specify app behaviour is to view the running app as a black-box and focus on its interaction with the Android OS. In this case, a typical interface to monitor is the set of system and API calls that the app invokes during running time. Every action that involves communication with the apps' resources (e.g., accessing the file system, sending SMS/MMS over the network, accessing the location services, calling Ads API libraries, and accessing the network) requires the app to launch OS services or API calls.

In order to instrument apps to capture the behaviour logs, we developed our own tool *DroidCat*. We did not choose to use existing instrumentation tools, such as Robotium and

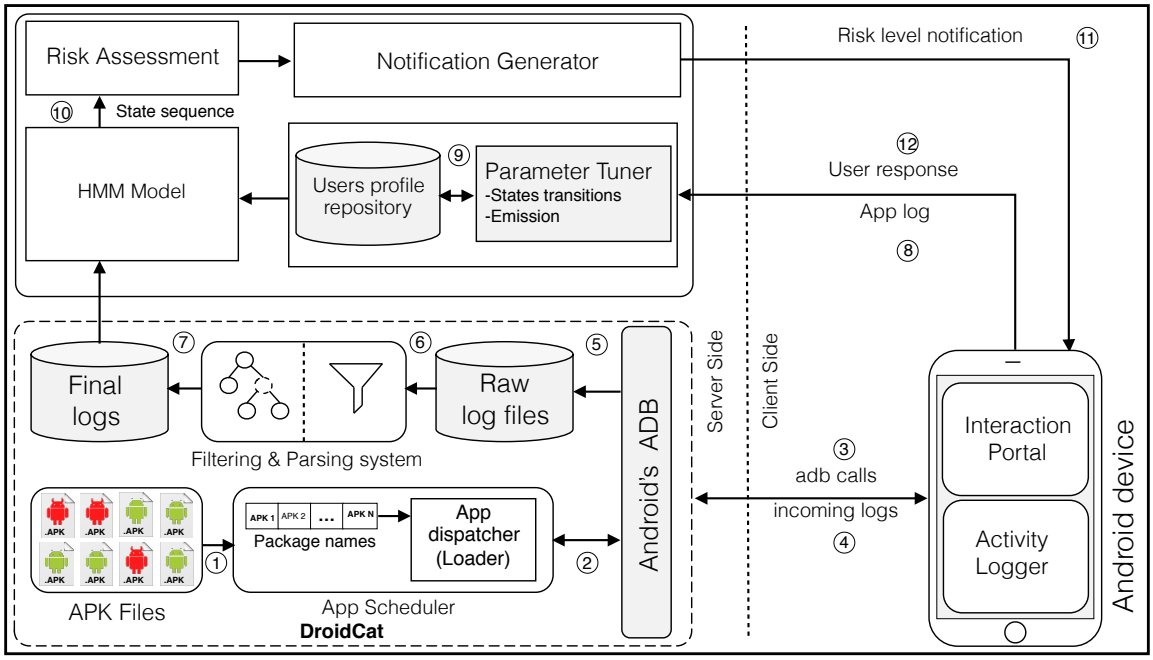


Fig. 3.1.5.: The architecture of the XDroid system

uiautomator, because of their drawbacks. For example, Robotium cannot handle Flash or Web components nor simulate the clicking on soft keyboard, and it is not suitable for multi-process applications tests. Therefore, we decided to develop DroidCat. Figure 3.1.5 shows the XDroid system extended with the integration of DroidCat. The main merit of DroidCat is that it instruments apps through real human-interaction, so we can get behaviour logs which highly represent real world Android apps' behaviours. In the rest of this section we briefly explain the major instrumentation process of DroidCat, which includes the following steps:

3.1.4.8 Extracting packages' names.

DroidCat extracts Android packages' names using the Android `aapt` tool. This tool is part of the Android SDK (and build system) and allows us to view, create, and update Zip-compatible archives (`zip`, `jar`, `apk`). This component creates a queue and add the

packages' names into it. As you can see from Figure 3.1.5, this component is embedded into the *App Scheduler* component of XDroid.

3.1.4.9 App dispatcher.

DroidCat automatically loads Android `apk` files into an Android device, installs and runs them using the Android ADB `logcat` tool and add them to the *App Scheduler*. This component has a timer that lets apps to run for a *specific time period*. It executes the same process for all apps.

3.1.4.10 Recording apps' logs.

DroidCat records every app's log into a text file (raw log files) separately without applying any of `logcat` priority filtering tags (e.g., V, D, I etc.). We use apps' PID as an identifier to capture their logs.

3.1.4.11 Filtering.

In this step DroidCat filters the logs and removes irrelevant logs such as the logs related to loading, installing, running and killing apps' process.

3.1.4.12 Parsing.

After filtering the log files, DroidCat eliminates unnecessary information and extract important keywords. Each keyword refers to a sensitive resource access request, an API call, or Android action constants. In the context of HMM, we call them *observations*. In our model, we focus not only on the Intents generated by the apps but also on API libraries that generate unwanted Ads or cause permission escalation. In total we defined 150 keywords under various categories. Table 3.1.3 lists some selected sample keywords from 6 categories.

Table 3.1.3.: Keyword samples

Resource	Corresponding Keywords
Ads libraries	'AdMob', 'Ads', 'Wooboo', 'AdsMOGO', etc.
Network	'browser', 'http', 'wifi', 'signal', 'cell', etc.
Messaging	'MMS', 'SMS', 'MmsService', 'getSmsCount', etc.
Location services	'GPS', 'ACCESS-COARSE-LOCATION', 'location', etc.
File system	'mount', 'unmount', 'Storage', 'Modify', etc.
Calling/Contacts	'CallLOG', 'CARRIER', 'INCALL', 'TELECOM', etc.

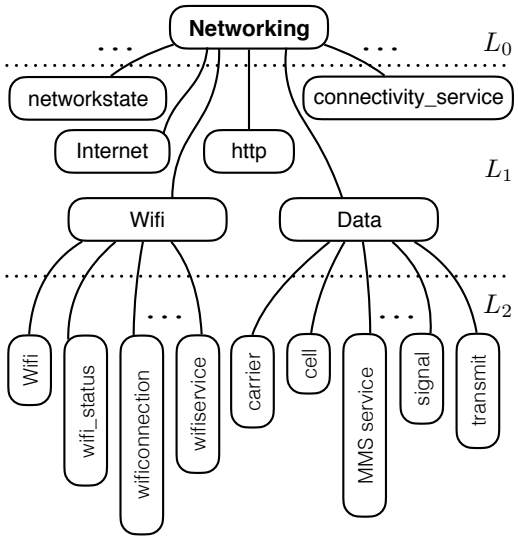


Fig. 3.1.6.: An illustration of the Networking observation tree

```
D/AdsMOGO SDK(30331): Hashed device ID is:
e8a1112658fdeb899033567722ef3e8c
D/AdsMOGO SDK(30331): AdsMOGO SDK Version:…
I/AdsMOGO SDK(30331): Finished creating adM
I/AdsMOGO SDK(30331): Stored config info
not present or expired, fetching fresh data
D/AdsMOGO SDK(30331): Hashed device ID is:
e8a1112658fdeb899033567722ef3e8c
D/AdsMOGO SDK(30331): AdsMOGO SDK Version:…
I/AdsMOGO SDK(30331): Finished creating adM
D/AdsMOGO SDK(30331): ", "timestamp": -1}
D/AdsMOGO SDK(30331): location is ON
I/AdsMOGO SDK(30331): Rotating Ad
D/AdsMOGO SDK(30331): Dart is <...
D/AdsMOGO SDK(30331): Showing Config:
D/AdsMOGO SDK(30331):  Mogo_ID: b2609...
D/AdsMOGO SDK(30331):  CountryCode: us
D/AdsMOGO SDK(30331): HTTP/1.1 200 OK
D/AdsMOGO SDK(30331): Prefs{b2609a99a...
 {"config": "{"extra":{"location_on"... \\
 "red":0,"green":0,"blue":0,"alpha":1},...
```

Fig. 3.1.7.: A sample snapshot of a malicious app's output log

Figure 3.1.6 shows the observation hierarchical tree of the *Networking* service (L_0) on a device. We can see that the networking service action keywords are split into two main categories (L_1) - networking through data service or wifi service. Both categories have their own action keywords (L_2).

Figure 3.1.7 shows a sample snapshot of a malicious app's log. This app uses a malicious advertising API library that can result in permission escalation. This advertising component has access the host app's permissions and can misuse the granted permissions. We can see that the app has access to the GPS service and IMEI information (highlighted with color).

When analyzing the parsed logs, we noticed that for some resources such as "WiFi",

malicious and normal apps have different patterns with respect to the timing of requests during app running. For example, the malicious apps tend to request the WiFi network during the first quarter of their running time period. Because of this, we include the timing of requests and library calls as a feature. Among of the 150 keywords, we added the timing feature to 55 of of them. We finally defined 205 *time-dependent* and *time-independent* observations in total. We summarize the features and factors that we have considered in our as follows:

- *App system calls:* system and regular API calls that apps invoke during runtime are considered as main part of our decision process. Every action that involves communication with the apps' resources (e.g., accessing the file system, sending SMS/MMS over the network, accessing the location services, and accessing the network) requires the app to launch OS services or API calls.

- *Ads API libraries:* in contrast with the described existing approaches and in order to make our detection process more comprehensive, we also captured Ads API libraries logs.

- *Activity timestamp:* after looking at the time that apps made their system calls, we noticed that malicious apps made system calls in the first quarter of their runtime. We decided to consider time as an additional (secondary) feature in our detection process.

3.1.4.13 Model Training and Testing

After defining observations and parsing logs into sequences of *time-dependent* and *time-independent* observations, we trained the HMM model using the Baum-Welch algorithm. In order to do it, we needed to define the initial state transition probabilities $\mathcal{A}$ and

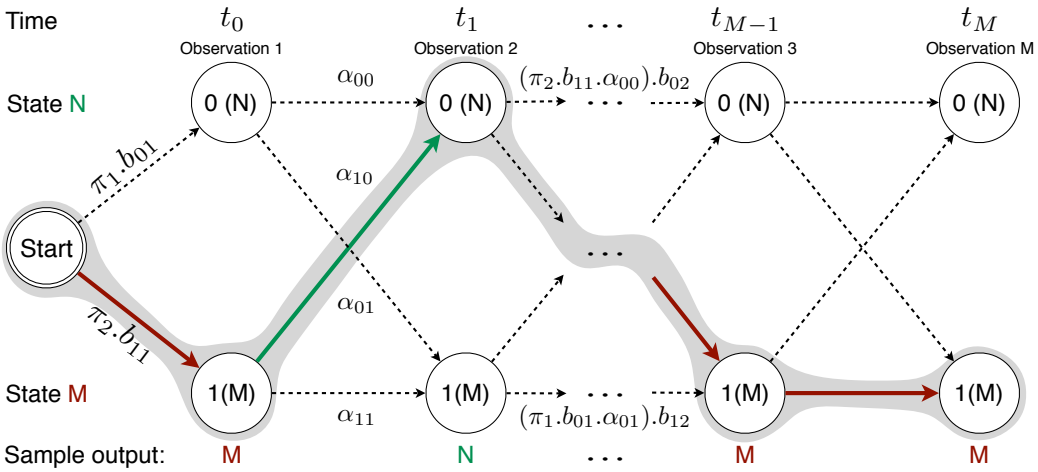


Fig. 3.1.8.: An illustration of the Viterbi algorithm

emissions probabilities $\mathcal{B}$. We computed the initial emissions probabilities of each observation based on its frequency of occurrence in all malicious and normal apps' logs. We also initialized all the states' transitions probabilities with fixed value 0.5. After initialization, we applied the Baum-Welch algorithm to find the parameters $\mathcal{A}$ and $\mathcal{B}$ given sequences of observations.

To validate the accuracy of the HMM-based risk computation, we used app's behaviour log as input to the *Viterbi algorithm* to get the most possible state sequence of the app. Figure 3.1.8 visualizes the test process based on the Viterbi algorithm. The Viterbi algorithm uses backtracking method (Eq. 3.34) to compute the optimal state sequence path given observations, such as (MNN$\cdots$NMM). For example, the highlighted path in Figure 3.1.8 shows a path from start state ($observation_1$) to the final observed data ($observation_M$) state. Algorithm 6 presents the risk calculation process of the XDroid including backtracking process and in more details. We explain the details of $ComputeRick(S)$ function in the next subsection.

3.1.5 Permission Risk Assessment

Another purpose of the HMM is to compute the risk of the resource access from apps. In this section we explain the process of the permission risk assessment and explain how we can perform user profiling and customized alerts.

3.1.5.1 Resource risk assessment

To assess the risk of each resource requested from an app, we track the observation logs and users' decisions on permission requests (allow or block). An online HMM learning technique is used to update the HMM parameter sets for each app. Note that the initial HMM parameters for each app are learned from the base training dataset and the HMM will be refined further through integrating inputs from users.

Algorithm 6 Risk computation

This algorithm is to find the most possible sequence of output given a sequence of observations
Input :
An observation sequence o[t]{Android apps' logs}
A transition matrix A
An observation likelihood vector b[i,o[t]]
Output :
$v[i,t]$: the probability matrix
S : the sequence of output
Notations :
$v[]$: the path probability matrix
c : the index to the v matrix
bp: the back pointer
the current probability $v[i,t+1]$
N : the number of states
//initialize voting parameters
$v[0,0] \leftarrow 1$
for $i = 0$ to $|T|$ **do**
    for $i = 0$ to $|N|$ **do**
        for each transition i from s **do**
            if $k > v[i,t+1]$ **then**
                {when the current probability is higher}
                $v[i,t+1] \leftarrow k$
                $bp[i,t+1] \leftarrow s$
            end if
        end for
    end for
end for
//finalizing the optimal path
repeat
    $add(S, MAX(v[:,c])$
    {highest probability state in the column c}
    $c++$
until $c = |T|$ {repeats until we have the full path}
ComputeRisk(S){computing the app's risk based on our formulation}

Figure 3.1.9 illustrates users' relationship with a graph. Each node represents a user. Each edge means there is an overlap of installed apps between two users. We define a set of users who share at least one app a *group*. A user can belong to different groups. A group can have overlap with other groups. As we can see in Figure 3.1.9, $G_1, G_2, \cdots, G_M$ are the groups and $G_1 \cap G_2$ is the overlap of G_1 and G_2. An *isolated user* is the user who does not have overlap with any other user. In Figure 3.1.9, u_i is an isolated user.

Let $P = \{p_1, p_2, \cdots, p_N\}$ be the set of all apps' permissions, $A = \{a_1, a_2, \cdots, a_M\}$ be the set of all apps, and $U = \{u_1, u_2, \cdots, u_K\}$ be the set of all users. Then the set of user i's apps and their permissions can be written as $U_i = \{(a_1, p_1), (a_1, p_2), \cdots, (a_M, p_N)\}$.

Let S be the optimal state sequence path given observations. We estimate the risk level of a single permission i as follows:

$$R_{pi} = \frac{\sum_{i=1}^{|S|} \delta(S_i, M)}{|S|} \tag{3.35}$$

where $\sum_{i=1}^{|S|} \delta(S_i, M)$ is the number of malicious states in the sequence, and $|S|$ is the length of the sequence.

In order to estimate the average risk level of a permission in the whole network, we use the following formula:

$$R_{all}(p_i) = \frac{\sum_{j=1}^{|U|} R_{p_{ij}}}{|U|} \tag{3.36}$$

where $R_{p_{ij}}$ is the reported risk level of user j for permission i and $|U|$ is the number of users that answered the risk alert for permission p_i in the network.

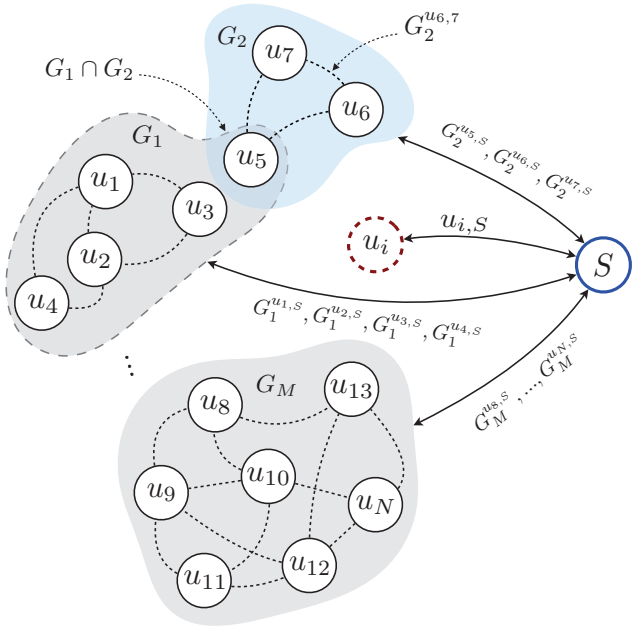


Fig. 3.1.9.: Users network overview

3.1.5.2 User profiling

We record all users' profiles into the repository on the server. Each user profile includes a customized HMM parameter tuner, users' past responses to apps' permission requests, apps' behaviour logs reported by the user's device, and the risk levels for all apps and corresponding resources.

3.1.5.3 Customized Alert generator

This component is responsible for generating customized alerts and notifying the users. Alerts are displayed through popups (Figure 3.1.3(b)), which contain the risk level of the permission and users can choose to either block or accept the permission request. Note that the risk level estimation is customized based on each user's tolerance level to that specific resource. For example, a GPS call may be acceptable for one user but prohibited for another.

Algorithm 7 Updating process

This algorithm is to train and update the model
Input :
Incoming observed data $D_0, \cdots, D_\infty$ {incoming blocks of observations from apps }
λ_0 : the initial model
Output :
Trained and updated model $\lambda_1, \cdots, \lambda_\infty$ {model updates, after every processed block of observation}
Notations :
$tflag$: model training status
$dflag$: new incoming data block
if $tflag = false$ **then**
    {if the model has not been trained}
    $\lambda_1 \leftarrow train(\lambda_0, D_0)$ {training the model}
else
    loop
        {an infinite loop, listening for incoming logs from apps}
        if $dflag = true$ **then**
            {if there is a new block of data for apps}
            $\lambda_n \leftarrow update(\lambda_{n-1}, D_{n-1})$ {updating the model using the existing new data blocks}
        end if
    end loop
end if

3.1.6 Parameter updating through online learning

To integrate users' preferences to the customized risk alerts, we use an *online learning* technique[54], which optimize a log-likelihood function through HMM parameters updating. These techniques are derived from another method which uses batch input for computation [54, 41]. However the key difference of online learning is that the HMM parameter updates are based on the currently presented subsequence (observations from apps) of observations without iterations [42, 41]. Algorithm 6 presents the learning process for updating the HMM model using incremental data blocks (apps observations), where the updates are triggered periodically after a certain amount of apps' logs are collected. As shown in Figure 3.1.11, $D_1, D_2, \cdots, D_n$ are the blocks of training data available to the model at time $t_1, t_2, \cdots, t_n$. The update process starts with an initial risk value m_0 which constitutes the prior knowledge of the domain through training process with existing datasets. During the incremental training, m_0 is updated to m_1 with input D_1, and so on. In our model, a data block contains a sequence of observations related to a resource with which the risk level is computed.

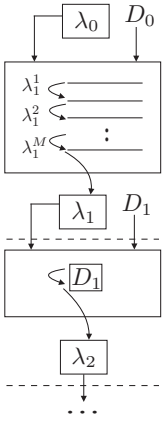


Fig. 3.1.10.: Training and updating process

Figure 3.1.10 shows the update process after receiving an incoming block of data. The model updates the parameters in a sequential manner after every single observation

and it results in a set of partially updated models $(\lambda_1^1, \lambda_1^2, \cdots, \lambda_1^M)$ after M steps. When the update process completes we have a final model λ_1. Therefore, the update process tunes the corresponding emissions probabilities based on the users responses to risk alerts. Algorithm 7 shows the updating process in more details.

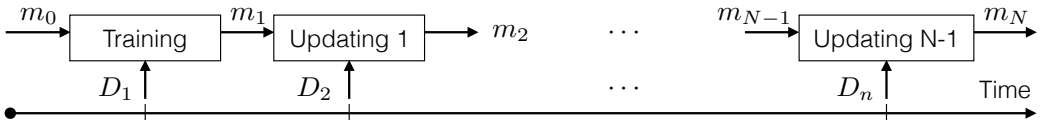


Fig. 3.1.11.: Training and update process using online learning technique

The online learning technique proposed by Mizuno et al. [54] is based on the *Baum-Welch algorithm*. It applies a decayed accumulation of the state densities and a direct update of the model's parameters after each subsequence of observations. Starting with an initial model λ_0, the conditional state densities are iteratively computed after processing each subsequence (*r*) of observations of length *T* by:

$$\sum_{t=1}^{T-1} \xi_{r+1}^t(i,j) = (1-\eta_r) \sum_{t=1}^{T-1} \xi_r^t(i,j) + \eta_r \sum_{t=1}^{T-1} \xi_{r+1}^t(i,j) \tag{3.37}$$

$$\sum_{t=1}^{T} \gamma_{r+1}^t(j)\delta(O^t, o_k) = (1-\eta_r) \sum_{t=1}^{T} \gamma_r^t(j)\delta(O^t, o_k) + \eta_r \sum_{t=1}^{T} \gamma_{r+1}^t(j)\delta(O^t, o_k) \tag{3.38}$$

where the learning rate (forgetting factor) η_k is expressed in polynomial form $\eta_r = (\frac{1}{t})$. The η_k parameter can be initialized manually or automatically (*time dependent*). The norm *time-dependent* means that the impact on the learning rate will discount through time. Using this parameter we can control the impact of the former models on the updating process. Note that the HMM parameters are directly updated using equations (3.25, 3.26).

3.1.7 Activity logger implementation

As shown in the architecture design (Figure 3.1.5), each XDroid client device contains an activity logger to capture the behaviour of apps and report them to the XDroid server.

To implement a real-time resource permission control, XDroid monitors all resource access requests and apps' activities at runtime. We modified a few components and methods in Android OS source code to meet our goal. Figure 3.1.12 shows the implementation architecture.

3.1.7.1 App installation pop-up

To give users the option to have their apps monitored, we modified the *Package Manager Service*, which plays the key role in permissions management. Installation is managed by the *PackageInstaller* activity and when an app installation process is completed, a notification is sent to *InstallAppPregress.java*, which is the place we prompt users and ask them if they like to monitor some permissions. If a user chooses to monitor some permissions of an app, XDriod records the app's UID and the selected permissions into the repositories for *Monitored Apps* (MAR) and *Request/Risk* (RRR) respectively.

3.1.7.2 System call and permission enforcement

Towards the goal of extensible and generic implementation, we managed to have multiple changes in one place, which avoids modifications on each permission request handler. The modification is presented in the form of an OS patch, which can be executed from a user's space, making this technique easier to adopt. More specifically, we modified methods `enforcePermission`, `checkPermission`, and `enforceCallingPermission` of the *ContextImpl.java* class in the *context* component. These methods are called whenever an app uses a permission which is not hardware related, through passing a UID and a permission name. On the server side, XDroid is used to filter and parse the logs (Figure 3.1.5), which will be the input source for the HMM model.

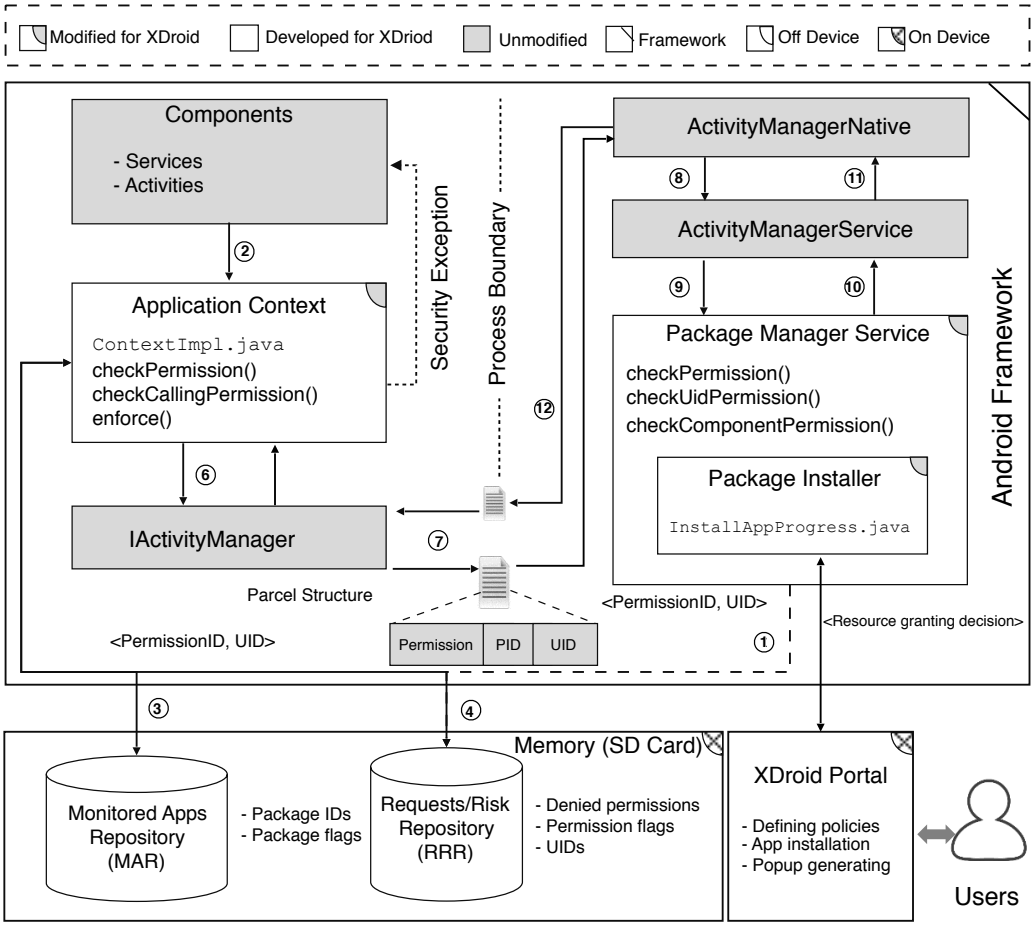


Fig. 3.1.12.: Permission request logging

3.1.7.3 XDroid server

Recording the users' responses and providing decision recommendations to users are essential to XDroid. For this purpose we maintain a remote server to record the responses on an online server and also compute recommendations according to the recorded responses from users. The XDroid clients request recommendations from the server when needed.

3.1.8 Experimental Results

In this section we present our experiments evaluating the proposed model. We first explain our experimental setup and then the results on the performance of the HMM based

risk assessment model. We validate the accuracy of the model in terms of recognizing malicious apps from normal apps. We also evaluate the computed risk levels and the impact from parameters.

3.1.8.1 Experiment Setup

In this section we describe the experimental environment, hardware, software and the dataset that we used to train and test the model.

Hardware: To log apps' behaviours we used 5 LG Nexus 4 devices equipped with Android OS version 4.3. We chose Android 4.3 version because all apps in our datasets are compatible with this version. We also configured the devices and turned on all sensitive resources (services) such as WiFi, Bluetooth and GPS. We ran our DroidCat on a 64-bit Windows machine with 3.30GHz Intel Xenon, 16G RAM.

Software: Our experimental environment is MATLAB 2015 running on the same Windows machine. We implemented the *Baum-Welch algorithm* with default tolerance 1e-4 and the *Viterbi algorithm* to train and test the model. The tolerance level of Baum-Welch algorithm controls how many steps the algorithm executes before the function returns an answer.

Datasets: To have an effective HMM risk assessment model, we need to train the model with sufficient behaviour logs from both malicious and normal apps. We obtained our malicious apps set from the Computer Security Group of University of Gottingen, which was collected under the Drebin project [13]. The dataset contains 5560 malicious apps from 179 different malware families. We selected 700 apps from this dataset so that we have multiple apps from every malware family. In addition to the 700 malicious apps, we collected 700 benign apps from various categories of Android apps. We randomly selected

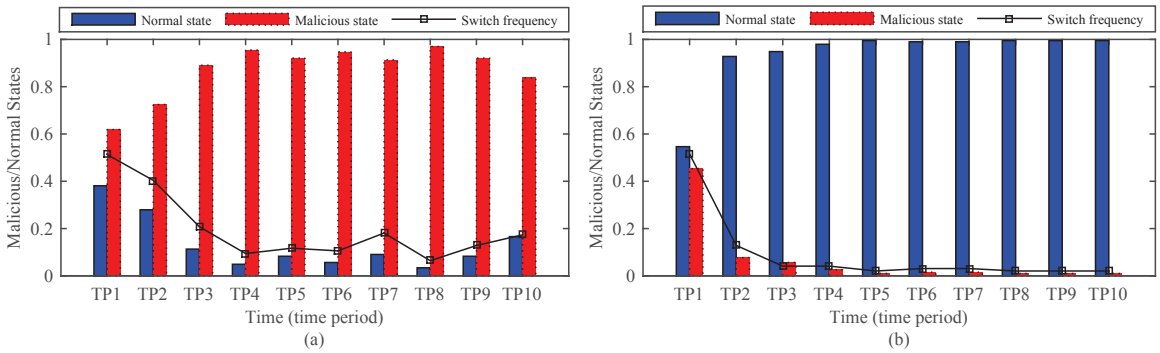


Fig. 3.1.13.: Model output and frequency of switches between the states: (a) the output of model for a given malicious app; (b) the output of the model for a given normal app.

500 malicious and 500 benign apps from both datasets (malicious and benign) and use them as *training sets* for the model. The remaining 200 apps from each dataset are used as *test set* to test the performance of the trained model.

In terms of behaviour logging, we set the timer in the DroidCat's App Scheduler to $2-5$ minutes per app and it took around 79 hours to capture all logs through human interactions with the 1400 apps.

3.1.8.2 The Running States of Malicious and Benign Apps

In order to demonstrate how the HMM switches between normal and malicious states during the risk-assessment process, we randomly selected two apps from the malicious and normal datasets. We used their behaviours logs as input and used the Viterbi algorithm to generate output sequences. We divided the outputs into 10 time-periods (T). Figure 3.1.13 (a) and (b) show the probability of normal and malicious output states and the frequency of switching in between them, given the behaviour sequences collected from the apps. We observed that the model switches frequently in between the normal and malicious states at the beginning (as a sign of uncertainty of categorization), and after receiving sufficient data it converges to a malicious (normal) state. Note that we define the *switch frequency* to be the percentage of transition states over the total number of states in the output sequence.

For example, let m_t be the number of times that the state switches from normal to malicious or from malicious to normal, and m_o be the overall number of output states observed, then we have the switch frequency $f = m_t/m_o$. Note that $0 \leq f \leq 1$. We observed that the output sequence from different apps follow the same pattern on convergency.

We also computed the average probability of being in normal or malicious states using probability mass function (PMF) for the apps in the previous experiment. Table 3.1.4 shows the probabilities of normal and malicious output states for both malicious and normal apps. We can see that the model results in high percentage of normal (93%) state for the benign app, and high percentage of malicious (86%) state for the malicious app.

Table 3.1.4.: Probability Mass Function results

State	Normal	Malicious
PMF (Normal app)	93%	7%
PMF (Malicious app)	13%	86%

3.1.8.3 Model accuracy and reliability

In this experiment, we study the accuracy of the model. After training the HMM, we measure the true positive rate (TP) and false positive rate (FP) regarding the computed risk levels' accuracy. TP refers to probability that malicious apps risk levels are computed correctly by the model, whereas FP is the probability that a benign app risk level is falsely computed. Note that true negative rate (TN) and false negative rates (FN) can be derived from TP and FP. We start with the risk threshold from 0 and increase it by 0.05 each round till it reaches 1. Figure 3.1.14(a) shows that TP and FP drop when the risk threshold increases. The FP and TP drop rate increase drastically when the threshold passes 0.5 and 0.6, respectively. From Figure 3.1.14(b) we can see that the TP and TN cross at around 90% accuracy when the threshold is around 0.7.

We applied the trained HMM model to the test set with 200 malicious and 200 benign

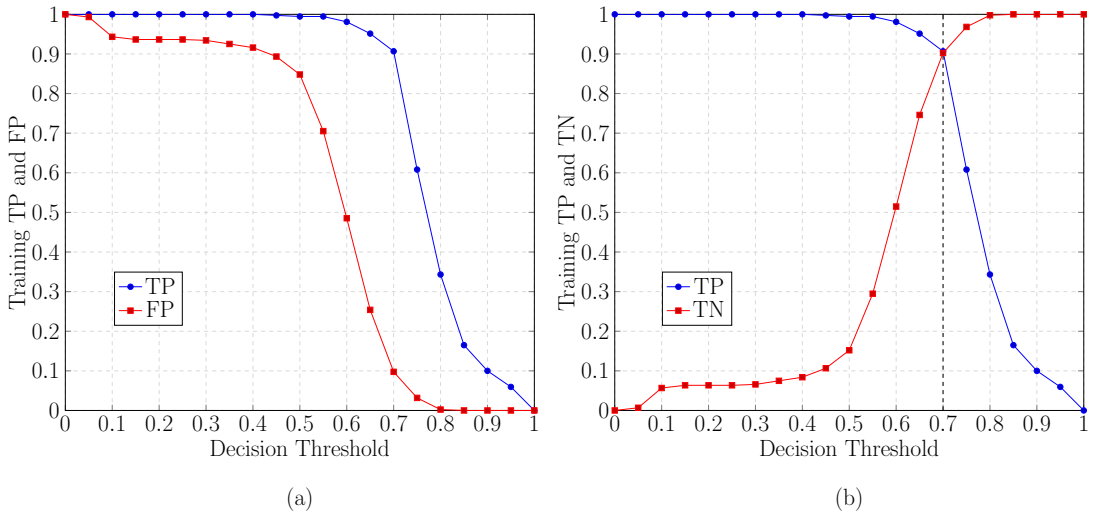


Fig. 3.1.14.: Accuracy of the model on the training sets

apps, and show the results in Figure 3.1.15. As we can see, the accuracy is slightly lower than the ones on the training set. From this experiment, we can see that our model achieves high accuracy to compute the malicious apps' risk levels. When the risk threshold θ is 0.7, the risk assessment system achieves 88% accuracy on TP and TN on the training set and 80% TP and TN on the test set. In the merged results presented in Figure 3.1.16, we can see that the false positive rate is higher in the test set than the training set.

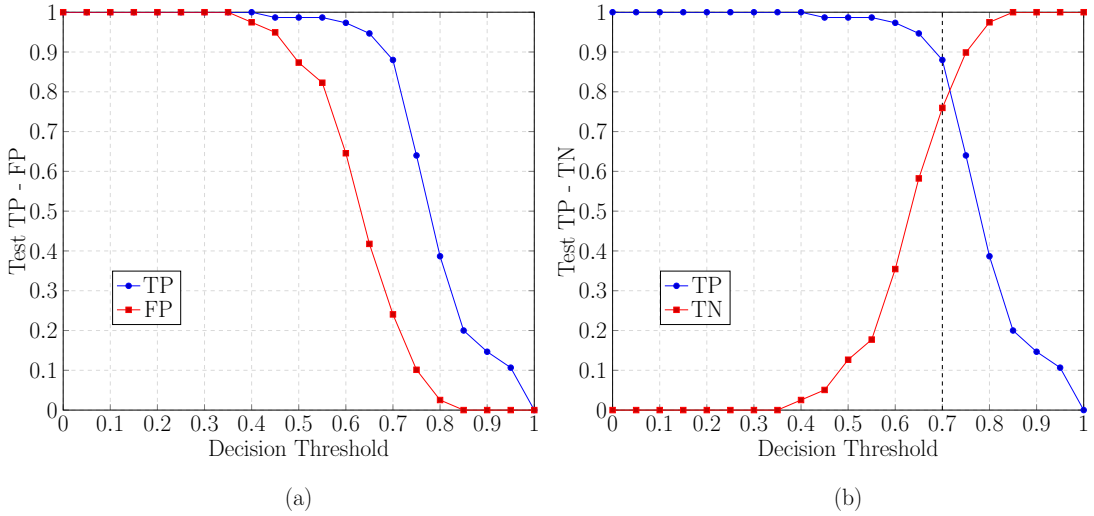


Fig. 3.1.15.: Accuracy of the model on the test sets

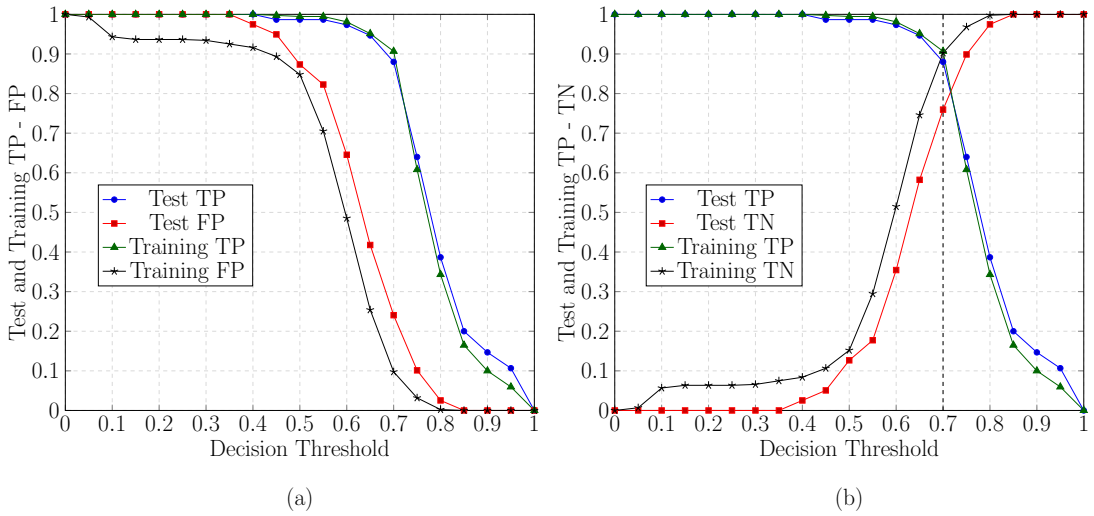


Fig. 3.1.16.: Accuracy of the model on both test and training sets

Considering 0.7 as an ideal risk threshold, we carried out another experiment to study the influence of training dataset size on the TP and TN. In this experiment, we cross-validated the model by splitting the training data into different sizes. We trained the HMM model with one set and test the result using the other. The size of the training set start from 100 and increases by 100 each round. Figure 3.1.17 shows that the TP and TN increase with the size of the training dataset. We also see that when the training datasets reaches 800 the accuracy of risk computation does not get better by increasing the training dataset, which means the training dataset of 800 (400 malicious apps and 400 benign apps) is sufficient.

Table 3.2.2 presents the performance of the model on the test dataset in terms of Recall, Precision, F-Measure and Accuracy for all ten training dataset sizes. We can see that all performance indicators increase with the training dataset size until it reaches 800.

3.1.8.4 Risk evaluation

In this section we study the risk levels of apps and their resources through experiments. We discuss the risk computation and the impact of users' preferences on the computed risk levels. We use cross validation to evaluate the impact of the parameter η_k (forgetting

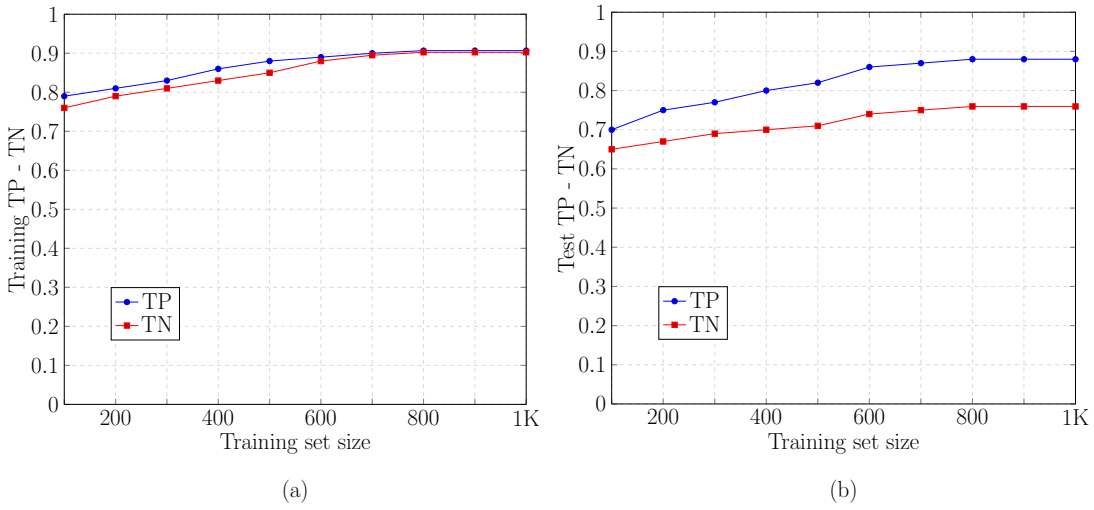


Fig. 3.1.17.: Accuracy of the model on both test and training sets with different set sizes

Table 3.1.5.: Performance Measurement - Recall (Rc), Precision (Pr), F-Measure (F), Accuracy (Ac)

Size	TP	TN	FP	FN	Rc	Pr	F	Ac
100	0.70	0.65	0.35	0.30	0.70	0.67	0.68	0.68
200	0.75	0.67	0.33	0.25	0.75	0.69	0.72	0.71
300	0.77	0.69	0.31	0.23	0.77	0.71	0.74	0.73
400	0.80	0.70	0.30	0.20	0.80	0.73	0.76	0.75
500	0.82	0.71	0.29	0.18	0.82	0.74	0.78	0.77
600	0.86	0.74	0.26	0.14	0.86	0.77	0.81	0.80
700	0.87	0.75	0.25	0.13	0.87	0.78	0.82	0.81
800	0.88	0.76	0.24	0.12	0.88	0.79	0.83	0.82
900	0.88	0.76	0.24	0.12	0.88	0.79	0.83	0.82
1000	0.88	0.76	0.24	0.12	0.88	0.79	0.83	0.82

factor) on the estimated risk levels for apps' resources access. The η_k rate for the first three experiments is set using the *time-variant* polynomial form.

In the first experiment we compute the average risk levels for all malicious and normal apps in our datasets. Figs. 3.1.18 (a1) and (a2) show the results of the risk computation for two app groups, malicious and normal apps respectively. We can see in the Figure 3.1.18 (a) that the computed risk levels for normal apps start from near 0 to 0.8, and risk levels for malicious apps start from 0.4 to 1. The distribution of the computed risk levels is presented in Table 3.1.6. We can see that 450 (90%) of the malicious apps have risk levels higher than 0.7, whereas only 30 (6%) normal apps risks are above this level. The results

of this experiment show that risk level is an effective criteria to separate malicious apps from normal apps. Figure 3.1.18 (a) also show the normal distribution ($\mathcal{N}(\mu,\sigma^2)$) of the computed risk levels using the *Gaussian* function. The median (μ) and standard deviation (σ) of the risk levels for the malicious and normal apps are ($\mu = 0.79, \sigma = 0.1$) and ($\mu = 0.56, \sigma = 0.14$) respectively.

Table 3.1.6.: Risk level distribution

Type	**0-0.1**	**0.1-0.3**	**0.3-0.5**	**0.5-0.7**	**0.7-0.1**
Normal	25	4	97	344	30
Malicious	0	0	2	48	450

In the second experiment, we study the impact of the user responses on the average risk level of an app. We define a scenario where we target the risk level of the network resource of a malicious app. Without any user input, the risk level of the resource is 0.9. We plot the change curves of the estimated risk levels of the same resource before and after users chose to "allow" or "block" the resource access. For the sake of clarity of figures we rescaled the app's running time from 0 to 1. At the beginning there is no log input for the resource so that risk level is 0. Then we start to feed log files of the malicious app and the risk level of the app increases drastically to 0.9. At time 0.3 we inject two types of responses to the system and observe its impact to the risk value of the app. Figure 3.1.18(b) illustrates the impact of the user's response to the estimated risk levels. We can see that if the user's response is "Allow", the model turns to be less conservative and the risk level of the resource decreases. On the other hand, if user's response is "Block", the risk level increases (the user is more conservative). When no user response is in place, the risk level of the app remains at around the same level. From this experiment we can see that the risk assessment model can adapt to the user's responses and provide customized risk levels.

In the third experiment, we still use the same scenario as the last experiment, but this time we focus on the estimated risk levels for the network resource only. Figure 3.1.18(c)

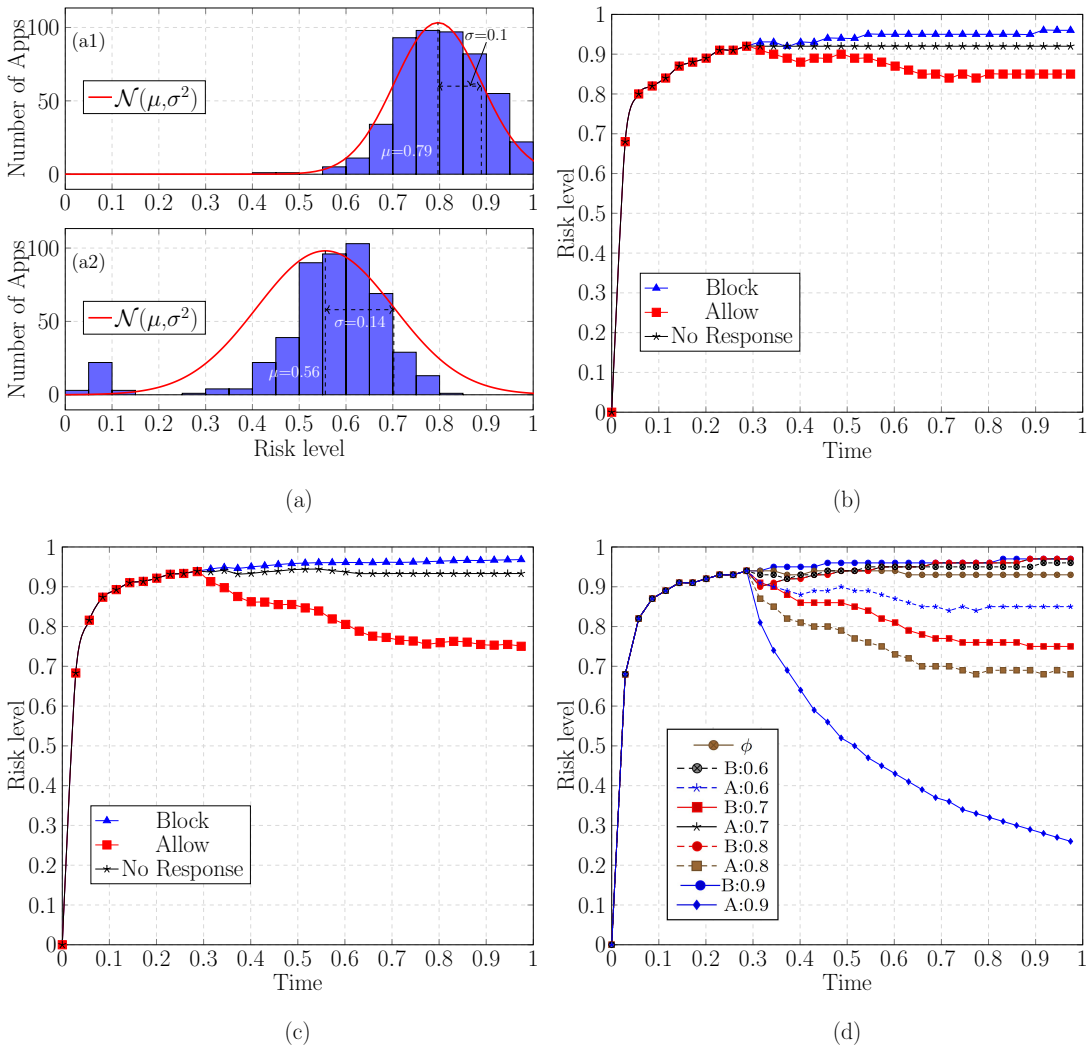


Fig. 3.1.18.: Applications' computed risk levels: (a) the distribution of risk levels for both malicious and normal apps training datasets; (b) the impact of user's response on the overall risk level; (c) the impact of user's response on the risk level of a resource; (d) the impact of the forgetting factor on the computed risk level of a resource

shows the computed risk level for both "Allow" and "Block" responses. We can see that this result also shows the influence of the user response. However, the risk level drops faster in the case that "Allow" is chosen by the user. The impact from user response is higher since it is focused on one resource only.

The last experiment is to study the impact of the learning rate parameter η_k (forgetting factor) on the computed risk levels by the system. In this experiment we ran the experiment

multiple times with different settings for η_k. We let η_k change from 0.6 to 0.9 and observe its impact on the risk values. Figure 3.1.18(d) shows that the higher η_k is, the higher impact user's responses have on the risk levels. This is because higher η_k means more emphases on the recent input, which is the user's responses.

Table 3.1.7.: Resource average usage statistics

Resource	Malicious	Normal
Ads API	115.3	11.8
Bluetooth	3.2	0.6
Browser	5	0.86
Call	4.8	0.87
IMEI	3.7	2.4
MMS	2.5	0.5
SMS	3.4	0.1
Root command	14.3	3.2
Boot inquiry	2.4	0.1
Zygote	7.6	0.9

Finally, we did a statistic analysis on the collected logs from the training and test datasets and computed the average resource usage per app for both malicious and normal apps. Table 3.1.7 shows the result and, as we expected, malicious apps requested and used sensitive resources more than normal apps.

3.1.9 Conclusion

In this section, we propose XDroid, an Android app resource access risk estimation framework using hidden Markov model. We first define and select features to represent behaviours of Android apps and collect them through our own developed human-oriented instrumentation tool DroidCat. A filtering and parsing method is then employed to synthesis and organize the captured behaviours. We train the model with a proper malicious app dataset using the Baum-Welch algorithm and test it with different test datasets using the Viterbi algorithm. Through our model, we can compute the risk level of those apps that behave malicious with high level of accuracy. The model informs users the risk level of their apps in real-time. Our model is able to update the model's parameters dynamically using

an on-line algorithm and users' preferences. Our experimental results demonstrate that our proposed model achieve a satisfying accuracy in terms of true positive and false positive rate. Our evaluation results also show that our model can effectively provide customized risk estimations depending on users' preferences. As our future work, we plan to apply more external features to further improve the detection accuracy. We also plan to include the users' expertise level into the risk level computation process for better accuracy.

3.2 Malware Detection Using Support Vector Machine and Active Learning

In this section, we elaborate the details of a machine learning-based model which is aim at detection malicious apps. The model detects malicious apps using a dynamic behavioral-based analysis. Our proposed model is equipped with an online learning feature which enables online training and updating our model. This way, the model is able to self-tune itself against new types of malicious apps. The major contributions of the work reported in this section include:

- An instrumentation tool that facilitates app behaviour logging in order to generate high quality dataset for analysis.
- A comprehensive time-aware Android app behaviour analysis, which is based on the apps' intents and actions, as well as extra features that further improve detection accuracy.
- A trained SVM model which can decide whether an app is malicious or not based on its behaviour.
- An Active Learning model which can retrain the SVM model to be able to detect malicious apps with behaviours different than the apps in training set.

3.2.1 Problem Definition

The increasing popularity of Android phones and its open app market system have caused the proliferation of malicious Android apps. The increasing sophistication and diversity of the malicious Android apps render the conventional malware detection techniques ineffective, which results in a large number of malicious applications remaining undetected. This calls for more effective techniques for detection and classification of Android malware. Hence, in this project, we present an Android malicious application detection framework based on the *Support Vector Machine* (SVM) and *Active Learning* technologies. In our approach, we extract applications' activities while in execution and map them into a feature set, we then attach timestamps to some features in the set. We show that our novel use of time-dependent behaviour tracking can significantly improve the malware detection accuracy. In particular, we build an active learning model using *Expected error reduction* query strategy to integrate new informative instances of Android malware and retrain the model to be able to do adaptive online learning. We evaluate our model through a set of experiments on the DREBIN benchmark malware dataset. Our evaluation results show that the proposed approach can accurately detect malicious applications and improve updatability against new malware.

3.2.2 Background

In this section we briefly review some background knowledge about *Support Vector Machine (SVM)* and *Active Learning*, including how an SVM model works and can be evaluated. We also explain how active learning can help update the model by incoming new instances.

3.2.2.1 Support Vector Machines

A Support Vector Machine (SVM) is a discriminative classifier formally defined by a separating hyperplane [24]. In other words, given labeled training data (supervised learning), the algorithm outputs an optimal hyperplane which can be used to categorize new data examples. An SVM training algorithm builds a model that assigns new instances to one class or another, making it a non-probabilistic binary linear classifier. Regardless of the dimensions of the sets (finite or infinite), if the input sets are not linearly separable, SVM maps the original sets into a higher-dimensional space, presumably making the separation easier. This transformation to a high-dimensional space increases the computational load [24].

To reduce the computational load of the dot product operation which is needed in the dimension transformation and improve the accuracy of classifying data sets, SVM uses *kernel* functions. A kernel function helps accelerate the dimension transformation computation [24]. The mathematical definition of a kernel function is as follows:

$$K(x, y) = \langle (x), f(y) \rangle \tag{3.39}$$

where K is the kernel function, x, y are n dimensional inputs, f is a map from n-dimension to d-dimension space (d is much larger than n). In this equation, $\langle x, y \rangle$ denotes the dot product. In other words, a kernel function can also be understood as a measure of similarity between two data points. For example, a kernel function K takes two data points x_i and $y_j \in \mathbb{R}^d$, and produces a similarity score, which is a real number, i.e., $K : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$.

SVM has some advantages that make it unique. Some of the advantages are: (i) it has a lower computational complexity, (ii) it is effective in cases where the number of dimensions is greater than the number of samples, (iii) it uses a subset of training points in the decision

function (called support vectors), so it is also memory efficient, and (iv) different kernel functions can be specified for the decision function. Next we elaborate the two key steps of SVM: training and evaluation [69].

Training: In order to classify datasets, the SVM model needs to be trained. As we described previously, SVM is a supervised learning model. The supervised learning is the process of inferring a function from labeled training dataset. The training dataset shall consist of a set of data points together with their labels (classes). The training dataset is then used by SVM to produce an inferring function, which can be used for classifying new instances [69].

In addition to the training set, a kernel function needs to be selected for the SVM model. The effectiveness of the selected kernel depends on the training datasets. The kernel selection and its regularization parameters is a challenging issue. Model overfitting may occur if the kernel model or its parameters are not selected appropriately. There are a few options for kernels such as Linear, Radial Basis Function (RBF), and Polynomial. Formal definition of these kernels are listed in Table 3.2.1.

Table 3.2.1.: Kernel Definitions

kernel	**Mathematical Formulation**
Linear	$K = (X, Y) = X^T Y$
Polynomial	$K = (X, Y) = (\gamma \cdot X^T Y + r)^d, \gamma > 0$
Radial Basis Function (RBF)	$K = (X, Y) = exp(-\gamma \cdot \parallel X - Y \parallel^2), \gamma > 0$

In the table r, d, and γ are the coefficient value, degree of polynomial, and the influence of a single training example. When training an SVM with the RBF kernel, two parameters must be considered: C and γ. The parameter C, which is common to all SVM kernels, controls the trade off between the misclassification of training examples and the simplicity of the decision surface. γ defines how much influence a single training example has. The larger γ is, the closer other examples must be to be affected.

Evaluation (Validation): In order to evaluate the performance and accuracy of a SVM model, cross-validation can be used. Cross-validation is a technique to assess how a statistical analysis will be generalized to an independent set. Cross-validation has different types such as *Leave-p-out*, *k-fold*, and etc [69, 24]. In the leave-p-out technique we use p data points as the validation set and the remaining data points as the training set. The model can be evaluated for different values of p. In k-fold validation, the dataset is randomly sliced into k equal sized data chunks. Of the k chunks, one chunk is retained as the validation set for testing the model, and the remaining $k - 1$ chunks are used as training data. The cross-validation process is then repeated k times (the number folds), with each of the k chunks used exactly once as the validation data. In the evaluation section of this project, we evaluate our model using these techniques.

3.2.2.2 Active Learning

Active learning is sometimes called "query learning". It is a special case of semi-supervised machine learning in which a learning algorithm is able to interactively query the user (or some other information source) to obtain the desired outputs at new data points [27]. Active learning technique can be used as a tool to retrain a machine learning model with new instances (unlabeld data points). For example, to be able to detect new trends of "spam" in an email service, newly flagged emails as spam by users can be considered as new instances to retrain machine learning models. Here mailing service users are called "Oracle" and active learning techniques utilizes their opinions to label new instances and add them to the training sets. The labeling process refers to the process of measuring app's risk by experts and labeling them as malware or benign. In other words, the key idea behind active learning is that a machine learning model can achieve higher accuracy if it is allowed to choose the instances from which it learns. This is why the model chooses instances with a higher level of informativeness and achieve higher accuracy con-

secutively [72, 27].

There are several different problem scenarios in which the learner (model) may be able to ask queries on new data instances. Out of all the active learning techniques, the major two are the *stream-based selective sampling* and the *pool-based sampling*. The former is used when the model queries the unlabeld instance in an online (real-time) manner and the latter is used when new instances are stored in a pool (collection) of data and the learner queries the pool to label when needed. In our proposed model we use the stream-based technique. Next we elaborate the formal definition of active leaning and its query strategy [36].

Formal definition: Let T be set of all data in the original dataset. For example, in our model, T includes all apps that are known as malware or benign. During each retraining, say the i^{th}, the dataset T is split into three sub-datasets: T_{Ki} - known data points, T_{Ui} - unknown data points, and T_{Ci} - a subset of T_{Ui} selected to be labeled by the Oracle. If the active learning technique is based on a stream-based learning, then $\mid T_{Ui} \mid = 1$.

Query strategy: Query strategy, also called *"utility measures"*, is the process of choosing new incoming data instances to retrain the model. In other words, this process should determine which data point should be labeled [72]. There are several query strategies in active learning. For example, a query strategy based on *"uncertainty sampling"* selects query instances which have the least label certainty under the current trained model. This simple approach is not computationally expensive compared to others.

3.2.3 Support Vector Machine Model

In this work, we use SVM and active learning for Android malicious app detection. We model the malicious app detection problem as a machine learning problem with two classes: *malicious* and *benign*. We map the app's behaviour onto the SVM training dataset

features. To train the SVM model, we capture the behaviours from both malicious and benign apps and use them to generate an initial trained SVM for malicious app detection. In this section we first present our SVM model and then explain how we can retrain the model using new instances to be able detect new types of malicious apps.

3.2.3.1 Data Collection

Our vision to specify app behaviour is to view the running app as a black-box and focus on its interaction with the Android OS. In this case, a typical interface to monitor is the set of system and API calls that the app invokes during its running time. Every action that involves communication with the apps' resources (e.g., accessing the file system, sending SMS/MMS over the network, accessing the location services, calling Ads API libraries, and accessing the network) requires the app to launch OS services or API calls.

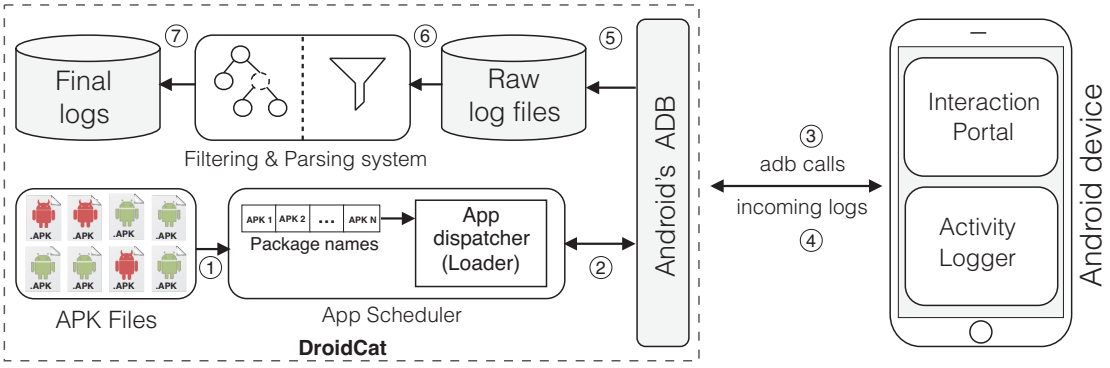


Fig. 3.2.1.: DroidCat instrumentaiton tool architecture

We used *DroidCat* instrumentation tool to capture apps' logs (behaviours). The reason we did not use the existing instrumentation tools in the market such as Robotium and uiautomator was because of their drawbacks and low accuracy. For example, Robotium cannot handle Flash or Web components nor simulate the clicking on soft keyboard, and it is not suitable for multi-process applications tests. Using DroidCat, we can capture the actual log activities of apps. DroidCat's architecture is illustrated in Figure 3.2.1. One of the main advantages of DroidCat is that it instruments apps through real human-interaction, so we

capture the actual activities of apps which highly assimilate real-life apps' behaviours. As you can see in the architecture, DroidCat is composed of multiple components. Every one of the components is in charge of a task.

The first task is to extract the packages' names. We used `aapt` tool to accomplish it. The tool is designed to work with archive files. Since Android apps are in the format of APK (a type of archive files), we utilize the *appt* tool to read archive files. In addition, because this tool is embedded into the Android SDK, it does not impose a high performance overhead to the process.

After reading the packages' names and recording them, out next step is to run the apps. In order to run an app, we should load it into the device's memory. We used ADB `logcat` tool to load the apps. The loading process also includes installing the apps and running them as well. *App dispatcher* component is in charge of the loading process. This component also determines the amount of time that the apps should run. By running the apps, we are able to capture the activities logs and record them at the time of instrumentation.

We call the collected logs from the previous step "raw" logs. The raw logs need to be filtered in order to eliminate the unnecessary information such as loading/installing/running logs.

Parsing: After filtering the log files, DroidCat eliminates unnecessary information and extracts important keywords. Each keyword refers to a sensitive resource access request, an API call, or Android action constants. We can also call them *features*. In our model, we focus not only on the generated Intents by apps but also on API library calls that cause permission escalation or generate unwanted Ads. In total we defined 150 keywords under various categories. When analyzing the parsed logs, we noticed that for some resources such as "WiFi", malicious and benign apps have different patterns in the timing of requests during app running. For example, the malicious apps tend to request the WiFi network

during the first quarter of their running time period. Because of this, we include the timing of requests or library calls as an additional feature. Among of the 150 keywords, we added the timing feature to 56 of them. Therefore, we defined 206 *time-dependent* and *time-independent* observations in total.

3.2.3.2 Model Building

In this section we describe our RBF-based SVM model and its components. We start from the motivation of using SVM as a malicious app detection method and why RBF works the best for the model. We also elaborate the active learning component of the proposed model and its query strategy in details.

Figure 3.2.2 shows the overall view of our model. The key components of the model are *Model building*, *Model evaluation*, *Model optimization*. The model also has the capability of active learning using *Informativeness Measurement* and Oracle. The decision model is trained by a set of instances, called *Historical Data*, which is used as training set for the model. For new incoming instances, our model is able to label them using Oracle for active learning.

3.2.3.3 Model Training

For our model we choose SVM for classification algorithm. This is due to the large number of captured features from the data (206 features). It is difficult to find a separation boundary using other machine learning classifiers or clustering algorithms. A major advantage of SVMs is that the data can be transformed to a high-dimension space, where we can find a separation hyperplane using linear or RBF kernels.

By visualizing our collected data, we noticed that our data is not linearly separable. This is common when the training dataset has a large number of features. Figure 3.2.3 illustrates our training dataset using two features. The X axis and Y axis are the features.

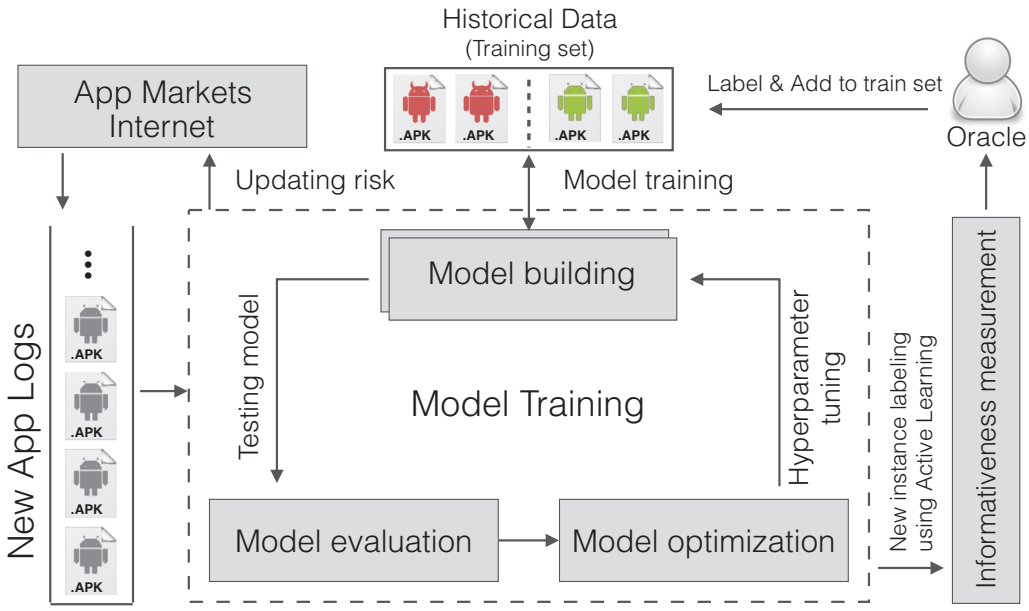


Fig. 3.2.2.: SVM model and the active learning component architecture

We trained our classification model using a sample set of our data with both linear and RBF kernels and plotted the separation boundaries. We can see that the RBF-based model can achieve cleaner separation compared to the linear kernel. Therefore, we use RBF as the kernel function in our model.

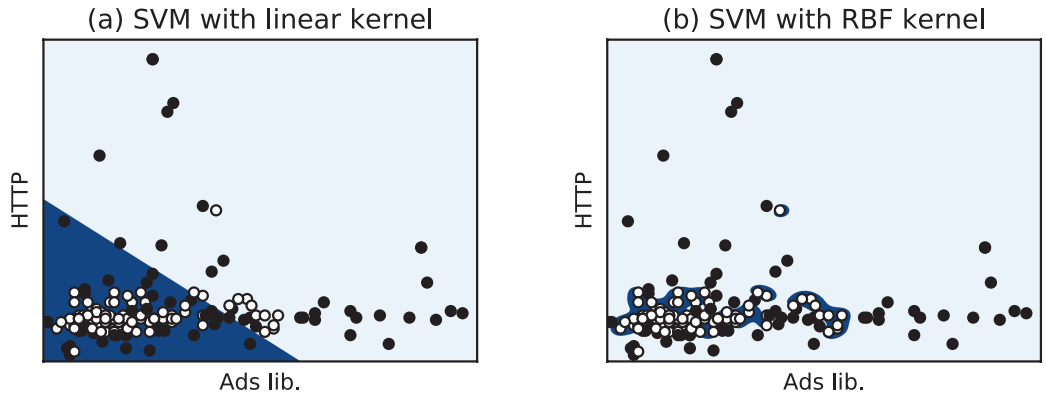


Fig. 3.2.3.: Radial-Basis Function (RBF) and Linear kernel comparison on a sample dataset

3.2.3.4 Active Learning

We also integrate active learning into our model to be able to use new apps as training data points. To design an active learning model, two critical questions must be answered: 1) How often the model should be retrained to keep the detection rate high?, and 2) what query strategy should be used? We elaborate our answers to these questions in what follows.

Learning: Our strategy is to use a stream-based learning. This way the model queries new apps instantly and makes a decision on whether it should be labeled by the Oracle or not. There are two different types of stream-based learning: *Online learning* (one new app at a time) and *Batch learning* (a group of apps at a time). In online learning, the model decides whether to discard or process (label) a new data. In batch learning, the model collects new data until it reaches a certain size (batch size) and then decides whether to process them or not. Figure 3.2.4 (a) and (b) shows an overview of both online learning and batch learning respectively.

Query Strategy: Query strategy is the most influential part of an active learning-based model. Choosing a good strategy increases the accuracy of the model for future unknown apps. To achieve this goal, we chose the *Expected error reduction* strategy to query new apps. This strategy aims at labeling apps that minimizes the model's future generalization error. The idea is to estimate the expected future error of a model trained using the original dataset $\mathcal{L} = \langle x, y \rangle$ and test it with the remaining data $\mathcal{U}$ (new data) and then query the new apps with minimal expected future error. An approach to minimize the expected 0/1-loss is as follows:

$$x_0^*/1 = \operatorname{argmin} \sum_{u \in \mathcal{U}} E_y[H_{\theta^+\langle x,y \rangle}(Y \mid u)] \tag{3.40}$$

where $\theta^{+\langle x,y\rangle}$ denotes the updated model after it has been retrained with the training app $\langle u_x, u_y\rangle$ added to $\mathcal{L}$. In this equation, E_y and $H_{\theta^{+\langle x,y\rangle}}$ are the expectation over possible labeling of x and uncertainty of u after retraining with x respectively. Here, since we do not know the actual label of the query app, we approximate the label using expectation over all possible labels under the current model θ.

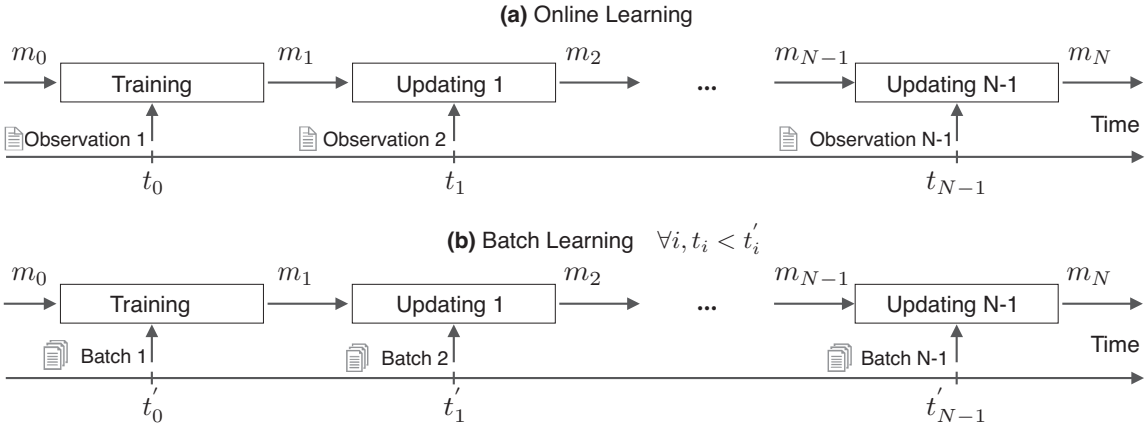


Fig. 3.2.4.: Illustration of Batch learning and Online learning

3.2.4 Evaluation

We present our evaluation results in this section. More specifically, we measure the accuracy of the model and then evaluate the stability of the model by adding noise to the training set. After that we evaluate the model under different settings of features using K-Best features. Finally we evaluate the active learning model under different settings of Oracle's expertise and batch sizes.

3.2.4.1 Experiment Setup

Software and Hardware Our experiment environment is python 3.6 running on same machine. We implemented all of our experiments using `scikit-learn` libraries powered by Google [57]. The libraries that are used in our experiments include *model_selection*,

train_test_split, *confusion_matrix*, *SVC*, *cross_val_score*, *SelectKBest*, and etc. It is worth mentioning that to capture the actual logs of apps in our experiments, we used 4 LG Nexus 4 smartphones. We turned on all the sensitive resources such as Wifi, Bluetooth, and GPS.

Datasets In order to conduct our evaluation experiments, we used a Android malware dataset called Drebin project [13]. The dataset includes more than 5K apps from 179 different malware families. We selected 700 apps from this dataset so that we have multiple apps from all the malware families. In addition to the 700 malicious apps, we collected 700 benign apps from various categories of Android apps. We randomly selected 500 malicious and 500 benign apps from both datasets (malicious and benign) and use them as *training sets* for the model, and the remaining 200 apps from each dataset are used as *test set* to test the performance of the model.

We defined the running time per app to be $2-5$ minutes in the App Dispatcher component. In total, the log capturing process (instrumentation) took around 79 hours for all the apps through human interactions.

3.2.4.2 Training Dataset Visualization

Before presenting the validation results, we visualize our data set using four features pairwisely. We selected permission manipulation, Ads lib. usage, HTTP request, and Activity manager interference features from the original dataset. Figure 3.2.5 shows the visualization results. In this figure, the x and y axis are a pair of the selected features representing the number of times that malicious and benign apps have requested those resources. We can see from the figures that the distribution of the malicious apps is more diverse than benign apps. Regardless of some differences, there is some overlap among malicious and benign apps. To visualize the distributions, we visualize the average usage of resources in Figure 3.2.6. We can see the comparison of the average resource usage by malicious and

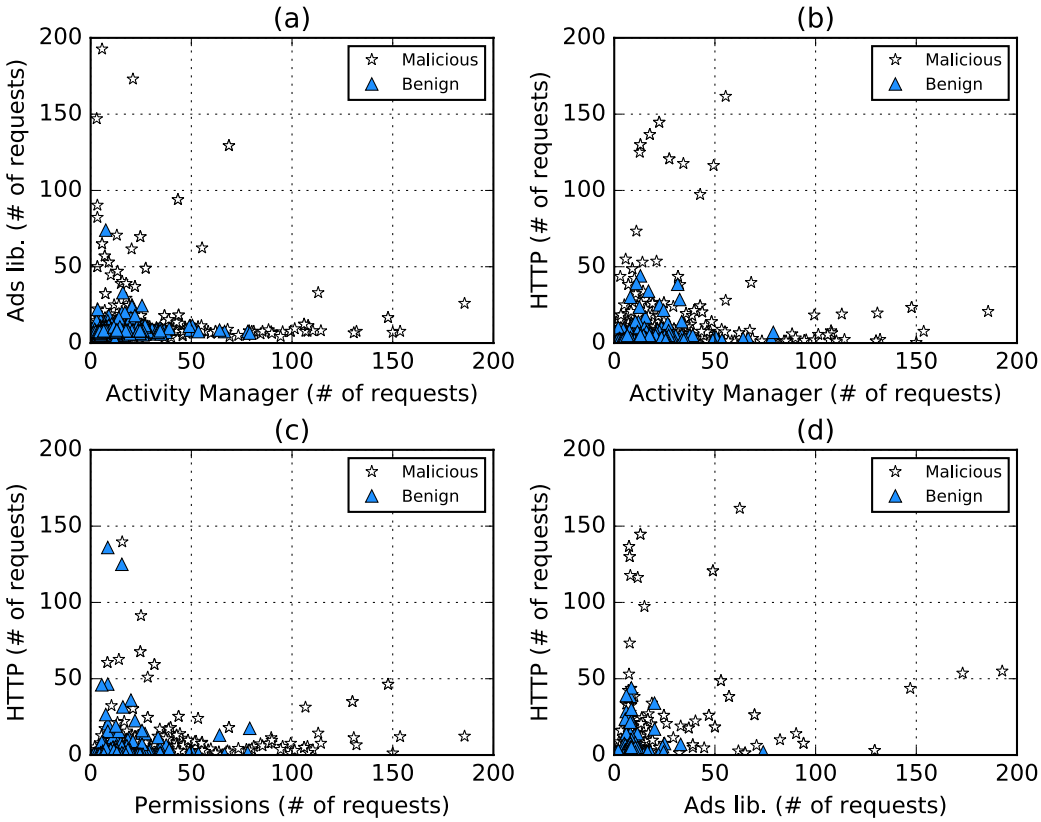


Fig. 3.2.5.: Pairwise visualization of the dataset used for training and testing the model by four app behaviours (permission request, Ads lib., WiFi, and Activity Manager) as axises

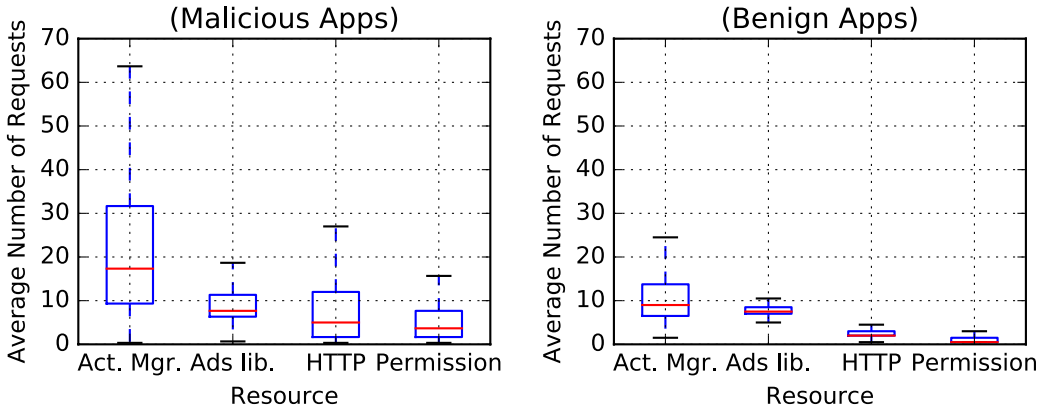


Fig. 3.2.6.: Visualizing average resource request by malicious and benign apps for permission request, Ads lib., WiFi, and Activity Manager: (a) malicious apps; (b) benign apps.

benign apps. The blue boxes in the figures represent the data range from the second quarter to the third quarter of the data sample, while the red bars are the medium values of the samples. The vertical whiskers indicate the range of all data except outliers. The outliers

are not plotted in the figures (which can be very large number).

3.2.4.3 Model Accuracy and Reliability

In the first experiment, we assess the accuracy of the model. We first tuned the model to find the best parameter configuration. Since we use RBF as our kernel function, we need to find the best values for C and γ. We used `scikit-learn` *model_selection* library to find the optimal kernel function and values for parameters. As a result, RBF was selected as the optimal kernel with $C = 1$ and $\gamma = 5.5e^{-4}$. After finding the optimal configuration for the model, we cross-validated our model on the training set to measure the average accuracy μ and standard deviation σ. Figure 3.2.7 shows the cross-validation results under different settings of C. We can see that with optimal setting for γ when $C = 1$, we have the highest average accuracy above 90% and a low deviation (Figure 3.2.7(a)) and when $C = 0.1$ (model is not tuned), the average accuracy drops and the standard deviation is higher (Figure 3.2.7(d)).

The second experiment is also on the model accuracy. We evaluated the reliability of the model using the leave-p-out validation technique. We split the training set into 10 subsets and train the model with one set or multiple sets. We increase the size of the training set to start from 10% and increases by 10% each round. We validated the accuracy using the training as well as our validation sets. We ran the experiment for 10 times to plot the average accuracy and confidence interval in Figure 3.2.8(a). The results show that by increasing the training set size, the accuracy of model increases. We can also see that the accuracy of the model on the test set is lower than that of the the training set. The reason that we achieved such high accuracy is that, in contrast with existing approaches, we consider time in our features. We also see that with only 50% of the training set, the model is able to predict almost as accurately as that with 100% on the training set, which shows the reliability of the model.

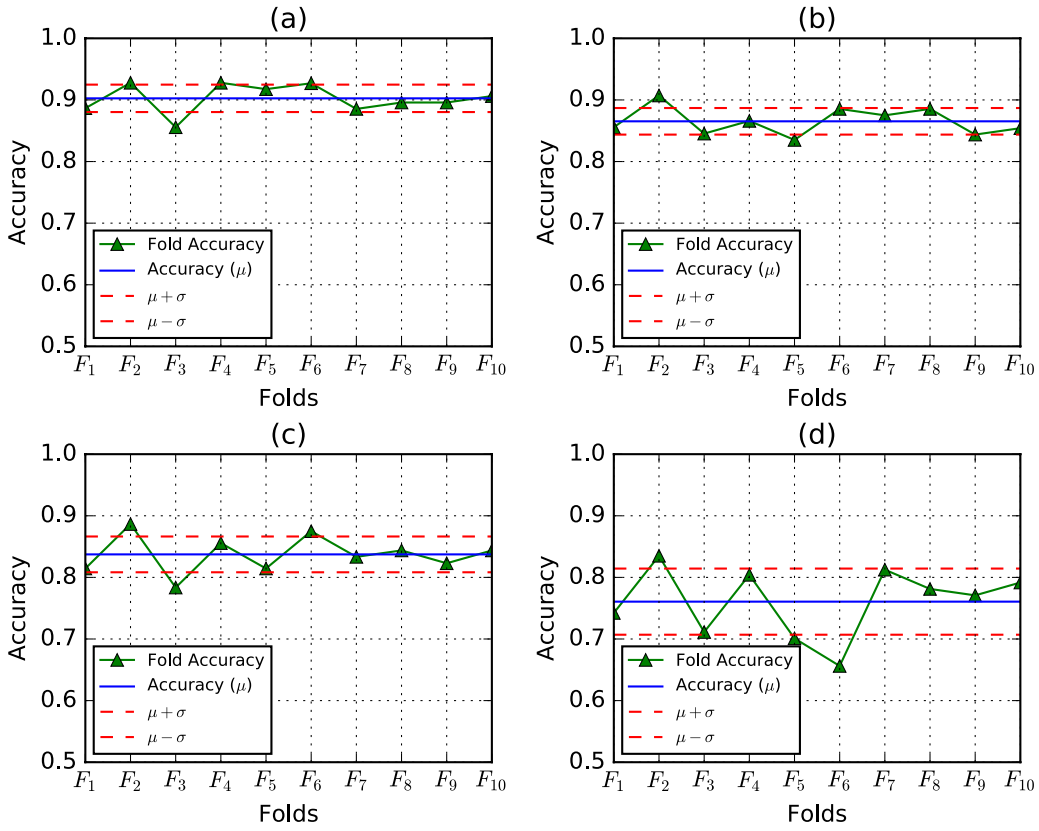


Fig. 3.2.7.: Evaluation of model's accuracy using Cross-Validation and parameter (γ, C) tuning: (a) $(\gamma = 5.5e^{-4}, C = 1.0)$; (b) $(\gamma = 5.5e^{-4}, C = 0.5)$.(c) $(\gamma = 5.5e^{-4}, C = 0.3)$; (d) $(\gamma = 5.5e^{-4}, C = 0.1)$.

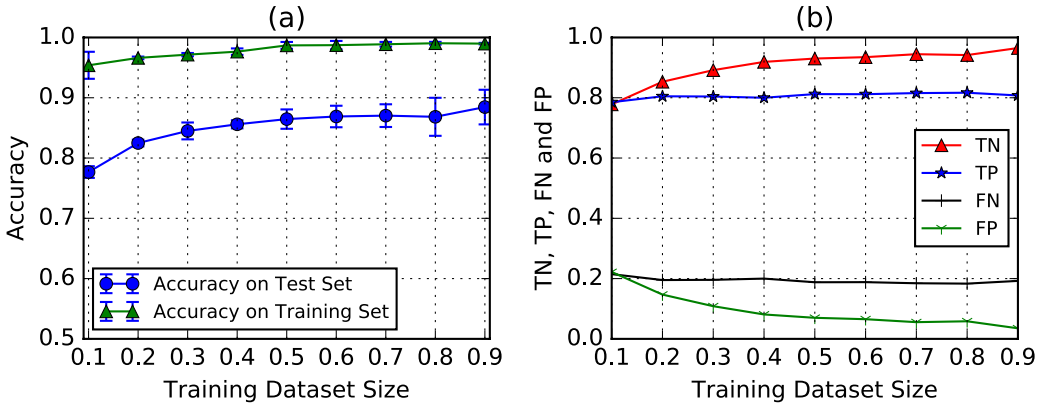


Fig. 3.2.8.: Model accuracy evaluation: (a) accuracy of the model with different sizes of training dataset and evaluated by training and test datasets for 10 runs; (b) evaluated True Positive (TP), True Negative (TN), False Positive (FP), and False Negative (FN) of the model.

We also carried out another experiment on the accuracy by measuring the True Positive (TP), True Negative (TN), False Positive (FP), and False Negative of our perdition.

Figure 3.2.8(b) shows the evaluation results. Table 3.2.2 shows the performance of the model on the test dataset in terms of Recall, Precision, F_1-Measure for different training set sizes. We can see that all performance indicators increase with the training dataset size.

Table 3.2.2.: Performance Measurement - Recall, Precision, F1-Measure

Size	Recall	Precision	F1-Measure
10%	0.78	0.80	0.79
20%	0.80	0.86	0.83
30%	0.80	0.89	0.84
40%	0.80	0.91	0.85
50%	0.81	0.93	0.86
60%	0.81	0.93	0.87
70%	0.81	0.94	0.87
80%	0.82	0.94	0.87
90%	0.83	0.96	0.88

3.2.4.4 Model Stability Evaluation

In this subsection we assess the stability of the model. We define stability as the robustness of the model against noisy or incomplete malicious apps behaviours added to the training set. To evaluate the stability, we generated noisy data and mixed it with the training set. We generated the noise using the *Gaussian* distribution ($\mathcal{N} = (\mu, \sigma)$), a very common noise in machine learning validation process with different strengths 1 to 10. The generated noise is a vector of numbers in the size of ($|Features|$) that is added to the data points. Figure 3.2.9 shows the generated noise for this experiment. We also added the noise to the training set under different size settings from 5% of the dataset to 20% to see the effects of the noise if different portion of the dataset is affected. Figure 3.2.10(a) and (b) show the impact of the noise on the accuracy of the model using both training and test sets respectively. As you can see, when the strength and size of the noise increase, the accuracy slightly decreases. From this result, we can see that the model is still stable even after adding high strength noise to 20% of the training set.

In the second experiment on model's stability, we used *univariate feature selection* to measure the impact of features. Univariate feature selection works by selecting the best features based on univariate statistical tests. Among the univariate feature selection

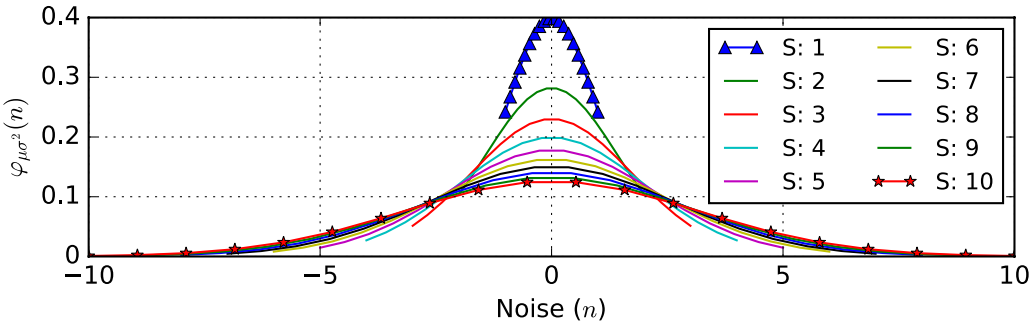


Fig. 3.2.9.: Applied noise to the training set for evaluating the stability of the model under different strengths (S) (strength=$1, \cdots, 10$)

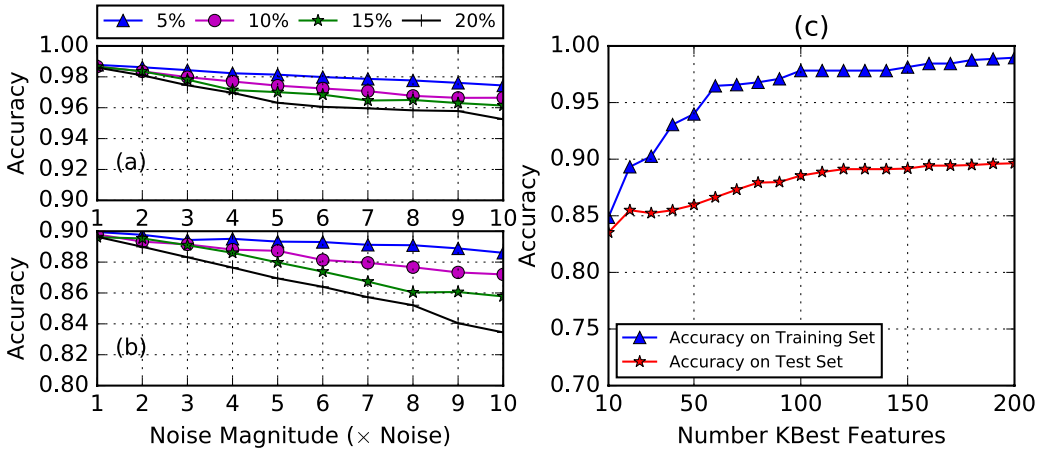


Fig. 3.2.10.: Evaluation of noise and feature impacts: (a) accuracy of the model under different settings of noised data and noise strengths; (b) evaluating model's accuracy by training the model with different sets of the features using K-Best features ($K = 1, \cdots, 10$)

methods, we used *KBest* feature selection and evaluated the accuracy of the model for different sets of features on both training and test sets. Figure 3.2.10(c) shows the results of this experiment. From those results, we understand that a combination of features including high and low score features is necessary to keep the accuracy high. The reason is that, if the top K high score features do not represent the differences of classes, the result will be a biased and overfit trained model. As you can see the accuracy of the model increases by adding more features and at some point (starting from 100) it stays stable. Table 3.2.3 shows the top 10% best features together with the average requests by malicious and benign apps.

Table 3.2.3.: Resource average usage statistics (top $K = 10$ best features)

Feature	WiFi	Ads	MMS	SMS	Blue.	Brow.	Root	Boot	Zygot	Call
Malicious	29.9	42	12.3	36.4	17.2	12.8	14.3	2.4	7.6	4.8
Benign	12	8.1	1.7	5.2	5.4	6.3	3.2	0.1	0.9	0.87

3.2.4.5 Active Learning Evaluation

In this section, we evaluate the active learning component of the model. We use three metrics to evaluate the performance of the active learning model. They are: 1) the speed of retraining the model to detect new instances; 2) the impact from the Oracle's expertise on the model's accuracy for the future apps; 3) the impact of the batch size when retraining the model using batch learning.

In the first experiment, we designed a scenario in which the model is initially trained by all the benign apps with different number of malicious apps. The rest of the malicious apps will be evaluated by the trained model and then verified by Oracle before they are added to the training set. Our goal is to assess how the active learning can help retrain the model to achieve higher accuracy in real time. In this experiment, since we are using labeled data, we can emulate an Oracle with expertise 1 who can accurately label all new apps. Figure 3.2.11 shows the evaluation results for this experiment. As you can see, the model is able to retrain the model by adding new labeled apps and improves the accuracy of the model. We can also see that when the initial model is trained with more malicious apps, the initial accuracy is higher. For example in Figure 3.2.11(d), when the number of initial malicious apps is 100, the accuracy increases quickly. We can also see that the detection accuracy fluctuates at the beginning in almost all cases. The reason is that the model is not sufficiently trained at the beginning and makes wrong predictions until more training data come in to improve the detection accuracy.

In the second experiment, we set the expertise of Oracle to different levels (0.6, 0.7, 0.8, and 0.9). In this scenario the initial model is trained with 50 malicious apps and all

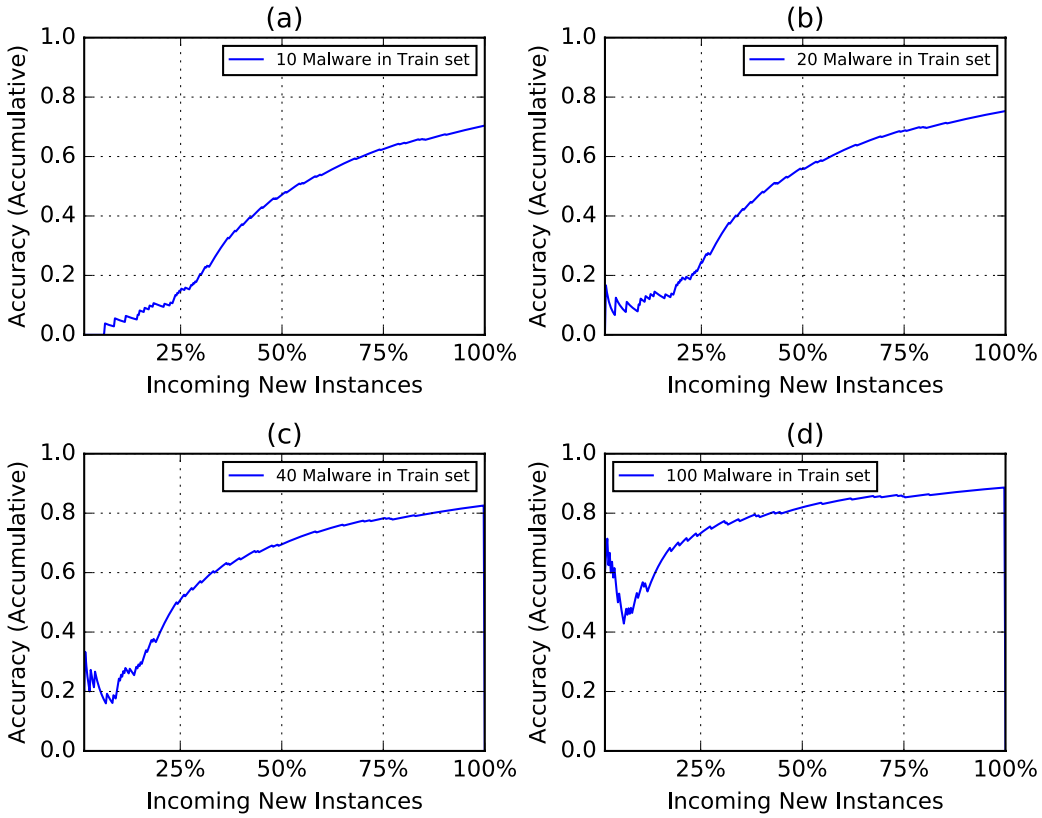


Fig. 3.2.11.: Accumulative accuracy of the model using Active Learning for incoming new instances with different trained models and Oracle's expertise "1": (a) accuracy of model trained with 10 malware; (b) accuracy of model trained with 20 malware; (c) accuracy of model trained with 40 malware; (d) accuracy of model trained with 100 malware.

the benign apps. In this experiment we aim to see the impact of Oracle's expertise on the model's accuracy. Figure 3.2.12(a) shows the accumulative accuracy of the model for the new incoming malicious apps. We can see that with lower expertise, the accumulative accuracy increases slower than that with higher Oracle's expertise.

In the last experiment, we evaluate the accuracy of the model when our query strategy is based on batch learning. For this experiment, we fixed the Oracle's expertise to be 1 and 50 apps are used for the initial model. We also set the size of batch to be 10,30,60, and 90. Figure 3.2.12(b) shows the experiment results. We can see that the accumulative accuracy has a direct relation with the batch size. With smaller batch size, the model gets retrained more frequently than when using large batch sizes. Compared with the previous two ex-

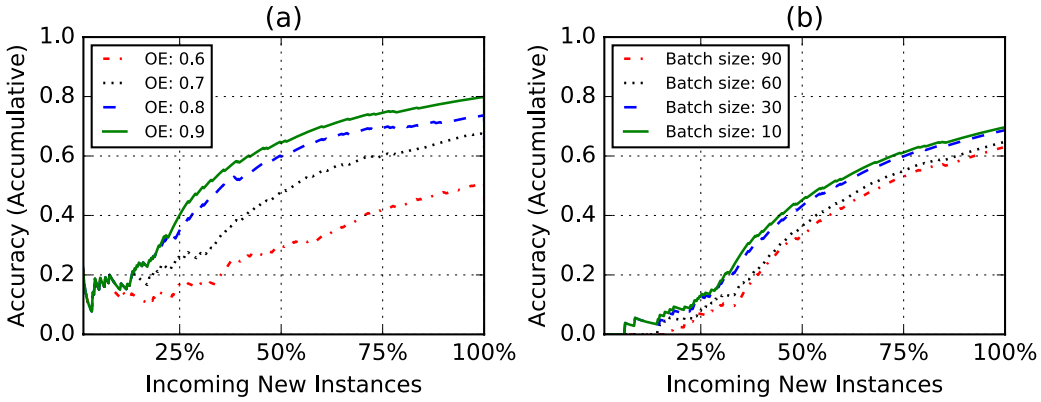


Fig. 3.2.12.: Accumulative accuracy of the model using Active Learning for incoming new instances: under different settings of Oracle's expertise (OE) and batch sizes: (a) accuracy of model with different Oracle's expertise (0.6, 0.7, 0.8, and 0.9); (b) accuracy of the model with different batch sizes (10, 30, 60, and 90).

periments with online learning (batch size=1), the accumulative accuracy is proportionally lower even though it has higher computational complexity. Therefore, choosing a proper batch size is critical for active learning.

3.2.5 Conclusion and Future Work

In this project, we propose an Android malicious app detection framework using SVM and active learning. We first define and select features to represent behaviours of Android apps and collect them through our own developed human-oriented instrumentation tool DroidCat. A filtering and parsing method is then employed to synthesize and organize the captured behaviours. We train the model with a proper malicious app dataset using the RBF kernel and test it with test datasets. Through our model, we can detect malicious apps with high accuracy. On top of the SVM model we implemented an active learning method to improve the stability and robustness of the detection model against new instances of Android malware. Our experimental results demonstrate that our proposed model achieves satisfying accuracy in terms of true positive and false positive rate and adapts the detection model for new malware trends.

Optimal Query Strategy: In our model, the type of query strategy that we use has a high impact on the model's performance and accuracy. In this model, we used the expected

reduce error strategy and achieved a reasonable performance. In order to improve the model, we can apply other query strategies to see if they can have better performance in terms of cost-effectiveness and accuracy. For example, some of the existing strategies measure the quality of the new data points and if they pass a quality threshold they can be included into the training set [73].

New Classes of Malware: One of the main important threats to any malware detection system is new trends (classes) of malware with very different behavioural patterns. In such a scenario, the model should be able to detect and retrain itself. One solution to this is using *progressive learning*. In progressive learning, the model is able to detect new classes and add them to the training set. Therefore, the training set will have more labels. This method can be implemented on top of the active learning method. Such a method for learning and training the model can be used if the model can be trained with a dataset with multiple (more than 2) labels.

CHAPTER 4

BOT USER DETECTION

In this project, we study the potential threats to the permission control recommendation system and propose corresponding solutions to defend against them. Since the system heavily relies on the responses from expert users, a number of dishonest users who have already received high expertise rating may misguide the system into providing wrong recommendations if not detected and handled properly. The research of this part is to answer this questions: *How to detect those dishonest users and mitigate their impact on the system?*. We propose two game-theoretical and clustering-based models to protect the system agains malicious users.

4.1 A Game-Theoretic Model for Defending Against Malicious Users in the Recommendation System

In this section, we propose a game-theoretical model to protect the recommendation component of the system against malicious users. Malicious users can mislead the recommendation component by giving false responses to permission requests. The proposed game-theoretical model (static Bayesian game) aims to predict malicious users' behaviors and take proposer actions against their attacks.

4.1.1 Problem Definition

RecDroid is a smartphone permission response recommendation system which utilizes the responses from expert users in the network to help inexperienced users. However, in such system, malicious users can mislead the recommendation system by providing un-

truthful responses. Although detection system can be deployed to detect the malicious users, and exclude them from recommendation system, there are still undetected malicious users that may cause damage to RecDroid. Therefore, relying on environment knowledge to detect the malicious users is not sufficient. In this work, we present a game-theoretic model to analyze the interaction (request/response) between RecDroid users and RecDroid system using a static Bayesian game formulation. In the game RecDroid system chooses the best response strategy to minimize its loss from malicious users. We analyze the game model and explain the Nash equilibrium in a static scenario under different conditions. Through the static game model we discuss the strategy that RecDroid can adopt to disincentives attackers in the system, so that attackers are discouraged to perform malicious users attack. Finally, we discuss several game parameters and their impact on players' outcome.

4.1.2 Background

In this section, we study the attack scenario on the recommendation system.

4.1.2.1 Attack RecDroid Recommendation System

RecDroid makes recommendations on permission requests based on the responses from expert users. In order to successfully attack the recommendation system with minimum cost, the attacker needs to do the following:

- The attacker should create multiple malicious expert users. Note that since only one user is recognized per phone by RecDroid, creating each new users implies obtaining new phones, which costs certain amount of money for the attacker.

- Each phone has to be managed to install sufficient number of apps and answer permission requests from those apps correctly in order to obtain expert ranking from RecDroid. Since RecDroid only takes responses from expert users for recommenda-

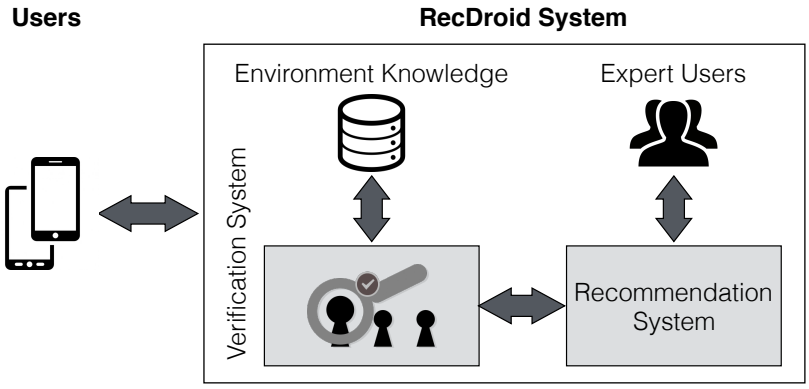


Fig. 4.1.1.: RecDroid system environment and detection system

tion, malicious users who does not reach expert ranking will not influent the recommendation system of RecDroid.

- When all malicious users are trained to be experts, all users start to install the malicious app (that the attacker possibly owns) and answer the permission requests dishonestly in order to mislead RecDroid.

- The attack should be performed as soon as the malicious app is released to public to minimize its cost. This is because when the app are already responded by many other normal expert users, the influence from a few malicious expert users will be reduced due to the voting mechanism of RecDroid to make decisions. The best time to influence RecDroid is at the time which apps are released.

4.1.2.2 Malicious User Detection

RecDroid has a malicious user detection process to assist its recommendation system. During the detection process, RecDroid detects the users' type (malicious or regular) based on the users' action in responding to permission requests. There are two distinct approaches for malicious user detection: one is the machine-learning (ML) approach and the other is the human verification approach. The ML approach utilizes the behavior similarity among

malicious users and it labels a user to be malicious if it is adequately similar to a known malicious user in terms of behavior. For example, they were created at about the same time and installed similar set of apps. Compared to the human verification, it costs much less and is much more efficient. However, it can be easily evaded if the malicious users are sufficiently well designed so they do not share much commonality. The human-based verification approach dedicates a human labor to verify the responses to the app requests and can discover malicious users. This approach can have much lower false positive rate and false negative rate compared to the ML-based approach. However, the cost of human verification has much higher cost so it shall not be used to verify all applications.

In our context, we assume RecDroid uses a human-based approach to verify malicious users. It is a practical approach since it is reliable and is much easier to implement. Figure 4.1.1 illustrates the relation between the users (malicious or regular) and the RecDroid system. As we can see, the recommendation system takes input from expert users (normal or malicious) to generate recommendation to new users regarding whether to grant access to permission requests or not. For an application that is chosen for verification, the recommendation will go through a verification process first to check whether it matches the correct answer from a human expert (verifier). If it does not match, then there might be a malicious user attack and the verifier can easily find out malicious expert users. Therefore, with the verification a malicious user attack can be easily discovered and the attack easily fails. If an app is not chosen to be verified, then the recommendation will be sent to new users directly. If there is a malicious user attack then the attack succeeds. Note that although human verification is reliable, there are still false positive and false negative cases since human makes mistakes sometimes.

4.1.2.3 Bayesian Games

In many situations the model may begin with some player having some information about something relevant to her decision making. These are called games of *incomplete information*, or *Bayesian games*. (Incomplete information is not to be confused with *imperfect* information in which players do not perfectly observe the actions of other players). Although any given player does not know the private information of an opponent (i.e. payoffs), she will have some beliefs about what the opponent knows, and we will assume that these beliefs are common knowledge [77].

A Bayesian game can be modeled by introducing Nature as a player in a game. Nature assigns a random variable to each player which could take values of *types* for each player and associating probabilities or a probability density function with those types (in the course of the game, *nature* randomly chooses a type for each player according to the probability distribution across each player's type space) [77].

Regarding the features of the RecDroid and its users, modeling the recommendation system through designing a Bayesian game model is a feasible and appropriate way. The idea behind the Bayesian game model is that generally an attacker/defender game is an incomplete information game where the defender is uncertain about the type of its opponent (regular or malicious). A Bayesian game model provides a framework for the defender to select its strategies based on its belief about the type of its opponent. As we mentioned, the most important characteristic of the system is that players are aware of the type of others and that is why in this type of games we have an external node (nature) to distinguish them from each other [33]. In the next section we describe the game model and how we design it.

4.1.3 Game Theoretic Model

Here are two types of users in RecDroid (malicious users and regular users). Detecting malicious users is a critical task for RecDroid. Since one attacker can create multiple malicious users, so what the RecDroid really plays with is either an attacker or a normal user. To study the interaction between the RecDroid system and users, we use a two-player static Bayesian game to model the behaviors of both parties. The static Bayesian game has two players: RecDroid users and the RecDroid system. The RecDroid users have private information about their types and the types are unknown to the RecDroid system. However, the type of RecDroid system is a common knowledge to all players (system and users).

An attacker player has two strategies: *Attack* and *Not attack* when sending responses to permission requests from an app. The regular user has only one strategy: *Not attack*. Although regular users may make mistakes sometimes (e.g. click some wrong button when providing a feedback) and behave similarly to an attacker, but this issue does not have a high impact to be included in strategies and we assume that regular users always provide correct responses [79]. When attack strategy is used, the attacker manipulate all malicious users to respond dishonestly to permission requests from an app. For example, the malicious expert users accept all malicious resource requests from an app, in order to mislead RecDroid into wrong recommendations. Correspondingly, the RecDroid system has two strategies: *Verify* and *Not verify* the correct responses for the app. When a verify strategy is used, RecDroid system allocates a seed expert to verify the responses to the requests from an application. This way the malicious attack fails.

We use ω_1, ω_2 to denote security value, which are the gain/loss for both the attacker and RecDroid. For example, an attacker choose to attack RecDroid, it needs to create multiple malicious users and train them into expert users (with cost), and then all attackers accept malicious permission requests from an application. If RecDroid does not detect

Table 4.1.1.: payoff matrices (RecDroid, Users)

(a) Player *i* is malicious

	Verify	**Not verify**
Attack	$(1-2\alpha)\omega_1 - c_a, (2\alpha-1)\omega_2 - c_v$	$\omega_1 - c_a, -\omega_2$
Not attack	$0, -\beta\omega_2 - c_v$	$0, 0$

(b) Player *i* is regular

	Verify	**Not verify**
Not attack	$0, -\beta\omega_2 - c_v$	$0, 0$

this behavior, RecDroid will make incorrect recommendation and losses ω_2 amount of its security value, which can be the loss of reputation by making wrong recommendations to users. For an attacker ω_1 means the gain of a successful attack by tricking inexperienced users into accepting malicious permission requests of a malicious application.

4.1.3.1 Normal Form

In this subsection we present the game in a static normal form. Table 4.1.1 shows the payoff matrices of the Two-player game in normal form game style. In the payoff matrices, α and β indicate the detection rate (true positive) and false alarm rate (false positive) of RecDroid by using human verification, and $\alpha, \beta \in [0, 1]$. ω_1, ω_2 are the security value as we previously mentioned. The cost of attacking and verifying are denoted by c_a and c_v, where c_a, $c_v > 0$. For example, the attacker needs to spend c_a amount of money to purchase a number of smart phones and spend time to set them up into malicious mode to be able to influent the RecDroid system. The RedDroid needs to pay a seed expert c_v amount of money to verify the correct responses to the permission requests from an application, in order to detect attacker's attacks. We assume that ω_1, ω_2 are greater than c_a, c_v, since otherwise attackers and RecDroid system do not have incentive for attacking the system and verifying the users' responses, respectively.

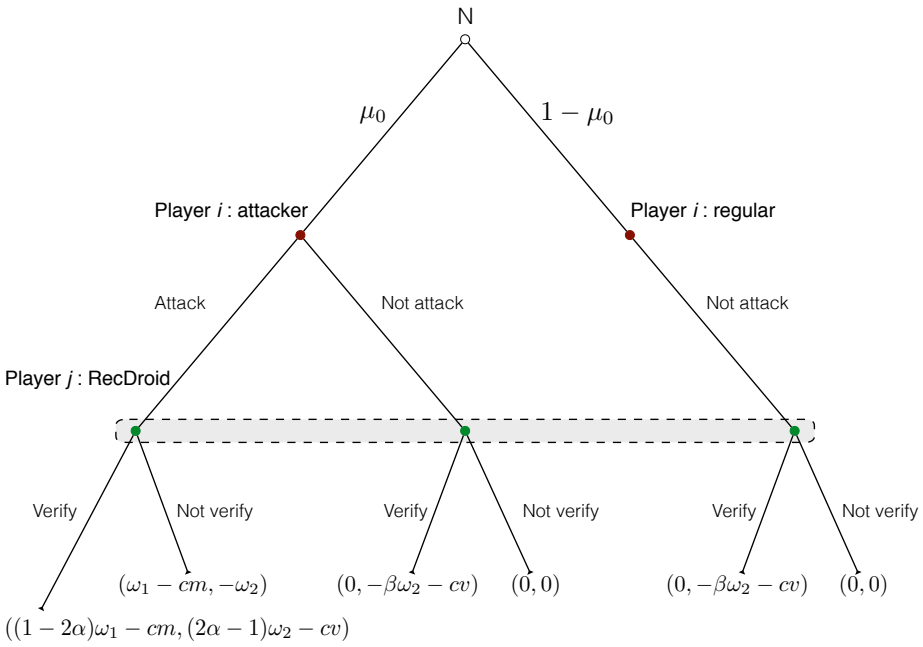


Fig. 4.1.2.: Extensive form of the Bayesian game

Table 4.1.1(a) describes the payoff matrices of the RecDroid system and attackers. We can see that the expected gain for the RecDroid in (*Attack*, *Verify*) strategy combination is $\alpha\omega_2 - (1-\alpha)\omega_2 = (2\alpha-1)\omega_2$, in which $(1-\alpha)$ is the false negative rate and a same for the attacker's payoff. This can be explained as: if verification successfully detects attack, then RecDroid gains reputation, otherwise RecDroid loses reputation. With the combination of *Not attack* and *Verify* or *Not verify* strategies, the payoff for the attackers is always 0 and the RecDroid's payoff is depends on the false positive rate.

Table 4.1.1(b) shows the payoff matrices for the RecDroid system and regular users. The payoff value for the user is 0 for all the RecDroid's strategies, and the RecDroid's payoff when it plays *verify* for the *Not attack* responses is $-\beta\omega_2 - c_v$, which is based on the false positive rate of the detection system. Although we mainly focus on malicious users in the model, but we also consider the regular users and their actions in the calculations.

4.1.3.2 Extensive Form

In this subsection we show another presentation of the game. Figure 4.1.2 shows the extensive form of the game. In this form we can see the possible moves of the players and their choices on each decision state. In the figure, node N indicates the nature, and μ_0 represents the probability that RecDroid is playing with an attacker, who will create malicious smartphone users to cheat in the system. For regular users, the only strategy is to respond to the requests honestly. RecDroid decides whether to verify the application permission requests or not depending on the μ_0 value as well as other parameters. Each terminal (leaf) node of the game tree has a 2-tuple of payoffs (RecDroid/user), which implies there is a payoff for each player at the end of every possible play.

4.1.3.3 Bayesian Nash Equilibrium (BNE)

As the main objective for players in the game is to try to gain higher expected payoff. On one hand, attackers try to minimize the probability of being detected by RecDroid system, on the other hand, RecDroid tries to detect attackers with less cost. As described previously, the payoff of the strategies for both the players depends on different parameters. The parameters include the false positive and true positive of malicious user detection or nature node (N). It is worth noting that the parameter μ_0 determined by nature has a high impact on payoffs of the players.

In the rest of this section, we analyze the Bayesian Nash equilibrium of the game under different circumstances. The μ_0 represents the defined probabilities, by which nature node shows the prior knowledge of the system about the users.

- The first case is when player *i* decides to plays *Attack* if it is an attacker, *Not attack* when it is a regular user. In this case, if RecDroid decides to *Verify* the incoming

responses. the expected payoff of RecDroid is as follow:

$$E_{pj}(Verify) = \mu_0((2\alpha - 1)\omega_2 - c_v) - (1 - \mu_0)(\beta\omega_2 + c_v), \tag{4.1}$$

and when the RecDroid decides to play *Not verify* the expected value would be

$$E_{pj}(Not - verify) = -\mu_0\omega_2, \tag{4.2}$$

then if

$$\begin{aligned} E_{pj}(Verify) &\leq E_{pj}(Not - verify) \\ \Rightarrow \mu_0 &\geq \frac{(1+\beta)\omega_2 + c_v}{(2\alpha + \beta - 1)\omega_2}, \end{aligned} \tag{4.3}$$

the best strategy for RecDroid is to play *Verify*. In this case if the RecDroid plays *Verify*, then *Attack* strategy is not the best strategy for attackers and they change their strategy to Not attack. So, this strategy combination can not be a BNE. However, if

$$\mu_0 < \frac{(1+\beta)\omega_2 + c_v}{(2\alpha + \beta - 1)\omega_2},$$

the best strategy for the RecDroid is *Not verify*. Therefore, attacker strategy for attackers, *Not attack* for regular users and *Not verify* for RecDroid is a pure-strategy BNE.

- The other case is when an attacker plays the *Not attack* strategy and regular user plays *Not attack* strategy, regardless of μ_0, then RecDroid's dominant strategy is to play *Not verify*. If the RecDroid plays *Not verify*, then the best strategy for the attackers is to play *Attack*. Therefore, this strategy combination cannot be a BNE.

The above analysis shows that there is no pure-strategy BNE when $\mu_0 \geq \frac{(1+\beta)\omega_2 + c_v}{(2\alpha+\beta-1)\omega_2}$. However, we can find a mixed-strategy BNE for this case. We let *p* denote the probability that an attacker plays *Attack*. We let *q* denote the probability that RecDroid plays *Verify*. We have

$$\begin{aligned} E_{pj}(Verify) = p\mu_0((2\alpha-1)\omega_2 - c_v) \\ -(1-p)\mu_0(\beta\omega_2 + c_v) \\ -(1-\mu_0(\beta\omega_2 + c_v)), \end{aligned} \tag{4.4}$$

$$E_{pj}(Not-verify) = -p\mu_0\omega_2 \tag{4.5}$$

then by imposing

$$\begin{aligned} E_{pj}(Not-verify) = E_{pj}(Verify) \\ \Rightarrow p^* = \frac{\beta\omega_2 + c_v}{(2\alpha+\beta)\omega_2\mu_0}. \end{aligned} \tag{4.6}$$

In order to calculate mixed-strategy for users we impose

$$\begin{aligned} E_{pi}(Attack) = E_{pi}(Not-attack) \\ \Rightarrow q^* = \frac{\omega_1 - c_a}{2\alpha\omega_1}. \end{aligned} \tag{4.7}$$

Finally, after calculating the *p* and *q* probabilities, we have the mixed-strategy ((p^* if *Attack* attacker, *Not attack* if regular), q^*, μ_0) when $\mu_0 \geq \frac{(1+\beta)\omega_2 + c_v}{(2\alpha+\beta-1)\omega_2}$.

4.1.3.4 Practical Implication of BNEs

We can summarize the implication of the BNEs we have found above as follows:

- When the probability of attacker is small enough, aka. $\mu_0 < \frac{(1+\beta)\omega_2 + c_v}{(2\alpha+\beta-1)\omega_2}$, then there is no incentive for RecDroid to verify any application responses.

- When the probability of attacker is not small enough, aka. $\mu_0 \geq \frac{(1+\beta)\omega_2 + c_v}{(2\alpha+\beta-1)\omega_2}$, then the RecDroid should play verify sometimes in order to minimize its damage.

- However, in the second case, if RecDroid plays *verify* with probability $q^* = \frac{\omega_1 - c_a}{2\alpha\omega_1}$, there is no profit difference whether the attacker plays *attack* or *not attack*. Therefore, there is no incentive for attackers to perform attack under this case. Furthermore, if $q^* > \frac{\omega_1 - c_a}{2\alpha\omega_1}$, then it is better off for attackers to play *not attack*.

From the above results, we can see the impact from μ_0 to the solutions of the game model. In the discussion section we will further discuss the impact of other parameters in different cases.

4.1.4 Discussion

As we described in the previous sections, we have some parameters in the proposed model and they all have some impact on the outcome of the players in a Bayesian game. In this section we discuss two important parameters α and μ_0, which are essential in our formulations. When the RecDroid's belief of μ_0 is high, which means the probability that the RecDroid system playing with an attacker is high, then RecDroid should have a high probability to play *verify* strategy in order to get optimal payoff. If the parameter α is high and $beta$ is low, which means the human verifier is more reliable, then the probability of RecDroid playing *verify* is low.

From Equation (7) we can see that the border-line *verify* probability q^* is influenced by parameters $\omega_1, c_a,$ and α. Higher ω_1, lower c_a, and lower α imposes higher probability of verity strategy. It is because (1) the cost of being attacked is higher, the RecDroid system should be more cautious and play more "verify strategy". (2) the lower that cost of attack, the attackers will be more likely to attack and therefore, RecDroid should increase probability of verification. (3) the lower α is, the less reliable the human experts, then

RecDroid should increase the probability of verification.

4.1.5 Conclusion

In this work, we presented a game-theoretic model for the RecDroid recommendation system to analyze the interaction between RecDroid and the users (attacker, normal). In the system, we try to maximize the security of the system when it does not have enough information about the users' type. We defined the strategy space for both the players (RecDroid system, users) in a static scenario. We also discussed the parameters that influence the final outcome of the players. In the proposed static game, the RecDroid always assume fixed prior probabilities about the types of his opponent throughout the entire game period. We show that the static game leads to a mixed-strategy BNE when the RecDroid's belief of player *i* (users) being malicious is high and to a pure-strategy BNE when the RecDroid's belief of player *i* being malicious is low.

We proposed a game-theoretic model to optimize the RecDroid's strategies to minimize the damage from the attacks. RecDroid can use the proposed model to detect the malicious responses from normal responses of the users and eliminate them from recommendation system. Although it can increase the accuracy of the system, there is still a way to improve the model. The proposed model was a static Bayesian game and modeling the system in a dynamic way can improve the accuracy more. Using a dynamic model can help the RecDroid to use the behavioral history of the users and the game to improve its prior knowledge, α, and consequently the malicious user detection rate.

4.2 Extended Game-Theoretic Model

In this section, we present an extended version of the proposed game-theoretical model in Section 4.1. We extended the set of actions that RecDroid can take in case of an attack from malicious users side. Extending the actions set by including Machine Learning and

Human Verification actions can help RecDroid to defend the recommendation component against rational attackers.

4.2.1 Problem Definition

RecDroid is an Android smartphone permission control framework which provides fine-grained permission control regarding smartphone resources and recommends the permission control decisions from savvy users to inexperienced (novice) users. However, malicious users, such as dummy users created by malicious app owners, may attempt to provide untruthful responses in order to mislead the recommendation system. Although a sybil detection function can be used to detect and remove some dummy users, undetected dummy users may still be able to mislead RecDroid framework. Therefore, it is not sufficient to depend on sybil detection techniques. In this work, we investigate this problem from a game-theoretical perspective to analyze the interaction between users and RecDroid system using a static Bayesian game-theoretical formulation. In the game, both players choose the best response strategy to minimize their loss in the interactions. We analyze the game model and find both pure strategy Nash equilibrium and mixed strategy Nash equilibrium under different scenarios. Finally, we discuss the impact from several parameters of the designed game on the outcomes, and analyzed the strategy on how to disincentives attackers through corresponding game design.

4.2.2 Malicious User Detection

In order to reduce the influence from malicious users, RecDroid has a malicious user detection function. In the detection step, RecDroid detects the type of users (malicious or normal) based on the users' behavior in responding to resource requests. There are two distinct options for RecDroid to perform malicious user detection: one is the machine-learning-based (ML) approach and the other is the human-based approach. The ML ap-

proach uses the similarity among malicious users in terms of their behaviors, and labels malicious users if they adequately similar to known malicious users in terms of behavior. For example, they were created at around the same time, installed similar set of apps, and they have similar responses to app requests. Compared to the human verification, ML-based detection costs much less and is much more efficient. However, it may not be able to detect malicious users created by sophisticated attackers. For example, malicious users can be created without sharing much commonality. On the other hand, the human-based verification approach uses a human labor to manually verify the responses to the app requests and compare them with the responses from other users. This way malicious users can be discovered and the attack fails. The human-based approach can have much lower false-positive rate and false-negative rate compared to the ML-based approach. However, the cost of human verification approach can be much higher so it shall not be used verify all applications.

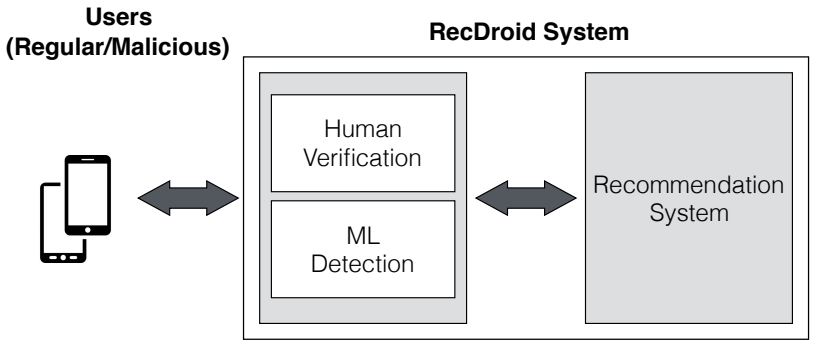


Fig. 4.2.1.: RecDroid system environment and detection system

Figure 4.2.1 illustrates the interaction between the users (malicious or normal) and the RecDroid framework. As show in the figure, the recommendation system takes responses from expert users (normal or malicious) and make recommendation to new users regarding whether to grant access to requests or not. An application that is chosen for verification, the recommendation is used to detect if there is a malicious users attack. If a ML-based detection method is used, then it will compare the similarity among users of the app and raise

alarm if suspicious malicious users are found. After that the responses from suspicious malicious users are then removed from RecDroid system. If a human-based approach is found, the ground truth of responses is revealed and dishonest malicious users are discovered. However, if an application is not chosen to be verified, then the recommendation will be sent to new users directly. Therefore, without malicious user detection, a malicious user attack may succeed. However, malicious user detection methods may bring false-positive and false-negative. They also bring extra cost to RecDroid.

4.2.3 Extended Game-Theoretical Model

There are two types of users in RecDroid, malicious and regular users. Detecting malicious users is a critical tasks for RecDroid. Since one attacker can create multiple malicious users, so what the RecDroid really plays with is either an attacker or a normal user. To study the interaction between the RecDroid system and users, we use a two-player static Bayesian game to model the behaviors of both parties. The static Bayesian game has two players: RecDroid users and the RecDroid framework. The RecDroid users have private information about their types and the types are unknown to the RecDroid system. However, the type of RecDroid framework is a common knowledge to all players (system and users).

Previously, we proposed a game-theoretical model [62], in which RecDroid has only two actions *Verify* and *Not Verify*. In the previous work, the verification system was only based on collected environmental information from apps and their developers and static analysis results. Since malicious users are in different levels of maliciousness, relying on these information as criteria to detect all types of malicious users (compolex and simple behaviour) was not effective. We improved the model in a way that verification system is based on *Machine Learning* and *Human Verification* approaches. An attacker player has two strategies: *Attack* and *Not attack* when sending responses to permission requests from

an app. The regular user has only one strategy: *Not attack*. When attack strategy is used, the attacker manipulates all malicious users to respond dishonestly to permission requests from an app. For example, the malicious expert users accept all malicious resource requests from an app, in order to mislead RecDroid into wrong recommendations.

Correspondingly, the RecDroid system has three strategies: *Human Verification*, *ML Verification*, and *No Verification*. When the no verification strategy is used, RecDroid makes recommendation based on all experts' responses without caring whether those responses are from malicious users or not; when ML verification is selected, RecDroid uses a machine learning approach to detect suspicious malicious users who are controlled by the same attacker, and those responses from suspicious users will not be considered by RecDroid recommendation; when human verification is chosen, a human expert will manually respond to permission requests from an app. ML verification is based on collected responses from users. In our ML verification system we consider users' responses as *response history* in order to assess the risk of considering responses in our recommendations. If assessed risk is high the response will be ignored with high probability and accept if the risk is low. This way malicious users' responses are not considered and the attack fails. Although human verification is the most accurate verification strategy in our system, it requires extra cost to the defender side due to human labor.

In order to track the payoffs of players in the game, we use ω_1 to denote the security value of the attacker, which is the gain of the attacker by performing a successful attack. For example, after a successful attack to RecDroid, a number of extra users accept malicious requests and it brings ω_1 extra profit through the attack. ω_2 is used to denote security value of RecDroid, which is the gain of a successful recommendation or the loss if compromised. For example, if RecDroid does not detect this behavior, RecDroid will make incorrect recommendations regarding the app and losses ω_2 of its security value, which can be the loss of reputation by making wrong recommendations to users. If RecDroid successfully

detects attacks and removes malicious users, it gains reputation and we assume the gain is also ω_2.

4.2.3.1 Normal Form

In this subsection we present the game in a static normal form. Table 4.2.1 depicts the matrices of payoffs of the Two-player game in normal form game style. In the payoff matrices, α_m and β_m indicate the detection rate (true-positive) and false alarm rate (false-positive) of RecDroid by using machine-learning detection, and $\alpha_m, \beta_m \in [0, 1]$. We assume the cost of using machine detection is negligible (i.e., $c_m = 0$). We also assume human experts are reliable so that their decisions are consistently correct (i.e., $\alpha_h = 1, \beta_h = 0$). ω_1, ω_2 are the security value as we previously mentioned. The cost of attacking and human verification are denoted by c_a and c_h, where c_a, $c_h > 0$. For example, the attacker needs to spend c_a amount of money to purchase a number of smart phones and spend time to set them up into malicious mode to be able to influent the RecDroid system. The RedDroid needs to pay a seed expert c_h amount of money to verify the correct responses to the permission requests from an application, in order to detect attacks. We assume that ω_1, ω_2 are greater than c_a, c_h, since otherwise the attacker does not have incentive to attack and the RecDroid system has no incentive to use human verification, respectively.

Table 4.2.1(a) describes the payoff matrices of the RecDroid system and attackers. We can see that the expected gain for the RecDroid in (*Attack*, *ML Detection*) strategy combination is $-(1 - \alpha_m)\omega_2 = (\alpha_m - 1)\omega_2$, in which $(1 - \alpha_m)$ is the false-negative rate and a same for the attacker's payoff. This can be explained as: if the ML detector does not detect attack, then RecDroid generates incorrect recommendation and loses reputation. When the attacker chooses the *Not attack* strategy, the payoff for the attackers is always 0 and the RecDroid's payoff is depends on the false-positive rate and detection cost, in the *ML Detection* and *Human Verification* cases, respectively.

Table 4.2.1.: payoff matrices (RecDroid, Users)

(a) Player *i* is malicious

	ML Detection	**Human Verification**	**No Detection**
Attack	$(1-\alpha_m)\omega_1 - c_a,$ $(\alpha_m - 1)\omega_2 - c_m$	$(1-\alpha_h)\omega_1 - c_a,$ $(\alpha_h - 1)\omega_2 - c_h$	$\omega_1 - c_a, -\omega_2$
Not attack	$0, -\beta_m\omega_2 - c_m$	$0, -\beta_h\omega_2 - c_h$	$0, 0$

(b) Player *i* is regular

	ML Detection	**Human Verification**	**No Detection**
Not attack	$0, -\beta_m\omega_2 - c_m$	$0, -\beta_h\omega_2 - c_h$	$0, 0$

Table 4.2.1(b) shows the payoff matrices for the RecDroid system and regular users. The payoff value for the regular user is 0 for all RecDroid's strategies. The RecDroid's payoff depends on the strategies it plays. For example, when it plays *ML Detection* the payoff is $-\beta_m\omega_2$, which means the cost that RecDroid system lose the chance to make correct recommendation by falsely label some users to be malicious. Although we mainly focus on malicious users in the model, but we also consider the regular users and their actions in the calculations.

4.2.3.2 Extensive Form

In this section we show a different presentation form of the proposed game. Figure 4.2.2 shows the extensive form of the game. In this form we can see all the possible moves of the players and their choices on each decision state. In the figure, the root N indicates the nature node, and μ_0 represents the probability that RecDroid is playing with an attacker over an app. The attacker may create multiple malicious smartphone users to attack the system. For regular users, the only strategy is to respond to the incoming requests honestly. RecDroid decides whether to verify the app's resource request or not depending on the μ_0 value as well as the other parameters of the game. Each terminal (leaf) node of

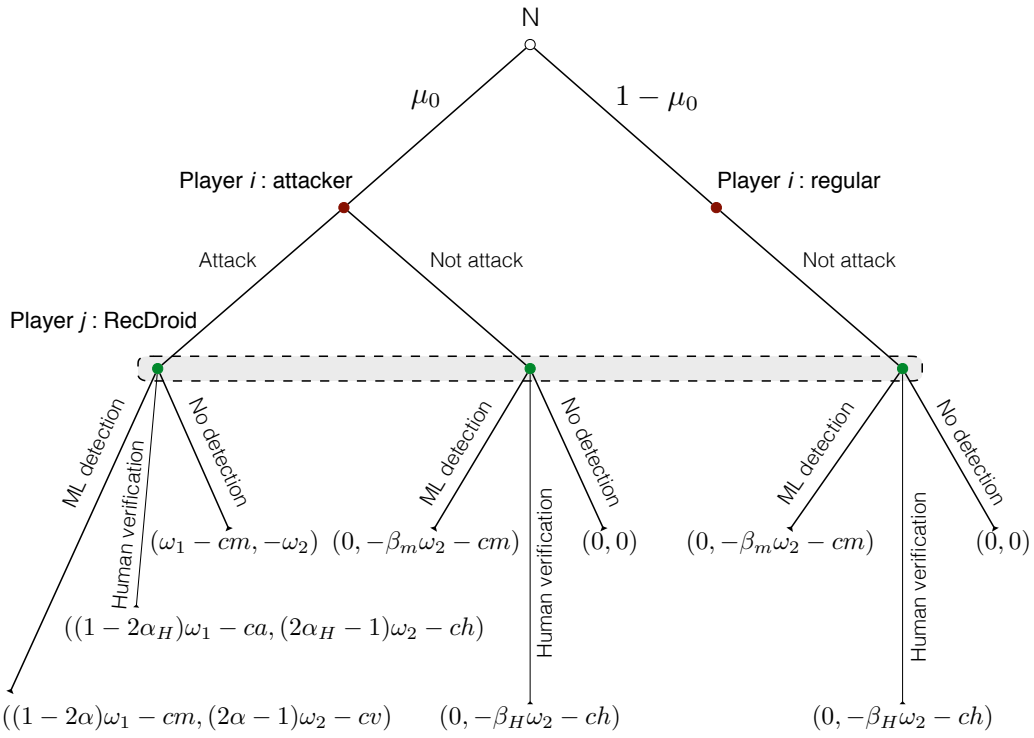


Fig. 4.2.2.: Extensive form of the Bayesian game

the game tree has a 2-tuple of payoffs (RecDroid/user), which implies there is a payoff for each player at the end of every possible play.

4.2.3.3 Bayesian Nash Equilibrium (BNE)

As the main objective for all players in the game is to try to gain higher expected payoff. On one hand, attackers try to minimize the probability of being detected by RecDroid system, on the other hand, RecDroid tries to detect attackers with less detection cost. As described previously, the payoff of the strategies for both the players depends on different parameters. The parameters include the false-positive and true-positive of malicious user detection or nature (N). It is worth noting that the parameter μ_0 determined by nature has a high impact on payoffs of the players. In order to simplify the analysis, we group the strategy *ML Detection* and *Human Verification* into the strategy *Detection* and drop their subscripts $_{h,m}$ in presentation. We will compare the two detection strategies when the

Detection strategy is selected.

We first analyse the Bayesian Nash equilibrium of the game under different possible circumstances. The μ_0 denotes the defined probabilities, by which nature node shows the prior knowledge of the system about the users.

- The first possible case is when player *i* decides to play *Attack* if it is an attacker, *Not attack* when it is a normal user. In this case, if RecDroid decides to play *Detection* for the incoming responses from users. The RecDroid's expected payoff is as follow:

$$E_{pj}(Detection) = \mu_0((\alpha - 1)\omega_2 - c) - (1 - \mu_0)(\beta\omega_2 + c), \tag{4.8}$$

and when the RecDroid decides to play *No Detection* the expected value would be

$$E_{pj}(NoDetection) = -\mu_0\omega_2, \tag{4.9}$$

then if

$$\begin{aligned} E_{pj}(Detection) &> E_{pj}(NoDetection) \\ \Rightarrow \mu_0 &\geq \frac{\beta\omega_2 + c}{(\alpha + \beta)\omega_2}, \end{aligned} \tag{4.10}$$

the best possible strategy for RecDroid is to play *Detection*. In this case if the RecDroid plays *Detection*, then the attacker only stays in strategy*Attack* if and only if $(1 - \alpha)\omega_1 - c_a > 0 \Rightarrow \alpha < 1 - \frac{c_a}{\omega_1}$, which forms a Bayesian Nash Equilibrium. Therefore, under this condition, the *Attack* strategy for attackers, the *Not attack* for regular users and *Detection* for RecDroid is one pure-strategy BNE.

In another case, if

$$\mu_0 < \frac{\beta\omega_2 + c}{(\alpha + \beta)\omega_2},$$

the best strategy for the RecDroid is *No Detection*. Therefore, under this condition, the *Attack* strategy for attackers, the *Not attack* for regular users and *No Detection* for RecDroid is another pure-strategy BNE.

- The other case is when an attacker plays the *Not attack* and regular user plays *Not attack* strategy, regardless of μ_0, then RecDroid's dominant strategy is to play *Not verify*. If the RecDroid plays *No Detection*, then the best strategy for the attackers is to play *Attack*. Therefore, this strategy combination cannot be a BNE.

The above analysis shows that there is no pure-strategy BNE when $\mu_0 \geq \frac{\beta\omega_2 + c}{(\alpha + \beta)\omega_2}$ and $\alpha \geq 1 - \frac{c_a}{\omega_1}$. However, we can find a mixed-strategy BNE for this case. We let *p* denote the probability that an attacker plays *Attack*. We let *q* denote the probability that RecDroid plays *Detection*. We have

$$\begin{aligned} E_{pj}(Detection) = p\mu_0((\alpha - 1)\omega_2 - c) \\ -(1-p)\mu_0(\beta\omega_2 + c) \\ -(1-\mu_0)(\beta\omega_2 + c), \end{aligned} \tag{4.11}$$

$$E_{pj}(No - Detection) = -p\mu_0\omega_2 \tag{4.12}$$

then by imposing

$$\begin{aligned} E_{pj}(No - Detection) = E_{pj}(Detection) \\ \Rightarrow p^* = \frac{\beta\omega_2 + c}{(\alpha + \beta)\omega_2\mu_0}. \end{aligned} \tag{4.13}$$

In order to calculate mixed-strategy for users we impose

$$E_{pi}(Attack) = E_{pi}(Not - attack) \Rightarrow q^* = \frac{\omega_1 - c_a}{\alpha\omega_1}. \tag{4.14}$$

In summary, we have the mixed-strategy (($P[Attack] = p^*$ for attackers, *Not attack* for regular), $P[Detection] = q^*$ for RecDroid) under the condition that $\mu_0 \geq \frac{\beta\omega_2 + c}{(\alpha + \beta)\omega_2}$ and $\alpha \geq 1 - \frac{c_a}{\omega_1}$.

Correspondingly, we have the payoff for both players at mixed-strategy BNE are

$$\begin{aligned} E_{pi}(Attack) = E_{pi}(Not - attack) = 0, \\ E_{pj}(Detection) = E_{pj}(No - Detection) = \frac{\beta\omega_2 + c}{\alpha + \beta} \end{aligned} \tag{4.15}$$

4.2.3.4 Comparison Between the Two Detection Strategies

The above results shows the binary condition of the defender's strategies: *Detection* or *No Detection*. However, when the strategy *Detection* is chosen, there are actually two choices: *ML Detection* or *Human Verification*. RecDroid always chooses the strategy that brings higher payoff for the system. From Equation (4.15) we have, at the mixed strategy BNE, if $c_h > \frac{\beta_m\omega_2}{\alpha_m+\beta_m}$, then the human verification strategy is chosen at the mixed-strategy BNE, otherwise the ML detection strategy is chosen at the mixed BNE.

4.2.3.5 Incentive Compatibility of RecDroid

We can summarize the implication of the BNEs that we have found above as follows:

- When the probability of attacker is small enough, aka. $\mu_0 < \frac{\beta_m\omega_2 + c}{(\alpha_m + \beta_m)\omega_2} = \frac{\beta_m\omega_2}{(\alpha_m + \beta_m)\omega_2}$ and $\mu_0 < \frac{\beta_h\omega_2 + c_h}{(\alpha_h + \beta_h)\omega_2} = \frac{c_h}{\omega_2}$, then there is no incentive for RecDroid to use any detection process for users' responses.

- When the probability of attacker is not small enough, aka. $\mu_0 \geq \min(\frac{\beta_m \omega_2}{(\alpha_m + \beta_m)\omega_2}, \frac{c_h}{\omega_2})$, then the RecDroid should either use *ML detection* or *Human verification* to check the input users responses, whichever costs less.

- However, in the second case, if RecDroid plays *Detection* with probability $q^* = \frac{\omega_1 - c_a}{\alpha \omega_1}$, there is no profit difference whether the attacker plays *attack* or *not attack*. Therefore, there is no incentive for attackers to perform attack in this case. Furthermore, if $q^* > \frac{\omega_1 - c_a}{\alpha \omega_1}$, then it is better off for attackers to play *not attack*. This way, RecDroid system *disincentives* attackers from attacking the system.

- It is worth noting that the payoff in Equation (4.15) is the maximum payoff RecDroid can obtain given that the system provides disincentive for attackers to attack the system.

4.2.4 Discussion

As we described previously, we have a number parameters in our proposed model and they all have impact on the outcome of the players in this Bayesian game. In this section we discuss two important and main parameters α and μ_0, which are essential with a high impact in our formulations. When the RecDroid's belief of μ_0 is high (high probability), which means the probability that the RecDroid system playing with an attacker is high, then RecDroid should play *verify* strategy with a high probability in order to get optimal payoff. If the parameter α is high and $beta$ is low, which means the human verifier is more reliable, then the probability of RecDroid playing *verify* is low.

From Equation (7) we can see that the border-line *verify* probability q^* is influenced by parameters $\omega_1, c_a,$ and α. Higher ω_1, lower c_a, and lower α imposes higher higher probability of verity strategy. It is because (1) the cost of being attacked by system is higher, the RecDroid system should be more cautious and play more verify strategy.; (2)

the lower that cost of attack, the attackers will be more likely to attack and therefore, RecDroid should increase probability of verification; (3) the lower α is, the less reliable the human experts, then RecDroid should increase the probability of verification.

In addition, since verification system is based on human and machine learning approaches, it causes some challenges. First, using human verification as a tool to verify the incoming responses requires extra cost to the defender side due to human labor. Second, since applying machine learning approach needs enough collected information from users such as their responses history and environmental information, at the beginning of running the system we cannot provide the ML system with enough information for bootstrapping step of the system.

4.2.5 Conclusion

In order to analyse the interaction (permission request/user's response) between RecDroid recommendation framework and users (attacker/malicious, regular/normal), we proposed a static Bayesian game-theoretical model. Since RecDroid does not have enough knowledge about the users' type (maliciousness feature of users), we try to maximize and enhance the security of the framework's recommendations to users through training the system. As any two-player game needs to be provided with strategies spaces, we defined the possible strategy space for both the players (RecDroid system, users) based on a static game scenario. We also discussed the parameters of the proposed game that influence the final outcome of the players and challenges which are caused by Machine Learning approach and Human verification system as *verify* strategy of the RecDroid. In the proposed static Bayesian game, the RecDroid always assume fixed prior probabilities about the types of his opponent throughout the entire game period. Using Machine Learning and Human verification approaches can improve the accuracy of the prepared recommendations by RecDroid. We proved that the proposed static Bayesian game leads to a mixed-strategy BNE when

the RecDroid's belief of player *i* (users) being malicious is high (high probability) and to a pure-strategy BNE when the RecDroid's belief of player *i* being malicious is low.

Using game theory to optimize the RecDroid's strategies to minimize the damage from the attacks by malicious users (fraudulent users) is our main focus in this work. As we described before, using ML and Human verification approaches causes some serious challenges, but RecDroid can use the proposed model to detect the malicious responses from normal (non-malicious) responses of the users and disregard them from recommendation system. Although it can increase the accuracy of the system, there is still a way to improve the model. The proposed model was a static Bayesian game and modeling the system in a dynamic (updating the current players' beliefs during the time) way can improve the accuracy of recommendations more. Using a dynamic model can help the RecDroid to use the *activity history* of the users and the game to improve its prior knowledge, α, and consequently the malicious user detection rate. Although using a dynamic model brings more overhead such as more calculation, updating the prior knowledge can improve the accuracy significantly.

4.3 BotTracer: Bot User Detection Using Clustering Method in RecDroid

Following the presented game-theoretical models in Sections 4.1 and 4.2, in this section, we present a clustering-based model to protect the expert seeking component and detect bot users. The clustering model uses a set of well-defined features to detect bots at a very early stage.

4.3.1 Problem Definition

RecDroid is a smartphone permission management system which provides users with a fine-grained real-time app permission control and a recommendation system regarding whether to grant the permission or not based on expert users' responses in the net-

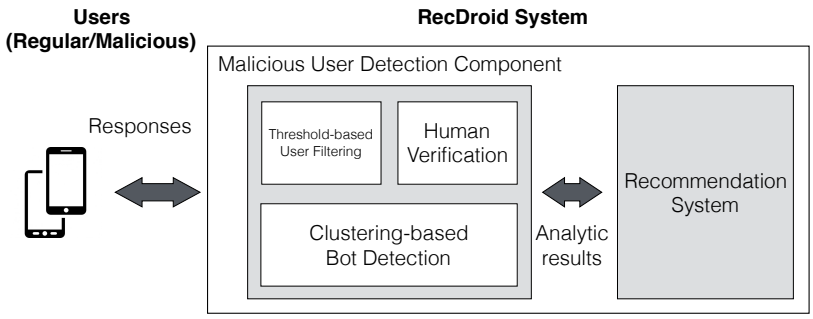


Fig. 4.3.1.: RecDroid system environment and detection system

work. However, in such a system, malware owners may create multiple bot users to misguide the recommendation system by providing untruthful responses on the malicious app. Threshold-based detection method can detect malicious users which are dishonest on many apps, but it cannot detect malicious users that target on some specific apps. In this work, we present a clustering-based method called BotTracer to finding groups of bot users controlled by the same masters, which can be used to detect bot users with high reputation scores. The key part of the proposed method is to map the users into a graph based on their similarity and apply a clustering algorithm to group users together. We evaluate our method using a set of simulated users' profiles, including malicious and regular ones. Our experimental results demonstrate high accuracy in terms of detecting malicious users. Finally, we discuss several clustering features and their impact on the clustering results.

4.3.2 Background

In this section we briefly describe the RecDroid recommendation system and potential strategies that an attacker can use to cheat in RecDroid system.

4.3.2.1 RecDroid Trust Computation and Malicious Users Filtering

In order to find expert users in the network, RecDroid uses Bayesian Inference model to compute the estimated trust level of users. The trust value of a user is based on the

cumulative number of correct responses and incorrect responses observed by RecDroid. The feedback from a user is only considered as expert responses if the trust value of the user is higher than a threshold. When the number of cumulative observations is sufficiently high, the trust value of a user is estimated using the following equation:

$$s \approx \frac{\alpha}{\alpha + \beta} \tag{4.16}$$

where α represents the cumulative number of correct responses the user has in the past; β is the cumulative number of incorrect responses.

RecDroid's recommendation system has a simple threshold-based user filtering system, which eliminates the impact from malicious users and inexperienced users. For instance, the responses from a user is only considered in the recommendation computation if its trust value is higher than a threshold θ_t. However, this method is not effective to detect *targeted bot attacks*, where bot users build up their expertise level by behaving honestly on all other apps except a few targeted apps. For example, an attacker may create a set of bot users and each bot installs a set of legit apps and respond to their request correctly. This way all bots will build high trust values after some time. However, those bots also install a malicious puzzle game app owned by the attacker and all accept the intrusive text messaging permission from the app in order to mislead the RecDroid recommendation. The simple trust-based filtering method is not effective to this type of attack. Figure 4.3.1 shows the overall architecture of RecDroid's malicious user detection.

RecDroid also uses human verification to detect malicious users [62, 61]. In this process, human verifiers randomly choose users and verify their responses manually. This way, malicious users may be discovered by human experts. However, the human verification has much higher cost so it is not practical to cover all apps.

In order to protect RecDroid from bot attacks on targeted apps, we need an automatic bot detection method which can discover bot users which are controlled by masters. We

propose BotTracer, a machine-learning based bot users detection mechanism. We first analyze the similarity features of bot users and then apply clustering method to group bot users together.

4.3.2.2 Attack RecDroid Recommendation System

The purpose of Recdroid is to provide low-risk recommendations on permission requests based on the responses from expert users. However, malicious users can attack the system by manipulating the RecDroid recommendation. For example, the owner of a malicious app may create many fake (bot) users and use them to mislead RecDroid recommendation system. A typical attacking strategy can be described as follows:

- The attacker can create multiple malicious expert users. Note that since RecDroid enforces one-user-per-phone policy [65, 66], which limits the number of bot users the attacker can create due to the high device cost.

- Each phone should install sufficient number of apps and respond permission requests from those apps correctly in order to obtain expert ranking from RecDroid. In order to boost the expertise levels of bot users, the attacker should prepare a pool of legit apps and corresponding correct responses to the requests from those apps. With that the bot users can build their trust value based on the provided correct responses. However, it takes time and effort to look for apps and find correct responses to the requests. Therefore, we can assume the app pool is relatively small.

- After all bot users are rated to be experts, all users start to install the targeted malicious app (that the attacker possibly owns) and answer the permission requests dishonestly in order to mislead RecDroid's recommendation system.

- The attack should be performed soon after the malicious app is released to public to minimize its cost. This is because when the app is already responded by many other

normal expert users, the influence from a few malicious expert users will be reduced due to the voting mechanism of RecDroid to make decisions. Therefore, the best time to influence RecDroid is at the beginning.

4.3.3 Model

In BotTracer, we try to involve applying clustering to detect malicious (fake) users. In order to defend against targeted bot attacks, we propose BotTracer, a clustering-based solution to detect bot users in RecDroid. There are two types of users in RecDroid system, malicious and regular users. We transform the malicious users detection problem into a clustering problem with two clusters, *malicious* and *regular*. To build the clustering method, we need to define related features of both malicious and regular users.

4.3.3.1 Malicious (bot) Users

We define bot users as fake users who are set up by attackers to serve the purpose of manipulating the RecDroid recommendation system through responding to permission requests dishonestly. A group of bot users can be controlled by the same master attacker. The attacker can set up a number of real devices equipped with RecDroid and the user on each device (one per device) installs a number of apps and responds to apps' permission requests. The master has control over the bot users to complete the objective, such as changing the permissions' recommendation of their targeted apps.

4.3.3.2 Feature Identification and Construction

There are three key types of information related to users' apps and their responses to requests: app overlaps (F1), time of response (F2) and response similarity (F3). We use a network graph to represent the users using RecDroid system and the distance between users is computed using a function of the three above features. The features are explained

in further detail as follows:

app overlaps: For a pair of users, the app overlap is defined to be the number of apps that the two users have in common (installed apps). Intuitively, the more app overlaps between two users, the more similarity between the users.

Time of response: the time period that the app is published in the market until a permission request is responded by a user after installation. Tracking the response time is important since the most effective time period that the attacker can influence the system is at the beginning of the app being published, such as the first 24 hours, when few regular users have installed and responded to the app. This way, malicious bot users can change the recommendations toward their objectives. Therefore, we predict that bot users are the first group of users that respond to the malicious apps' requests after the app is published.

Figure 4.3.2 illustrates the apps installation timeline. As we can see, users $(u_0 - u_3)$ respond to permission requests in the time period of Δ_{t1}, so that they have higher influence on recommendations than users respond in Δ_{t2}. Therefore, to manipulate the RecDroid response to a permission request effectively, bot users should respond to requests during the first time period. We use $t_i \in \{0, 1\}$ to represent the response time to a request i, where 1 represents early phase response (Δ_{t1}).

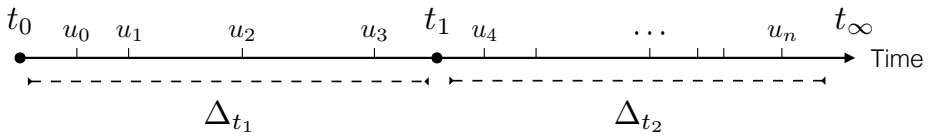


Fig. 4.3.2.: Users' response timeline to an app request

Response similarity: Because bot users share the same app response pool from the attacker, their responses to the same requests should be highly similar. As we mentioned, RecDroid's recommendation is based on majority voting of expert users. In other words, if most of the responses to a request are consistent with the ones from expert users, then with a high probability the system's recommendation would be the same as the majority

response. Attackers are more likely to succeed if the bots' responses are consistent and out-number the regular users' responses. We can use this feature along with the two other features to cluster bot users together.

The purpose of the defined features is to use them to define the distance between users so that we can apply a clustering method to group bot users. The proposed method is designed to capture key information from the proposed features in order to distinguish malicious users from regular ones.

Table 4.3.1.: Notations

Notation	Description
$\mathcal{D}$	$\{D^1, ..., D^n\}$: Set of n RecDroid users profiles.
n	The total number of users in the system.
m	The total number of app requests in the system
$\mathcal{S}(i,j)$	The distance function.
D^i_{pr}	The set of app requests and corresponding responses for user i.
D^i_{pr1}	The set of app requests and corresponding early phase responses for user i.
ω_1, ω_2	The assigned weights (0.5) to the features

4.3.3.3 Similarity Calculation

In this section we present our proposed distance function (similarity function) that we use in BotTracer. Since RecDroid users' profiles are high-dimensional data (high number of parameters) such as the set of installed apps, responses to requests and the time of responses, none of the commonly used distance functions are applicable to our case. In order to calculate the distances between users, we propose a *weighted* distance function, which weights the proposed features and aggregates them into the similarity between users. The details of the aggregation function is described as follows.

Let $\mathcal{D} = \{D^1, ..., D^n\}$ denote the set of n RecDroid users' profiles. The user i's profile can be presented by a set of app requests that user i responded, the responses, and corresponding response time, written as $D^i = \{\{p_1, r_1, t_1\}, \{p_2, r_2, t_2\}, ..., \{p_k, r_k, t_k\}\}$, where k is the number of app requests user i has responded. Note that $p_j \in \{1, 2, 3, ..., m\}$, $r_j \in \{0, 1\}$, and $t_j \in \{0, 1\}$, $\forall j \leq k$.

Let $\mathcal{S}(i,j)$ denotes the distance function between users i and j. The common set of responses by both users i and j can be written as $|D^i_{pr} \cap D^j_{pr}|$, where D^i_{pr} is the set of responded requests and corresponding responses for user *i*. We add ω_1 as the weight so we have $|D^i_{pr} \cap D^j_{pr}|\omega_1$. To integrate the timing into the function, we multiply the size of common set of requests responded by both users within early response time period, denoted by $|D^i_{pr1} \cap D^j_{pr1}|$, by ω_2, where D^i_{pr1} is the set of responded requests that are responded in early period by user *i* and their corresponding responses. Note that $\omega_1 + \omega_2 = 1$. Finally, we divide the sum of the above two values by the size of common set of requests. Equation (4.17) demonstrates the formula. Table 4.3.1 lists the notations we use in this section.

$$S(i,j) = \frac{|D^i_{pr} \cap D^j_{pr}|\omega_1 + |D^i_{pr1} \cap D^j_{pr1}|\omega_2}{|D^i_p \cup D^j_p|} \tag{4.17}$$

Using the proposed distance function, we are able to calculate the similarity of users. Since BotTracer works based on the distance of objects and not similarity, then we define the distance between user i and j to be $\mathcal{L}(i,j) = 1 - \mathcal{S}(i,j)$.

4.3.3.4 Clustering Method

In terms of the clustering method, we chose *hierarchical* clustering method. In data mining and statistics, hierarchical clustering (also called hierarchical cluster analysis or HCA) is a method of cluster analysis, which seeks to build a hierarchy of clusters. Strategies for hierarchical clustering generally fall into two types: *Agglomerative* and *Divisive*. We use Agglomerative type, which is a "bottom up" approach: each observation starts in its own cluster, and pairs of clusters are merged as one moves up the hierarchy. In general, the merges and splits are determined in a greedy manner, which is based on the calculated distances from our proposed distance function [83]. The results of hierarchical clustering are usually presented in a *dendrogram*. A dendrogram is a tree diagram frequently used to illustrate the arrangement of the clusters produced by hierarchical clustering.

4.3.4 Experimental Results

In this section we describe our experiments to evaluate the proposed model. We conducted a set of experiments to evaluate the accuracy of our proposed *HMM* model.

4.3.4.1 Simulation Setup

As a proof of concept, we created a set of RecDroid users' profiles (maximum 300 users) consisting of 11 different user groups including 10 bot groups controlled by different attackers and one regular user group. We assign a number of bot users (from 10 to 15 randomly) to each bot group and 150 users to the regular group. All malicious user groups consist of users with a high level of expertise. The expertise levels of regular users are uniformly distributed between $(0, 1)$. It is noteworthy as an evaluator that we have the correct responses to all apps' permissions.

We set the total number of apps in the system to be 550 and among them 50 are malicious apps. Each user in the system installs 10 distinct apps. However, for regular users the 10 apps are randomly chosen from the entire app pool (550 of them), while the bot users apps are randomly chosen from a smaller set selected by their masters (see Section II-C).

We configured each app to contain 5 permission requests. We generated both malicious and regular users' responses to permission requests based on their expertise levels. The higher the expertise level, the higher probability the response is correct. We recorded each response from a user using a tuple $\{p_k, r_k, t_k\}$, where p_k is the k^{th} request number; r_k and t_k are the corresponding responses and responding time period. All responses are recorded using a set D^i for user i.

Our simulation environment is MATLAB 2015 on a Windows machine with 2.5Ghz Intel Core2 Duo and 4G RAM. All experimental results are based on an average of 10

repeated runs with different random user profiles. All experiments are run numerous times and the average value is plot in results.

4.3.4.2 Performance and Accuracy

We use hierarchical clustering method to group users. The clustering result can be represented by a dendrogram tree. Figure 4.3.3 illustrates a dendrogram of one of our experiments. Each user is represented by a vertical blue line. When two lines join together, then they are clustered together. As we can see in Fig 4.3.3, users are merging at different heights in the dendrogram tree. A given distance threshold results in a number of clusters. For example, when the cut-off threshold is 0.6 (red line), it gives us three clusters (red boxes). Note that if the number of users in a detected cluster is higher than our threshold (10) we consider it as a *finalized cluster*.

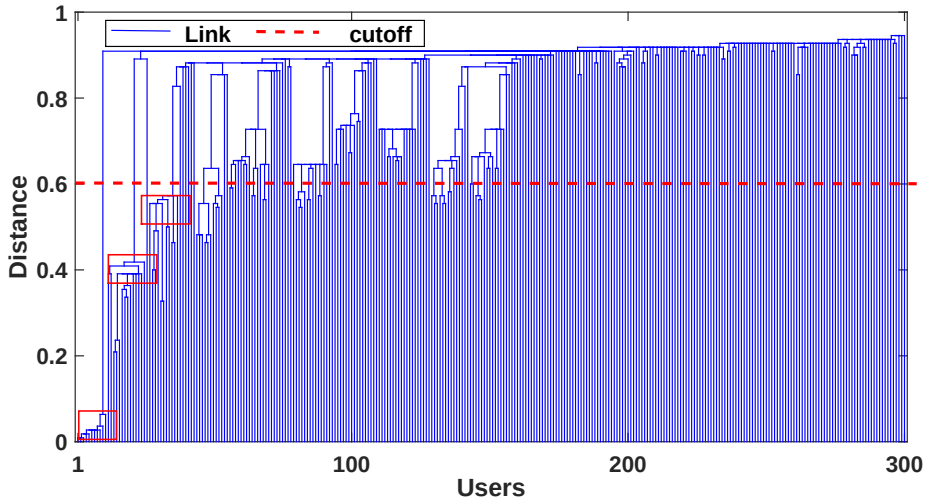


Fig. 4.3.3.: Dendrogram clustering results with given threshold.

In the first experiment, we study the impact from the number of malicious apps that a bot group has handled to the number of detected malicious groups, given different cutoff thresholds. Fig 4.3.4(a) shows that the number of detected malicious groups increases when each bot group handles more malicious apps. This is because when a bot group cheated

on more malicious apps, their bot users' similarity increases, thus easier to be discovered. However, regardless of the malicious apps quantity, an appropriate cut-off threshold can lead to higher cluster detection rate. The figure also shows that, in most cases, the optimal cut-off thresholds are between 0.6 to 0.8. When the threshold raises higher than 0.8, the number of detected clusters drops abruptly. This is because almost all users are clustered together.

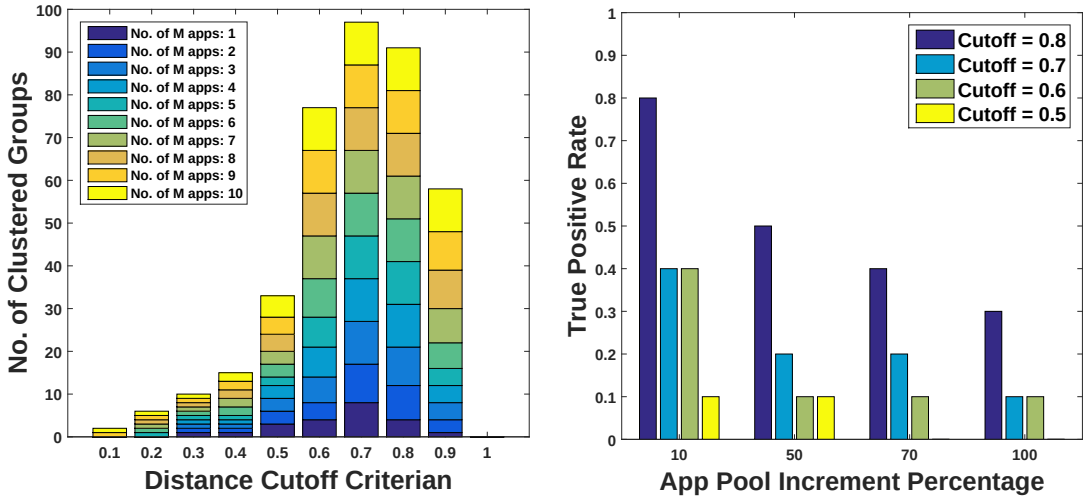
In order to verify the validity of the detected clusters by BotTracer, we carried out a cluster evaluation analysis. Since we have the highest number of clusters at cut-off 0.7, we evaluated all the detected clusters at this cut-off. Table 4.3.2 represents the evaluation results for all different numbers of malicious apps settings when the cut-off is 0.7. We calculate the F-measure (F-score) value using Equation (4.18). As we can see, by increasing the number of malicious apps, our clustering solution groups more users with a higher precision and recall. In addition, since the F-measure calculation is based on the precision and recall factors, it increases consequently.

$$F_{measure} = 2.\frac{precision.recall}{precision + recall} \tag{4.18}$$

Table 4.3.2.: Evaluation of clustering results

Feature	Precision	Recall	F-Measure	Clustered	Total
Malicious apps					
1	29%	100%	45%	36	123
2	41%	100%	58%	52	126
3	61%	100%	76%	83	134
4	69%	100%	81%	87	126
5	72%	100%	84%	90	124
6	79%	100%	89%	103	128
7	83%	100%	91%	110	122
8	90%	100%	94%	126	126
9	95%	100%	97%	123	123
10	100%	100%	100%	121	121

In the rest of the experiments, we will evaluate the accuracy of bot detection. In this section we use the *false negative rate*(FN) and *false positive rate*(FP) to measure the



(a) (b)

Fig. 4.3.4.: Impact of the number of responded malicious apps: (a) number of detected user groups for different number of malicious apps and cutoffs; (b) the influence of size of app pool on true positive rate.

detection accuracy of BotTrace. FN is the probability that a bot is not detected, while FP is the probability that a regular user is classified as a bot. Equation 4 show the formula we use to compute the FN and FP, where set *GT* contains the ground truth of all users (regular or bot) and set *A* contains the corresponding detection results. Note that $a_i, b_i \in \{0(regular), 1(bot)\}$.

$$A = \{a_1, a_2, ..., a_n\}\ GT = \{b_1, b_2, ..., b_n\}$$

$$FP = \frac{\sum_{i=1}^{N} a_i(1-b_i)}{\sum_{i=1}^{N}(1-b_i)}, \qquad FN = \frac{\sum_{i=1}^{N}(1-a_i)b_i}{\sum_{i=1}^{N} b_i} \tag{4.19}$$

As we described previously, an attacker should prepare a pool of correct responses to apps requests so that bot users can use them to boost their expertise levels. However, sharing common apps increases similarity among bots and make them easier to detect. In the second experiment, we observe the impact from the apps pool size that attackers use

to the true positive detection rate and false positive detection rate, given different cutoff settings. In this experiment, we change the size of the app pool for each group of users by 10%, 50%, 70% and 100% percentages. Fig 4.3.4(b) illustrates the results. As we can see the larger the pool size an attacker uses, the less likely it gets detected. This is because when the app pool is small, bot users tend to have high similarity since they have high overlap on the app choices and responses. However, building large app pool introduces high cost for attackers therefore not practical for attackers.

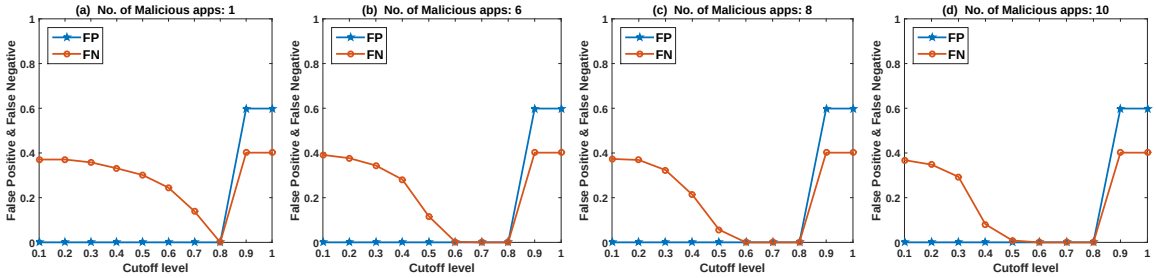


Fig. 4.3.5.: Influence of the responded malicious apps quantity on the accuracy of the clustered users.

In the last experiment, we study the impact from the cut-off threshold on the FN and FP bot detection ratios. Fig 4.3.5 depicts the results under different number of malicious apps settings. As we can see, with a higher number of malicious apps the bonnet handles, the more separated the regular users and bot users. This is because the more bots cheat on malicious apps, the higher similarity among them. In all cases the cut-off threshold 0.8 gives the lowest FP and FN rates.

4.3.5 Conclusion

RecDroid is an Android permission recommendation system which allows users to grand app permission request in a fine-grained manner in real time, and receive assistance from RecDroid recommendation based on expert users. However, attackers can create fake users (bots) to cheat in the system in order to mislead RecDroid to make incorrect recommendation. In this section, we propose BotTracer, a clustering-based method to detect bot

users in RecDroid. We first analyze the RecDroid recommendation mechanism, including how experts are evaluated and how recommendations are made. We then analyze the attacking strategies for attacker using bot users. In order to detect the bot users in RecDroid, we first define a similarity function between users based on their app overlap and response similarity. We then propose a customized distance function to measure the similarity between users. After that, we use a hierarchical clustering method users to group users based on their distances. Our evaluation based on simulated malicious and regular users has shown BotTracer has a strong performance and reliability. As our future work, we plan to apply other clustering methods such as k-means or some advanced clustering algorithms and compare them with the hierarchical method.

CHAPTER 5

PERMISSION NOTIFICATION MODEL

In this chapter, we study how to evaluate the usability of our privacy preserving framework and propose a permission risk notification mechanism of Android OS. We also discuss the shortcomings of the current smartphone security systems and propose a novel notification mechanism to enhance the usability for general smartphone users.

5.1 Permission Notification

In this section, we design a permission privacy notification mechanism to help users with different backgrounds to understand the actual privacy risks of apps. We also propose a set of usability measurement metrics to evaluate the usability of the notification system.

5.1.1 Problem Definition

Mobile devices have unprecedented access to sensitive personal information. While users report having privacy concerns, they may not actively consider the privacy risk while downloading apps from smartphone application marketplaces. Currently, Android users have only the Android permission request notification, which appears once an app attempts to access a resource and the users can choose to grant or deny the requests [71]. The current privacy notification interface provides little information to help users understand the risk of granting the permission requests. In this work, we will study how privacy notifications can play an active role in assisting users in making correct decision regarding whether to grant a permission request or not. To address this issue, we plan to design an effective privacy notification mechanism that helps user to understand the risk behind granting a

permission by taking user's technical background into consideration. The implementation of our model will include enhancing User Interface (UI), interpreting apps' activities risks, and users' preferences. We will also propose a set of metrics to evaluate the usability of the notification system. In order to evaluate the usability of our mechanism, we will conduct a set of user surveys.

5.1.2 Introduction

Current smartphone devices and mobile operating systems regulate applications access to all or a certain number resource permissions such as (e.g., SMS, camera, network, etc.). Different devices and operating systems have different strategies in handling the resource access permissions. For example, previous versions of Android requested all required permission by apps at the installation time. Android users had to grant all requested permission to apps up-front to be able to use the apps. On iOS and Android M and later, operating systems prompt users at runtime the first time applications request any of permissions on devices [17].

Research has shown that a small number of smartphone users read the Android installation-time permission list and requested permissions [7]. There are a few reasons that not many smartphone users pay attention to the privacy notifications or permission requests at the installation-time. First, users tend to rush through the installation process and install apps and use. Second, users are not aware of the fact that the privacy and security risks that an additional permission may cause can be very serious and result in private information leakage or other types of security threats. Third, users are not concerned about the privacy risks of granting unnecessary and excessive permissions to apps. This can be also a combinations of the last two reasons. Fourth, lack of knowledge in IT or mobile privacy and security is another major challenge. Users with lower level of knowledge in IT, may not be able to understand the permission requests and actions (grant or deny) they should take.

Fifth, considering the fourth reason and also the existing permission request notification models, users are left with “grant” or “deny” questions with no assistance. Therefore, in all above mentioned reasons, the result would be granting excessive permissions to apps [44]. Figure 5.1.1 shows permission notification for iOS, Android and Windows smartphones. They all have almost the same strategy and similar UIs and questions.

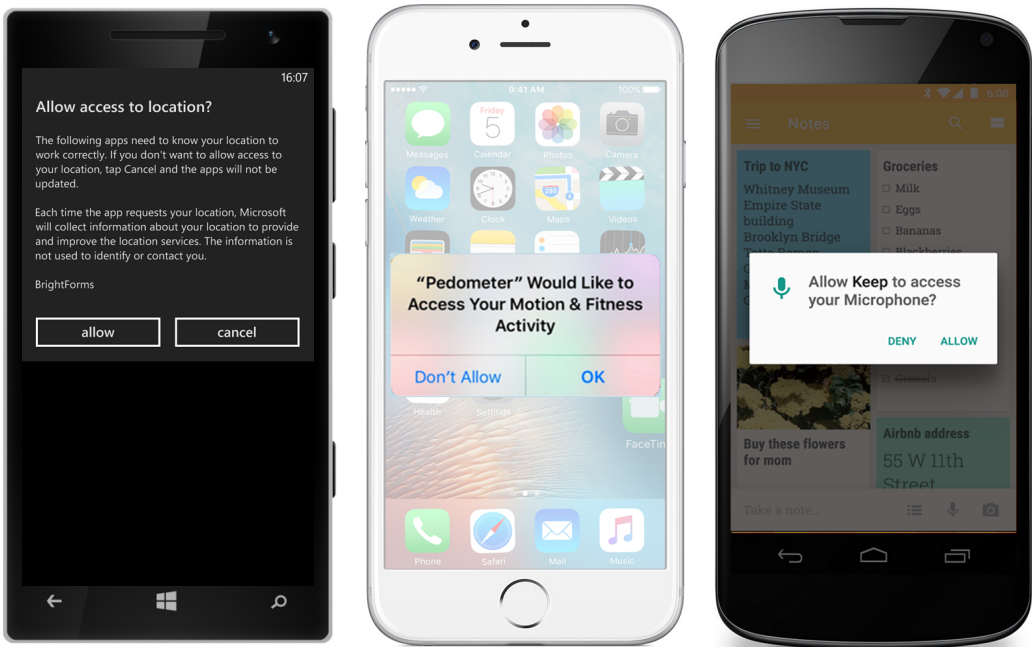


Fig. 5.1.1.: Current iPhone, Android and Windows phone’s privacy notification

In order to address such issue, there two important factors that need to be considered; *users knowledge* and *model design*. These two factors need to be considered before designing any permission notification model. We discuss these two factors in details in the next section.

In our proposed model, we consider all the important factors in designing an effective permission notification model. In this model, users’ knowledge, understanding of the permission notification, and also the design of the the UI are the key factors we consider.

5.1.3 Model

In this section we elaborate the proposed model in details. We first explain the factors we consider in our model and then our UI design process.

5.1.3.1 Factors

As we previously mentioned, two important factors that we consider in our model are users knowledge and the design of user interfaces.

Users knowledge: as one of the key factors in privacy and security notifications models, users' knowledge has a high impact. Regardless of the design of existing permission request notification models in which they consider all users the same and show users the same notification. This results in an ineffective permission notification model. We all are aware of the fact that users have different levels of knowledge when it comes to security, privacy, and IT. This is a challenge in designing uniform (existing models) permission notifications.

The users knowledge and background include the amount of knowledge that they have in IT, their education level, etc. The likelihood of a making random decisions on granting permission by a user who has never been exposed to information on IT is higher that a user who is at least familiar with IT concepts including permissions. The same fact can be true about users with a small knowledge and users with high level of IT knowledge. Therefore, in our model we consider users knowledge as an important factor.

Model design: design of a permission notification model is a challenging process. Considering the preferences, knowledge, and users attitudes towards security and privacy issues, makes it a challenging task to satisfy all users. Lack of considering these factors in design of a permission notification system results in an ineffective model. The consequences of a poorly designed model are not just limited to users satisfaction. It may

also result in private information leakage, financial loss, and serious security and privacy threats. The design of a permission notification has two aspects. First, what to put in the notification so that users can understand the content of it. This aspect becomes a complex task when users come from different backgrounds and have different understandings of the content. Second, the interface design of a permission notification is another important aspect of the design process. The user interface (UI) has a high impact on the success of delivering the actual privacy risk of permissions. Therefore, it is vital to consider these factors in designing any security and privacy model.

5.1.3.2 Multi-Interface

Users knowledge level has a direct relation with understanding the contents of a notification. To assist users, in our model, we design multiple interfaces containing the information about the requested permissions from apps. Each interface is called a *view* of information. Each view has three features defined as follows:

Granularity: we define the granularity to be the format, scale and quantity of information we put into a notification. For example, the actual activity logs of an app are considered as fine grained and showing the overall risk of an app is considered as coarse grained.

Intricacy: intricacy refers to the complexity of the information in terms of understanding, technical level and also interpretation by users. As an example, the actual activity logs are considered as high intricacy (because it needs knowledge of app API and system calls) and overall risk (the risk is presented in common spoken language) of an as low intricacy.

Co-equality: co-equality refers to the consistency of the information we put into the notification. For example, a skilled user's interpretation from the actual logs of apps should be the same or close to a user with low expertise from the overall risk of apps.

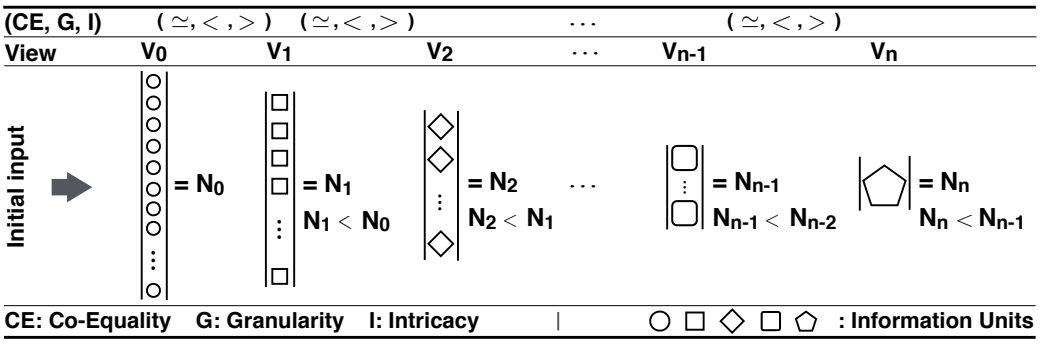


Fig. 5.1.2.: The process of generating multiple views of privacy risks to be presented to users

Figure 5.1.2 shows an overall view of the views we have designed in our model. As you can see, multiple views have different characteristics in terms of granularity, co-equality and intricacy. The format of contents in each view differs from other views (from left to right the granularity decreases). In addition, the level of intricacy is decreasing as well.

5.1.3.3 User Interface

In this case-study we provide users multiple views to present the actual privacy risks of Android apps running on devices, so that they can select their preferred view that they can interpret the best. Fig. 5.1.3 shows the designed views. The views are different from each other in terms of the features we mentioned above. The information that these views provide are as follows:

- *View 1*: This interface provides the most detailed information about applications' behaviours and activities (Privacy related raw data is shown).

- *View 2*: This interface does some risk assessment process on the information provided at level 0. The red numbers are assessed risks of the app. The red lines are suspicious activities that the app have had during runtime.

- *View 3*: This interface shows the assessed risk level of each requested permission

(resource) by the application. The assessed risk is course-grained to 5 levels.

- *View 4*: This interface shows the overall risk level of the app and possible misused resources.

- *View 5*: This view suggests that an app is malicious or not.

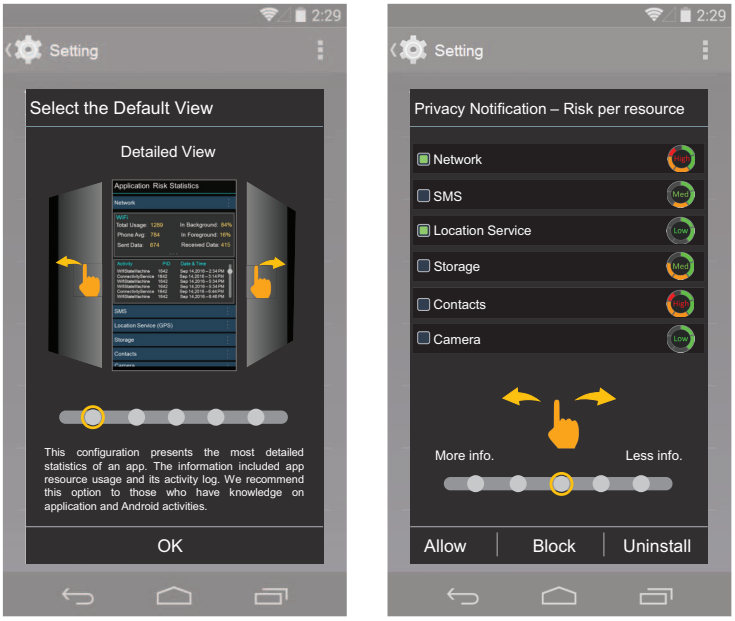
User preference: The interaction portal (interfaces and setting) is to facilitate the interaction between users and devices. Through this component, we enable users to manage the privacy risk of permission notification of their devices. They can change the configuration and select their preferred views to monitor the risk of installed apps.

In our model we designed the system in way that users can choose their preferred views manually or automatically in which users leave it up to the system. The system shows privacy risks of the system and requested permissions to users through the selected views. If a user does not select a view, system selects one as a default view to be used. In our design, users are able to select views per app. Fig. 5.1.4(b) shows the portal that users can see what views are being selected for what apps. Users are able to change it at any-time.

Users will be informed about the privacy risks in three different ways. First, they can find the information at the device's setting. Second, they can receive the information at runtime using pop-ups. Third, they can see the information at the installation time. In the third way, the information that users will see is calculated from other users in the community (those who have previously installed and used the app).

Action recommendation: In general, following the permission notifications, users are asked to take an action on apps. In the existing solutions, there are only two options "allow" or "deny". However depending on the activities and behaviours of apps, these set of actions

Fig. 5.1.3.: User Interfaces of 5 the views: (a) shows View 1 with highest of level intricacy; (b) shows View 2 which is similar to view 0 but it includes some assessed risks; (c) illustrates View 3 with the assessed risks for every requested resource; (d) shows View 4 that includes an overall risk of the app; (e) shows View 5 shows that an app is malicious or not



(a) (b)

Fig. 5.1.4.: Mode selection for privacy risk views (a) illustrates user interface of selecting a mode to see the views; (b) shows the interface of selecting views for both modes; (c) shows the interface of list of apps and their views.

may not work. For example, if an app is malicious and has access to a set of sensitive resources, denying a single permission may not block the privacy and security risks. The proper actions in such case would be uninstalling the app to avoid this scenario.

In addition to the multi-interface model, we also provide users with a set of actions they can take. The actions vary depending on the risk of apps. For example, if the risk of app is high, the actions we recommend users to take are different from an app which is low risk. For each notification that our model generates, we also recommend a set of actions. Some actions are more strict than others. Figure 5.1.5 shows a set of actions buttons we recommend to users. For example, Figure 5.1.5(a) is recommended when apps are low risk and it is not necessary to recommend more strict actions. Figures 5.1.5(c)(d) recommend more strict actions including "uninstall". In the case of uninstall, we also provide users with a detailed explanation to inform them about the reasons behind the action we recommend.

Using the action recommendation system, we not only show users the risk behind

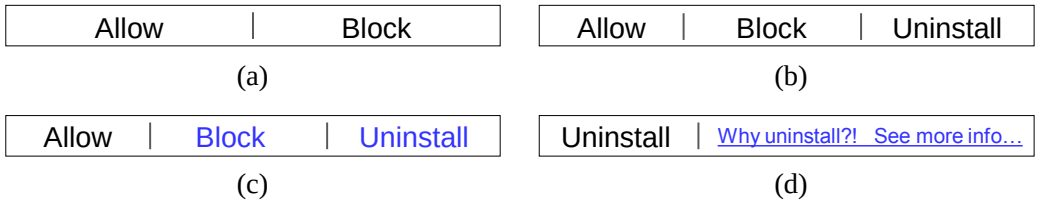


Fig. 5.1.5.: Designed buttons for the views.

apps, but also a set of proper actions that users can take.

5.1.4 User Study

In this section we discuss the results of a survey we conducted to evaluate the usability of our model. We tend to answer the following questions in our user study; "*Is there a need/interest for a privacy/security notification system showing different views with different amount of information about the privacy/security risk of apps/permissions?"* and also if the answer to the first question is yes, then "*Is the interest distributed among all views or only one or two specific views?*".

5.1.4.1 Study setup

In order to answer the questions mentioned in this section, we designed a survey to collect users feedback on the proposed model. Our user survey included three parts; "Lab study", "Training" and "Feedback collection".

Lab study: before we publish the survey on Amazon MTurk we conducted a lab study to resolve any ambiguity with the survey, so that participants will not have any issue to understand the topic of our survey and also the contents are easy to read and understand. During this step, we made a set of corrections and redesigned the survey.

Training: before asking users' opinions, we decided to give necessary information to participants, so they understand the concept and technical terms in our survey. We explained the concept of permission notification to participants together with explanation of the single view (existing models) and also our model (multi view). To avoid making any

bias in participants, we designed the survey to be neutral and only focusing on the facts about the models.

Feedback collection: in order to collect feedback from users we used Amazon MTurk platform to publish our survey. We set the language and location of participants to be English speaking countries and English respectively. The reason behind this was that the content of the survey was in English. This helped us to have a successful training process. Figure 5.1.6 shows the targeted locations to launch our user study. We set the locations to be US, UK, Canada, Australia and New Zealand.

Table 5.1.1 presents the education level of participants. We set the education to "High-School", "Undergraduate" and "Graduate and above". Table 5.1.2 shows the demographic information related to the age of participants. We did not set the configuration for the age in our survey, so that we could get feedback from a wide range of ages. Finally, Table 5.1.3 shows the diversity of gender among our participants. In our survey, we decided to keep a balance in number of participants from each gender.

In order to avoid making any bias in our survey, we decided to move the demographic information collection step to the end of survey. This way, participants will not be affected by the answers they give to the demographic information. Therefore, when participants are answering the preference questions, they do not have to give answers based on the answers they give to our demographic information.

Table 5.1.1.: Diversity of Participants (Education level)

Educational Level	High-School	Undergrad	Graduate and above
Number of participants	41	104	55

Table 5.1.2.: Diversity of Participants (Age)

Age	20-30	30-40	40-50	50-60
Number of participants	53	82	41	24

Table 5.1.3.: Diversity of Participants (Gender)

Gender	Female	Male	Others
Number of partiipants	90	107	3

Fig. 5.1.6.: Location of participant of our user study.

5.1.4.2 Model Preference

In this section, we present users' feedbacks on the single-view and multi-view models. Before we collect their feedbacks, we presented them a neutral explanation of the functionalities and features of each model. In this step, we only focused on the facts and features of the models. We designed this section in a way that users will navigate through the views. For each view, we also added an explanation, so that participants understand the features of each view.

Figure 5.1.7 shows the preferences of participants. As you can see, 170 of participants in our survey preferred the multi-view model and only 30 participants prefer to user the single-view model. This shows that participants significantly prefer to use the multi-view model.

In order to confirm the fact the participants have a clear understanding of both models, we decided to ask their reasons on choosing the models as a requirement. This way, we can make sure that they understand what they choose. We received 25 words per sentence in

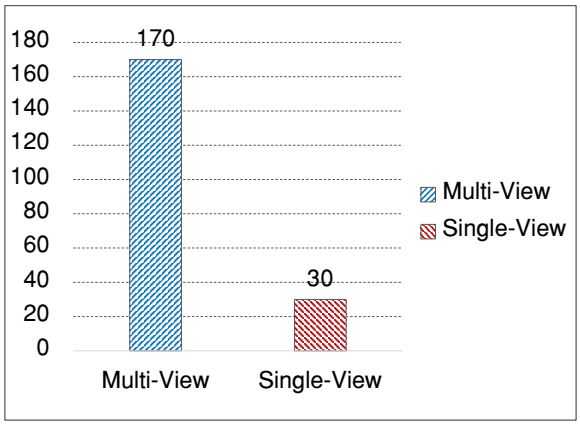


Fig. 5.1.7.: Participants model preferences in terms of single and multiple views.

average from participants, which is sufficiently long to understand/interpret their reasons. We used ”voyant-tools” word processor to do an analysis on the inputs from participants. Table 5.1.4 shows the most common phrases among the participants feedbacks for both models. As highlighted in the table, we can see that participants have a clear understanding of both models. The core reason behind choosing single-view is “*simplicity*” and “*less information*”. In contrast, those who chose multi-view mention “*more features*”, “*interest in more information*” and also “*”ability to choose*”. From these feedbacks, we can conclude that participant had a good understanding of models before they chose. They mainly pointed out the actual reasons we set to behind each model.

Table 5.1.4.: Users feedback on their preferred model

Model	Single-View	Multi-View
Phrases	’easier to use’	’a lot more information’
	’less information ’	’risks of applications’
	’simple’	**’has more features’**
	’less details’	’seems more useful’
	’simplistic’	**’i would like to know more’**
	’easy to understand’	’more options’
		’I can choose’

In order to see the correlation among some of the factors in our survey and the model preference, we measured the preference of participants with respect to “*malicious app experience*” and “*security concern*” factors. Figure 5.1.8(a) shows the model preference among

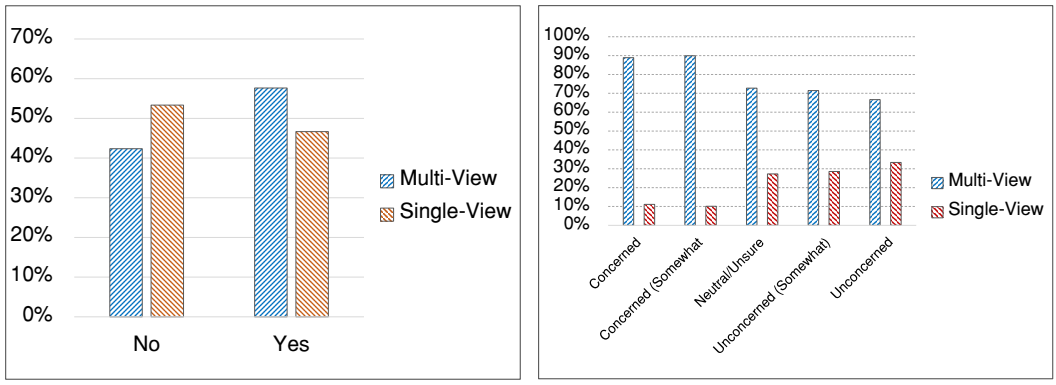
users who have experienced mobile malware apps. In order to make sure that participants have a good understanding of what a malware is, we explained this term during the training process. As you can see, those who have experienced malware, prefer the multi-view model over the single-view. The reason behind this may be the fact that they want to know more about the permission risks of apps. Additionally, those who have experienced malware on their devices, are already familiar with the concept of malware and they have a better understanding of malicious apps. We also noticed this fact from participants feedback in Table 5.1.4 (understanding of malicious activity and risks).

As security concern was one of the items we collected participants' answers, we decided to see the relation between the model preference with respect to the security concern of participants. Figure 5.1.8(b) shows the model preference of participants with considering the security concern. As you can see, participants with higher level of concern are more interested in using the multi-view model.

The feedback from participants answers the question "*Is there a need/interest for a privacy/security notification system showing different views with different amount of information about the privacy/security risk of apps/permissions?"*. 85% of participants preferred the multi-view model which is designed to give users more insight about their apps. In addition, users with different backgrounds and security concerns, chose our proposed model over the single-view model. Therefore, the answer to this question is that users are in need of such system and also they have the understanding of what they need.

5.1.4.3 View Preference

In this section, we analyze the feedback from participants who prefer to use the multi-view model. We wanted to see if the view preferences are being distributed among different views. The distribution of preference among views shows the need for a model with different views containing different types of information about the privacy risks of permission.

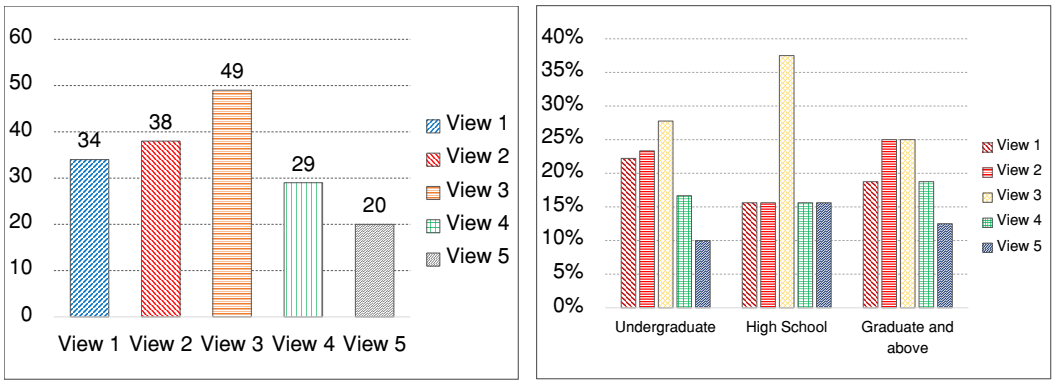


(a) (b)

Fig. 5.1.8.: Participants preferences in terms of single and multiple views: (a) Participants model preference by experience of malware; (b) Participants model preference by security concern.

Those participants who chose the multi-view model as their preferred model, were later ask to choose the view they prefer the most among all views. They were presented with details for each view including the features, purpose of the view, and the type of information included in the view. Figure 5.1.9(a) shows the distribution of participants' preferences among the views. As you can see, each view is being selected by a number of participants. This answers the questions of "*Is the interest distributed among all views or only one or two specific views?*". View 3 is the most preferred view among all views. We believe the reason behind this is the fact that this view provides a moderate amount of risk information and at the same time it is visually appealing. We can see that 34 and 38 participants are interested in view 1 and 2 respectively, which is surprisingly higher than our expectations.

We also calculated the correlation between view preference, gender and malware experience. Figure 5.1.10(a) shows the correlation between the gender factor and view preference. The correlation between female and male participants for all views is 0.71. As you can see, except for the View 2, both gender categories follow the same pattern. That is the reason behind the 0.71 correlation. It is worth mentioning the a correlation of 1 means that the two variables/factors always follow the same pattern. Figure 5.1.10(b) shows the correlation between malware experience and view preference. The calculated correlation



(a) (b)

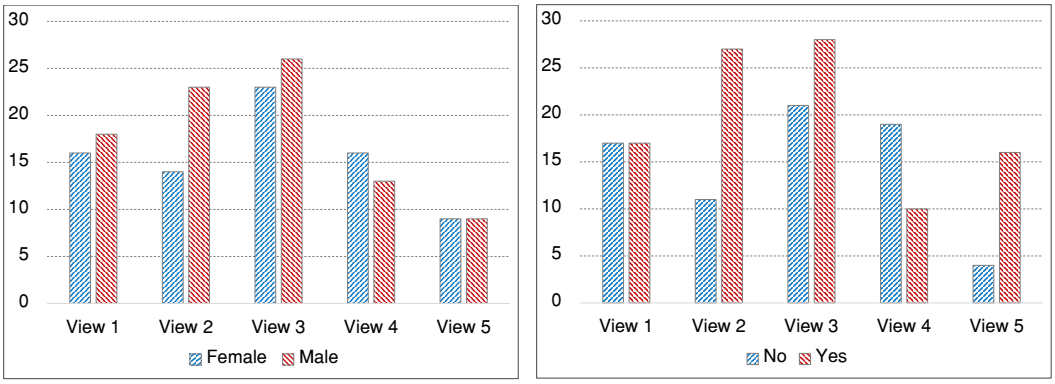
Fig. 5.1.9.: Participants preferences in terms of single and multiple views: (a) Participants model preference by experience of malware; (b) Participans model preference by security concern.

value for this factor was 0.07, which is very small. This shows that there is almost no correlation between experiencing malware and views that participants have selected. View 3 is the only view that both categories follow the same pattern.

As we did for the model preference, we required participants to leave a reason for the view they prefer. This way, we can make sure that they understand views they chose. We received 27 words per sentence in average from participants, which is sufficiently long to understand/interpret their reasons. Table 5.1.5 shows the processed feedbacks from participants. The core phrases from participants are highlighted in the table. As you can see, participants have correct understanding of the core purposed behind each view.

Table 5.1.5.: Users feedback on their preferred view of the multi-view model

View	View 1	View 2	View 3	View 4	View 5
Phrases	'most information' **'detailed information'** 'most detailed'	**'has risk assessment'** 'detailed information' **'malicious activity'**	**'multiple risks'** 'assessed risk' 'not much details' 'like the 5 level' **'visually appealing'** 'easy to understand'	'summarized' 'easy to understand' **'overall risk'** **'summary of risk'** 'informative' 'shows misused resources'	**'simple'** 'quick info.' 'less confusing'



(a) (b)

Fig. 5.1.10.: Correlation between gender, malware experience, and view preference: (a) Correlation between gender and view preference; (b) Correlation between malware experience and view preference.

5.1.5 Discussion

In this section we discuss some of the challenges related to the model. We also propose our solutions for the challenges. As we previously mentioned, consistency among the designed views is a key aspect. Users with different levels of knowledge should be able to understand/interpret the views the same. In addition, as we mentioned, recommending actions to users for each view is another challenge.

5.1.5.1 Consistency

Consistent views can help users to make correct decisions regardless of their background and knowledge. To be able to achieve such consistency, there should be a control system to evaluate the consistency of the views. Our proposed solution to address this challenges it to equip the model with *Control Theory* to be able to tune the views and enhance the consistency of views.

The main objective of control theory is to control a system. It helps a system to control the input-output flow and also make sure that the system's output follows a desired value. Control theory controls the input flow of a system and influences the behaviour of dynamic

systems [86]. On the other words, it models a (non)physical system, using mathematical modeling, in terms of inputs, outputs and various components with different behaviours, use control systems design tools to develop controllers for those systems and implement controllers in physical systems employing available technology. This way, we can achieve a certain level of control over the behaviour of systems and manipulate them to the system behave in a desirable manner [19].

In order to be able to apply control theory to our model, we made modification in the architecture of a control theory unit. Fig. 5.1.11 illustrates the basic architecture of an adapted control theory system. We made changes in the processing unit of a control theory unit. The reason we made this change is that we needed to evaluate the quality of view before we generate it. That is why we made a short-cut in the model and until the qualification are not met our model does not generate the views. The components are explained as follows:

Processing Unit: A plant of a control theory system is the part of the system to be controlled. On the other words, a part of system that is responsible for generating the output is usually referred to as the plant. The usual objective of control theory is to control plant, so its output follows a desired value, called the reference (r), which may be a fixed or changing value.

Input Quality Estimator: The controller (compensator or simply filter) determines the setting of the control input needed to achieve the reference input. The controller computes values of the control input based on current and past values of control error. On the other words, The controller provides satisfactory characteristics for the total system.

Input Adaptation: The system for measurement of a variable (or signal) is called a sensor. The sensor transforms the measured output so that it can be compared with the reference input (e.g., smoothing stochastic of the output). It monitors the output and compares it with the reference. The difference between actual and desired output, called

the error (e) signal, is applied as feedback to the input of the system, to bring the actual output closer to the reference.

Figure 5.1.11 shows an adapted version of a control theory unit. In this unit, the input is risk information we need to generate a view. The information can be the raw logs of apps or an evaluated risk. Depending on the design of a view, the units can change. The change can be the format of input required to generate the view and also the format of output for the view. The controller component of each unit, measures the qualification of the view. It measures the quality of input and makes decision whether the input and output requirements are being met or not.

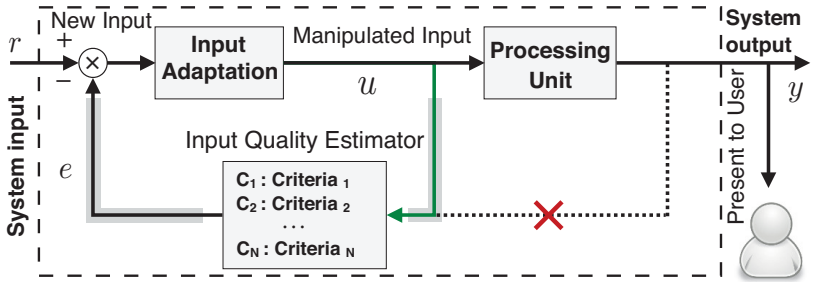


Fig. 5.1.11.: A chain of closed-loop control systems.

Because there are multiple views in out proposed model, we need a chain of nodes to be able to generate all the views. Figure 5.1.12 shows a chain of units. Except the first unit, the output of the previous unit is the input for the next unit. Using such model, we can make sure that each view is being controlled in terms required qualifications. The controller can be customized depending on the policies and rules we set. As you can see, there should be a set of criteria to meet. It is worth mentioning that to be able to evaluate the consistency, we also need to have an evaluation process after applying such model. The evaluation process can be in the shape of user survey and collecting users feedback to evaluate the fact that they have the same understanding of the views.

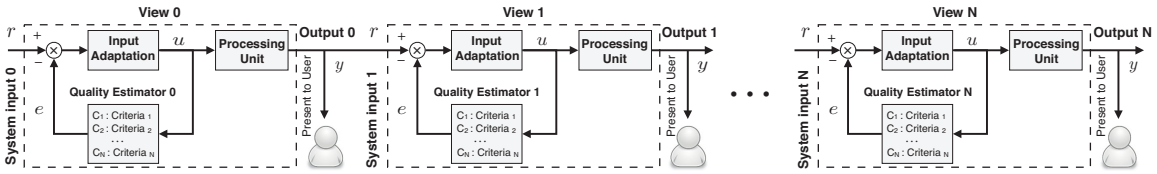


Fig. 5.1.12.: A chain of closed-loop control systems.

5.1.5.2 Action recommendation

As recommending actions (buttons) in our views and interfaces is a key part of the model, there should be a mechanism on which actions should be selected and offered to users. For such mechanism, we mainly rely on the actual risk of apps. A threshold-based system can be a solution to such challenge. Defining a set of thresholds and mapping each risk threshold to a set of actions is the solution. For example, for an app with high risk activities, we can define a set of "Block" and "Uninstall" and for apps with low risk we offer milder actions such as "Block" and "Deny".

5.1.6 Conclusion

In this section we proposed a model for generating multiple permission notifications. Our model considers users background and knowledge and generates views with different levels of intricacy and granularity. Using this model, users have the option to choose their preferred interface and view. Users can choose the view that they can understand the most and are more comfortable using it. Our user study shows that users are more interested in have a multi interface permission notification mechanism. Additionally, our study also shows that users' interests are distributed among different views. Users with different backgrounds have different preferences. Based on our study, users with higher concerns in terms of security and privacy of their smartphones and personal information, have higher interest in our multi view model. Finally, our user study showed that there is a need for such model and also users are interested in multiple views instead of one.

CHAPTER 6

SIMILAR SAFE APPLICATIONS RECOMMENDATION

In this chapter, we design and evaluate an Android secure application recommendation system that provides users with fine-grained and customizable app recommendation. This app recommendation system shall take the security aspect of apps into consideration through a security scoring method based on requested permissions. We will discuss the design, scoring metric and algorithms, and the graphical user interface for users to configure their preferences on rating metrics.

6.1 Similar Safe Applications

Android app recommendation system is an important feature for our user privacy preserving framework. Besides providing similar apps, the recommendation also takes the privacy and security aspect of the apps into account. The goal of this project is to provide a list of similar apps with the lowest privacy risks. This topic is one of our future works toward dissertation completion.

The contribution of this work can be summarized as follows:

- We propose a novel Android app recommendation system that takes the security features as well as three other app metrics into consideration.
- We propose multiple methodologies to score the apps on all corresponding metrics.
- We evaluate the effectiveness of the system and compare it to the existing Google recommendation using real data.

6.1.1 Problem Definition

In current Android systems, the application recommendation function is an important feature that users can use to find a similar application to replace a targeted one. The current recommendation system provided through Google and the Google Play store presumably recommends applications similar to a target application while accounting for the popularity of each application. However, it does not take the security features of each application or users preferences into consideration when doing so. In this project, we will design and develop an Android feature that provides users with fine-grained and customizable application recommendations. Compared to the Google store recommendation function, the new feature not only consider the similarity of the apps, but also other metrics such as popularity, security, and usability. More specifically, it allows users to configure the weight of the metrics that can be used to rate the apps. We also provide a recommendation algorithm that generates a list of recommended applications based on the combined scores. We will evaluate the usability of our system and the quality of recommendations and compare it to the existing Google recommendations.

6.1.2 DroidVisor: A Safe Application Recommendation System

In the current Android systems, the application recommendation function is an important feature that users can use to find a similar application to replace a targeted one. The current recommendation system provided through Google and the Google Play store presumably recommends applications similar to a target application while accounting for the popularity of each application. However, it does not take the security features of each application or user preferences into consideration when doing so. In this section, we propose DroidVisor, an Android tool that provides users with fine-grained and customizable application recommendations. Compared to the Google store recommendation function,

DroidVisor not only use the similarity to a preselected target application, but also considers other metrics such as popularity, security, and usability. More specifically, DroidVisor provides an interface for users to configure the weight of each metric and a recommendation algorithm that generates a list of recommended applications based on the combined scores. We evaluate our proposed criteria and the quality of recommendation through use case studies. Finally, we present our findings through a discussion of accuracy as well as possible ways to improve our recommendation results.

6.1.2.1 Background

To compute the similarity of two applications, we use an NLP approach. In this section, we discuss the Latent Dirichlet Allocation (LDA) [20]method and Lesk algorithm that are used to find similarity scores between apps.

Latent Dirichlet Allocation LDA [20], is described as a generative probabilistic model that can be used to model the topics of a document. LDA is a three-level hierarchical Bayesian model, in which each item of a collection is modeled as a finite mixture over an underlying set of topics. Each topic is, in turn, modeled as an infinite mixture over an underlying set of topic probabilities [20]. The primary goal of LDA is to define a general list of topics that represent key terms behind a document or set of documents. After each topic is defined, different word types are classified and placed into each topic.

Learning for Language Toolkit: The MAchine Learning for LanguagE Toolkit, referred to as MALLET, is a Java-based package for statistical natural language processing, document classification, clustering, topic modeling, information extraction, and other machine learning applications to text [51]. The MALLET topic modeling toolkit contains efficient, sampling-based implementations of Latent Dirichlet Allocation that we will be utilizing

later on in order to visualize outputs in a clear, textual format.

MALLET has multiple optional input parameters, but we need not concern ourselves with the utilization of most of them. We are however concerned with training MALLET, as it involves machine learning, and having it acclimate to different application genres and classifications. For the required inputs, we need the location of our applications descriptions, the number of "topics" we want, and the destination the output should be placed in.

Sense Disambiguation In the context of Natural Language Processing, Word Sense Disambiguation (WSD) is the process of identifying the “sense" of a word, or there meanings in the context of which they reside, that may possibly have multiple meanings. It is an approach that uses a words relative proximity to another in text and then attempts to analyze the context of which it resides in to find common themes.

The practice involves use of a predefined dictionary with the inclusion of homonyms, or words that sound the same. As every dictionary has a different interpretation of what a word can mean contextually, alternatively known as the "sense" of a word, problems arise when different dictionaries or thesaurus are available. One solution to this is the use of a common data bank known as WordNet. WordNet is a computational lexicon that encodes concepts as synonym sets

Lesk The Lesk algorithm, introduced by Michael E. Lesk in 1986, is based on the assumption that words in a given text file or description will tend to share a common topic or meaning. A simplification of Algorithm 8, which is a modified version of the original Lesk algorithm written by Satanjeev Banerjee and Ted Pederson [16], would typically be used to compare the dictionary definition of an ambiguous word with the terms contained in its neighborhood or surrounding section.

We later modify our own version of Algorithm 8 and use it to form our matrix visualization of textual similarity between applications.

Algorithm 8 LESK Algorithm Pseudo-Code

```
1: for all word w[i] in in the phrase do
2:     let BEST_SCORE = 0
3:     let BEST_SENSE = null
4:     for all sense sense[j] of w[i] do
5:         let SCORE = 0
6:         for all other word w[k] in the phrase, k != i do
7:             for all sense sense[l] of w[k] do
8:                 SCORE = SCORE + num of words that occur
9:                 occur in the gloss of both sense[j] and sense[l]
10:            end for
11:        end for
12:        if SCORE > BEST_SCORE then
13:            BEST_SCORE = SCORE
14:            BEST_SENSE = w[i]
15:        end if
16:    end for
17:    if BEST_SCORE > 0
18:        output BEST_SENSE
19:    else
20:        output "Could not disambiguate w[i]"
21:    end if
22: end for
```

6.1.2.2 DroidVisor Design

DroidVisor is an Android app recommendation system that takes the security aspect of apps into consideration through a novel security scoring method based on requested permissions. In this section, we discuss the design of DroidVisor, including the scoring metric and algorithms, and our graphical user interface for weighing metrics and recommendations.

Selection When providing app recommendations to users, the first step is to identify several metrics that users may be interested in, but which data are also available. The four metrics that we adopted are described as follows:

Similarity: Similarity is the core metric since the purpose of recommendation is to find replacement apps that are both similar and secure. The similarity of two apps can be

measured through comparing the textual descriptions of the apps. The description typically contains the app's purpose, what users may be able to achieve through usage, the device requirements, or what a developer may want to mention.

Security: The security risk of apps is also a critical concern for smartphone users. An important novelty of DroidVisor is taking the security risk of apps into account in the recommendation. The security risk level of apps can be estimated through the permissions an app requests.

Popularity: Popularity can be also taken into consideration for the recommendation algorithm because users tend to pick more popular apps to install. The popularity can be measured using the number of downloads by users.

Usability: Usability measures how well the apps are designed and function, which is another typical metric that users care about. Usability can be measured by user ratings given to each individual app on a scale from 0.0 to 5.0.

Design DroidVisor uses the process shown in Figure 6.1.1 to compute the recommendation scores in order to find the recommend apps. The process can be divided into four key steps: *initial filtration*, *keyword filtration*, *metric scoring computation*, and *normalized sorting*. Once the final similarity score, σ, is calculated, DroidVisor then displays the highest scored apps to its users as shown in Figure 6.1.2(a). When a user chooses to see the details of an app, it will be displayed in a fashion identical to Figure 6.1.2(b).

Initial filtration: In order to access the available apps, we did some primary filtration based on category (genre) to narrow down the scope of the initial search. We did this through a simple process that reads the category information of each app from its description in the dataset. We start with the set of apps Google Play lists as similar to the target app and recursively obtain similar apps in regards to our target app until the set is of sufficient size. Finally, we only keep the apps if they are in the same category of the target app and

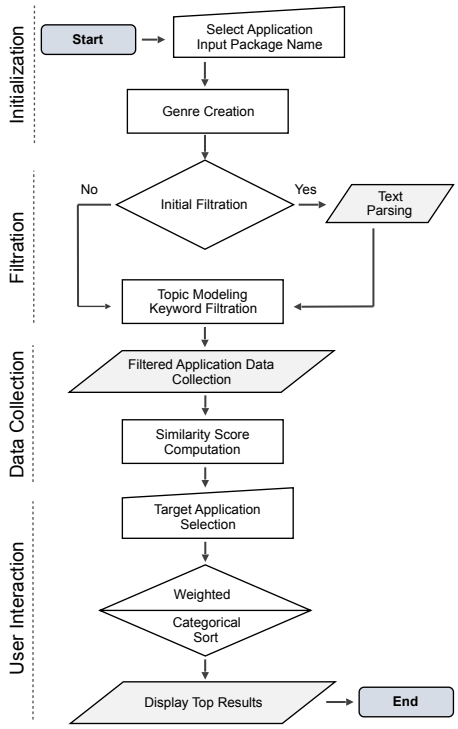
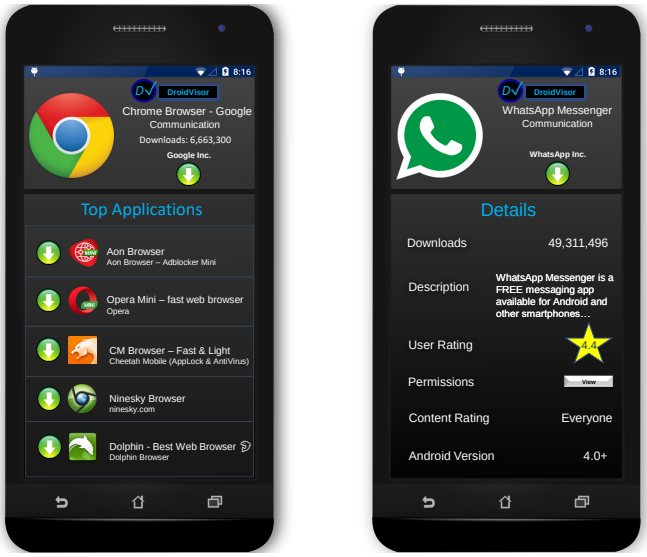


Fig. 6.1.1.: DroidVisor's process flowchart.



(a) (b)

Fig. 6.1.2.: User Interfaces: (a) illustrates Google Chrome's related apps using our proposed model; (b) shows more details of popular messenger application WhatsApp.

abandon the rest as they are not relevant enough to analyze.

Keyword filtration: After initial filtration, the list of apps that we have descriptions of is still relatively large. We then further filter the apps based on keywords. This is done by assigning each genre a list of non-generic keywords and setting a chosen threshold in which the minimum number of keywords in an apps description must be met for it to be added to our final pool. It can also be done by placing a cap on how many apps are allowed in score calculation. If we choose to employ the app capping method, we simply analyze *N* number of apps with the highest number of keywords in common with our target app. This is done through a process known as *Topic Modeling*, or *Latent Dirichlet Allocation* , using the *MAchine Learning for LanguagE Toolkit* (MALLET).

Once our app specific keywords have been defined, we then go through the process of checking each text description in the remaining target apps pool and assigning them scores corresponding to the number of keywords each of their respective texts contain. For example, if we had the textual description of "A dog jumps over a log" with the keyword pool $\{dog, wolf, rabbit, log, boat\}$ that particular app and description would receive a matching score of 2 due to containing "dog" and "log".

We are now given the choice of placing apps into our final pool based on whether they meet a threshold for a minimum number of keywords, or our capping method by selecting N amount of apps with the highest amount of matching keywords. In order to keep consistent sizes for our final app pools amongst various different genres and target apps we chose to employ our capping method setting N, our final app pool size, to 75. If in the anomalous case that our target app did not make it into its own final app pool we choose to remove the app with the lowest keyword matching score in our pool and replace it with our target app.

Metric score computation: At this stage we have established a pool of related apps

Permissions	Risk
Install shortcuts	L
Set an alarm	L
Control vibration	L
Network connection status	L
Device app history	M
Close other apps	M
Make app always run	M
Retrieve running apps	M
Photos/Media/Files	H
Read contacts	H
Get location	H
Use microphone	H

Table 6.1.1.: Examples of permissions and their security risk levels.

to the target app. By utilizing *Natural Language Processing* (NLP) overlap measures, we create a matrix visualization of app *similarity scores* for all apps inside of our pool using the *Word Sense Disambiguation* (WSD) algorithm known as Lesk. We then visualized and mapped each score in an adjacency matrix format, where each column and row represented a separate app and the matrices elements representing how related one apps description is to another.

To calculate the *security scores* we first compute the *risk score* by counting the number of permissions requested by each application. We analyze each application and its individual permissions and assign each of the permissions a value of "L", "M", or "H", corresponding with low, medium, and high risk when accessed by an application in an isolated manner (meaning not in concurrence with any other permissions). The examples of permission risks are shown in Table 6.1.1. We assign different weights to different levels, where low/medium/high correspond to weight $0.33/0.67/1.0$ respectively. Once we find the score corresponding to each permission we then sum them together. An application score for the popularity and user rating categories are taken by looking at the downloads and user rating metric provided by Google Play.

In addition, the *usability scores* can simply be the rating scores, and the *popularity scores* can be computed by taking the logarithm of the number of downloads an app has and then normalizing all popularity scores by dividing using the highest value found. All

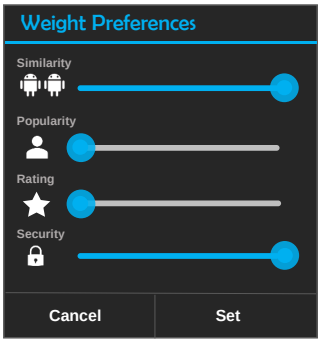


Fig. 6.1.3.: GUI design of weight tuner for DroidVisor.

four scores of each app are stored in a matrix for easy access.

Normalized scoring and sorting: The scoring methods on each metric produce metrics scores with different range scope. For example, the scores of usability is within $[0, 5]$ while the scores of popularity is within $[0, \infty]$. Our next step is to normalize them into the range of $[0, 1]$. This can be done by dividing each individual apps score with the highest value found in that metric. Note that $\bar{\beta}_j^i$ is the normalized score for metric i and app j and β_j^i is the unnormalized score.

$$\bar{\beta}_j^i = \beta_j^i / \max_{\forall j}(\beta_j^i) \tag{6.1}$$

To tune and customize results for each user we incorporated a weighted sort or slider to be placed into the graphical user interface (GUI), which can be seen in Figure 6.1.3. Each individual user is able to select how much is the importance of each metric, with each section of the slider ranging from 0.0 to 1.0. Each normalized score is then multiplied out by the chosen weight each user has picked. These totals per app are then summed together to simply see which apps receive the highest score, as shown in the equation below:

$$\sigma_j = \sum_{i=1}^{3} \alpha^i * \bar{\beta}_j^i + \alpha^4 * (1 - \bar{\beta}_j^4) \tag{6.2}$$

where σ_j refers to the total score for app j; α_i is the weight of metric i set by the user. Note that the *security score* is the complement of the *risk score* due to their opposite meanings.

6.1.2.3 Evaluation

In order to evaluate our proposed approach, we conducted a set of experiments following the DroidVisor design.

Experimental Setup To evaluate our work on a real-life app, we use the Google Chrome app, with the associated package name `com.android.chrome`, as an example. We collected the initial pool of apps, as described in Section 4.3.3, to populate our target app genre, which is"Communication".

In our sample, the actual size of the initial pool before any form of filtration was 471 unique apps. After the initial filtration it is diminished to 251 apps. We then collect the textual descriptions of each app, provided their respective developer, and stored them for more reliable access.

While the previous filtration narrowed our pool size, we are still relying on genre classification to further reduce the pool. As described in the filtration step, we did this through a process called Topic Modeling utilizing Latent Dirichlet Allocation and a tool developed by Andrew McCallum called MALLET.

After carefully analyzing the results and accounting for pool degradation, we came to consensus that 2 is a suitable number of topics for MALLET in which we have a sufficient number of keywords for various description lengths, yet repetition of words remains minimized to avoid any possible topic overlap (see Table 6.1.2). We continued on with our final filtration step, which generated our complete app pool of 75 on which we perform our comparisons as shown in Figure 6.1.4.

Table 6.1.2.: Evaluation of number of topics.

Keywords / Topics	Total	Unique	Non-unique	Overlap
1	19	19	0	0.00%
2	38	37	1	2.63%
3	57	55	2	3.51%
4	76	73	3	3.95%

After completing the app pool we moved on to evaluating the relevance and metrics of each of the apps that remains using our four primary categories. Obtaining our popularity and rating scores and then normalizing them was straight forward, however we had to compute our similarity and security parameters.

To compute our remaining apps similarity scores we needed to compare the textual description in the final pool of apps to one another. We did so using a Perl implementation of the Lesk algorithm. To get our permission scores we used our permission calculation method.

Once all of the four categorical scores were calculated, and normalized, we multiplied them with each of their corresponding α weights, or slider weights given by the user, as seen in Table 6.1.3, to get our σ scores. We recorded each response and visualized the top apps for "Chrome Browser - Google" in a fashion similar to Figure 6.1.2(a).

Table 6.1.3.: Trial category weights.

Category / Trial	α Similarity	α Popularity	α Usability	α Security
Trial 1	1.0	0.0	0.0	0.0
Trial 2	1.0	0.5	0.25	0.0
Trial 3	1.0	0.0	0.0	1.0

Our experiment environment is the IDE Eclipse Mars on a Windows 10 machine with a 2.6Ghz Intel i7 Core and 12G RAM. All experiments were run numerous times to confirm accuracy and are represented graphically as well as tabularly.

Trial 1 The results in our first trial were completed in order to emulate how we imagine the Google Play store currently displays top apps. We tuned the weight values of each category to its respective value shown in Table 6.1.3.

These values were then multiplied by DroidVisor's four categorical scores calculated for each measure to result in the final σ each app receives for that particular category. Table 6.1.4 shows our evaluation for the most relevant apps with highest scoring criteria and overall σ value when we used "Chrome Browser - Google" as our target app. These apps

represent the top-most relevant apps out of our final pool of 75 apps relating to our target app.

Table 6.1.4.: Evaluation of Trial 1.

Category / App	β Similarity	β Popularity	β Rating	β Security	σ Total
Aon Browser	1.00	.444	.851	.278	1.00
Opera Mini - fast web...	.409	.836	.936	.139	.409
CM Browser - Fast & Light	.333	.818	.957	.147	.333
Ninesky Browser	.278	.602	.894	.156	.278
Dolphin - Best Web...	.275	.830	.957	.102	.275

Trial 2 The results in our second trial were completed in an identical fashion to our first trial, to emphasize the addition of the popularity and usability categories in conjunction with the similarity category previously tuned in Trial 1 section. In order to see if we could recommend a list of apps which are slightly more well known amongst app users, we put medium emphasis on the popularity category and slight emphasis on usability (see Table 6.1.3) when tuning our α values in order to see if we could recommend a list of apps which are slightly more well known amongst app users.

Table 6.1.5.: Evaluation of Trial 2.

Category / App	β Similarity	β Popularity	β Rating	β Security	σ Total
Aon Browser	1.00	.444	.851	.278	1.43
Opera Mini - fast web...	.409	.836	.936	.139	1.06
CM Browser - Fast & Light	.333	.818	.957	.147	.982
WhatsApp Messenger	.199	1.00	.936	.075	.933
Dolphin - Best Web...	.275	.830	.957	.102	.929

Trial 3 We again follow the previous trials for our third trial in order to display the core purpose behind DroidVisor. We put heavy emphasis on the similarity and security categories (see Table 6.1.3) when tuning our α values in order to see if we could recommend a list of apps which focus on the users safety, privacy, and security.

Table 6.1.6.: Evaluation of Trial 3.

Category / App	β Similarity	β Popularity	β Rating	β Security	σ Total
Aon Browser	1.00	.444	.851	.278	1.28
Viber Messages & Ca...	.230	.686	.872	1.00	1.23
Opera Mini - fast web...	.409	.836	.936	.139	.548
Ghostery Privacy Browser	.196	.523	.872	.294	.490
CM Browser - Fast & Light	.333	.818	.957	.147	.480

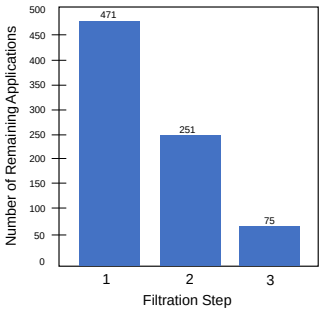


Fig. 6.1.4.: Progression of app filtration steps for the “Chrome Browser - Google" app.

Results Discussion We can see that with respects to the similarity feature, which is prioritized in Trial 1, DroidVisor’s quality of recommendations seems much higher than those of the current Google Play store. This is seen by simply examining the amount of browsers that appear in the top list of apps similar to “Chrome Browser - Google” (see Table 6.1.7). DroidVisor’s top 5 apps all consist of browsers while only 2 of the 5 apps Google Play recommend, Firefox and Opera Mini, are browsers.

While some of the apps that appear in our similarity list may not be well known, we adjust for this in Section Trial 2, by increasing popularity and rating weights, and the out-

DroidVisor	Google Play
Aon Browser	Gmail
Opera Mini - fast web...	WhatsApp Messenger
CM Browser - Fast & Light	Firefox. Browse Freely
Ninesky Browser	Messenger
Dolphin - Best Web...	Opera Mini - fast web...

Table 6.1.7.: DroidVisor’s versus Google Play’s similar app recommendation for “Chrome Browser - Google".

puts again look promising. However, our observed results for DroidVisor start to appear more closely related to the recommendations by the Google Play Store with the addition of the popular messaging app "WhatsApp Messenger", which currently has over 49 million downloads, confirming our hypothesis of Google accounting for downloads when recommending apps.

The first drastic change we see in DroidVisor's recommendations is when we tune to adjust for the addition of the security feature. We see the addition of two new apps to our top list. Immediately we notice one of these apps, "Viber Messages & Calls Guide", does not match the criteria of actual similarity to our app in the sense of its intended purpose. When analyzing why this app was added to our top list we can immediately see the reason was because it requests very few permissions from the user. However, the second new app, "Ghostery Privacy Browser", matches our tuned criteria weights almost perfectly. It is both a browser and places heavy emphasis on security and privacy, as indicated by its name and permission usage.

Overall results look promising with minor refinement being needed in regards to textual analysis, which is completely dependent on the developers part, and the possible inclusion of handling permission usage and requests in conjunction with one another to assess the true risk level of an app.

6.1.2.4 Conclusion

DroidVisor is an Android application recommendation system which allows users to select a target application, define preference to weight recommendation metrics, and then display a list of applications a user may find more usable than their selected target application. We propose the four metrics a user may find relevant are textual similarity, number of downloads, user rating, and security. Our evaluation based on real data has shown that in its current form the outputs are satisfying at recommending highly similar and secure apps.

As our future work, we plan to apply other categories into our summation/score measure in order to create a more accurate representation of what applications may fit the needs of a user.

REFERENCES

[1] What is the price of free. http://www.cam.ac.uk/research/news/what-is-the-price-of-free.

[2] Appslib android apps market, Last Visit: August, 2015. http://slideme.org/.

[3] Appslib android market, Last Visit: August, 2015. https://appslib.com/.

[4] F-droid - free and open source android app repository, Last Visit: August, 2015. https://f-droid.org/.

[5] Bit9 report: Pausing google play: More than 100,000 android apps may pose security risks, Last Visit: May, 2015. https://www.bit9.com/files/1/Pausing-Google-Play-October2012.pdf.

[6] Gartner: 1.1 billion android smartphones, tablets expected to ship in 2014, Last Visit: May, 2015.

[7] Mcafee q3 2011 threats report shows 2011 on target to be the busiest in mobile malware history, Last Visit: May, 2018. https://www.mcafee.com/hk/about/news/2011/q4/20111121-01.aspx.

[8] Yuvraj Agarwal and Malcolm Hall. Protectmyprivacy: Detecting and mitigating privacy leaks on ios devices using crowdsourcing. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 97–110, New York, NY, USA, 2013. ACM.

[9] Tansu Alpcan and T Basar. A game theoretic analysis of intrusion detection in access control systems. volume 2, pages 1568–1573.

[10] Ron Amadeo. App ops: Android 4.3's hidden app permission manager, control permissions for individual apps! http://www.androidpolice.com/2013/07/25/app-ops-android-4-3s-hidden-app-permission-manager-control-permissions-for-individual-apps/.

[11] Shahriyar Amini. Analyzing mobile app privacy using computation and crowdsourcing. In *Dissertations*, 2014.

[12] Chinmayee Annachhatre, Thomas H. Austin, and Mark Stamp. Hidden markov models for malware classification. *Journal of Computer Virology and Hacking Techniques*, 11(2):59–73, 2015.

[13] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*. The Internet Society, 2014.

[14] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. Boxify: Full-fledged app sandboxing for stock android. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 691–706, Washington, D.C., August 2015. USENIX Association.

[15] Thoms Ball. The concept of dynamic analysis. *SIGSOFT Softw. Eng. Notes*, 24(6):216–234, October 1999.

[16] Satanjeev Banerjee and Ted Pedersen. An adapted lesk algorithm for word sense disambiguation using wordnet. In *International Conference on Intelligent Text Processing and Computational Linguistics*, pages 136–145. Springer, 2002.

[17] David Barrera, Jeremy Clark, Daniel McCarney, and Paul C. van Oorschot. Understanding and improving app installation security mechanisms through empirical analysis of android. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 81–92, New York, NY, USA, 2012. ACM.

[18] Alastair R Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *HotMobile'11*, pages 49–54.

[19] Shankar P Bhattacharyya, Lee H Keel, and Aniruddha Datta. *Linear control theory: structure, robustness, and optimization*. CRC press, 2009.

[20] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, March 2003.

[21] Gerardo Canfora, Francesco Mercaldo, and Corrado Aaron Visaggio. An hmm and structural entropy based detector for android malware: An empirical study. *Computers & Security*, 61:1 – 18, 2016.

[22] L Carin, G Cybenko, and J Hughes. Cybersecurity strategies: The queries methodology. *Computer*, 41:20–26, 2008.

[23] Yang Chen, M. Ghorbanzadeh, K. Ma, C. Clancy, and R. McGwier. A hidden markov model detection of malicious android applications at runtime. In *Wireless and Optical Communication Conference (WOCC), 2014 23rd*, pages 1–6, May 2014.

[24] N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, 2000.

[25] Jonathan Crussell, Ryan Stevens, and Hao Chen. MAdFraud: Investigating ad fraud in android applications. In *12th CMSAS*, MobiSys '14, pages 123–134, New York, NY, USA, 2014. ACM.

[26] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S Wallach. Quire: Lightweight provenance for smart phone operating systems. In *USENIX Security Symposium*, 2011.

[27] G. Ditzler, M. Roveri, C. Alippi, and R. Polikar. Learning in nonstationary environments: A survey. *IEEE Computational Intelligence Magazine*, 10(4):12–25, Nov 2015.

[28] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol 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 393–407, Berkeley, CA, USA, 2010. USENIX Association.

[29] 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.

[30] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan. Android security: A survey of issues, malware penetration, and defenses. *IEEE Communications Surveys Tutorials*, 17(2):998–1022, Secondquarter 2015.

[31] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *18th CCS*, pages 627–638. ACM, 2011.

[32] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *UPS*, SOUPS '12, pages 3:1–3:14. ACM.

[33] D Fudenberg and J Tirole. Game theory. In *Game Theory*. MIT Press, Cambridge, Massachusetts.

[34] Carol J Fung and Raouf Boutaba. Design and management of collaborative intrusion detection networks. In *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, pages 955–961. IEEE, 2013.

[35] M. Gales and S. Young. *The Application of Hidden Markov Models in Speech Recognition*. Foundations and trends in signal processing. Now Publishers, 2008.

[36] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM Comput. Surv.*, 46(4):44:1–44:37, March 2014.

[37] Anjana Gosain and Ganga Sharma. A survey of dynamic program analysis techniques and tools. In Suresh Chandra Satapathy, Bhabendra Narayan Biswal, Siba K. Udgata, and J.K. Mandal, editors, *Proc. of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA'14), Odisha, India*, volume 327, pages 113–122. Springer International Publishing, November 2015.

[38] John C Harsanyi. Games with incomplete information played by bayesian players, i-iii part i. the basic model. *Management science*, 14(3):159–182, 1967.

[39] Qatrunnada Ismail, Tousif Ahmed, Apu Kapadia, and Michael K. Reiter. Crowdsourced exploration of security configurations. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 467–476, New York, NY, USA, 2015. ACM.

[40] J Jormakka and V.E. Molsa. Modeling information warfare as a game. volume 4(2), pages 12–25. Journal of Information Warfare, 2005.

[41] W. Khreich, E. Granger, A. Miri, and R. Sabourin. A comparison of techniques for on-line incremental learning of hmm parameters in anomaly detection. In *Computational Intelligence for Security and Defense Applications, 2009. CISDA 2009. IEEE Symposium on*, pages 1–8, July 2009.

[42] Wael Khreich, Eric Granger, Ali Miri, and Robert Sabourin. A survey of techniques for incremental learning of {HMM} parameters. *Information Sciences*, 197:105 – 130, 2012.

[43] Matthias Lange, Steffen Liebergeld, Adam Lackorzynski, Alexander Warg, and Michael Peter. L4android: A generic operating system framework for secure smartphones. In *SPSMD*, SPSM '11, pages 39–50, New York, NY, USA, 2011. ACM.

[44] Jialiu Lin, Shahriyar Amini, Jason I. Hong, Norman Sadeh, Janne Lindqvist, and Joy 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, New York, NY, USA, 2012. ACM.

[45] Jialiu Lin, Bin Liu, Norman Sadeh, and Jason I. Hong. Modeling users' mobile app privacy preferences: Restoring usability in a sea of permission settings. In *Symposium On Usable Privacy and Security (SOUPS 2014)*, pages 199–212, Menlo Park, CA, July 2014. USENIX Association.

[46] Bin Liu, Jialiu Lin, and Norman Sadeh. Reconciling mobile app privacy and usability on smartphones: Could user privacy profiles help? In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 201–212, New York, NY, USA, 2014. ACM.

[47] Bin Liu, Suman Nath, Ramesh Govindan, and Jie Liu. DECAF: Detecting and characterizing ad fraud in mobile apps. In *11th USENIX CNSDI*, NSDI'14, pages 57–70, Berkeley, CA, USA, 2014. USENIX Association.

[48] P Liu, W Zang, and M Yu. Incentive-based modeling and inference of attacker intent, objectives, and strategies. volume 8, pages 78–118.

[49] R. Liu, J. Cao, L. Yang, and K. Zhang. Priwe: Recommendation for privacy settings of mobile apps based on crowdsourced users' expectations. In *2015 IEEE International Conference on Mobile Services*, pages 150–157, June 2015.

[50] Kong Lye and Jeannette Wing. Game strategies in network security. *International Journal of Information Security*, 4:71–86, 2005.

[51] Andrew Kachites McCallum. Mallet: A machine learning for language toolkit. http://www.cs.umass.edu/ mccallum/mallet, 2002.

[52] Nikola Milosevic, Ali Dehghantanha, and Kim-Kwang Raymond Choo. Machine learning aided android malware classification. *Computers & Electrical Engineering*, pages –, 2017.

[53] Radhika Mittal, Aman Kansal, and Ranveer Chandra. Empowering developers to estimate app energy consumption. In *18th CMCN*, Mobicom '12, pages 317–328, New York, NY, USA, 2012. ACM.

[54] Jun Mizuno, Tatsuya Watanabe, Kazuya Ueki, Kazuyuki Amano, Eiji Takimoto, and Akira Maruoka. *Discovery Science: Third International Conference, DS 2000 Kyoto, Japan, December 4–6, 2000 Proceedings*, chapter On-line Estimation of Hidden Markov Model Parameters, pages 155–169. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[55] A. Narayanan, L. Yang, L. Chen, and L. Jinliang. Adaptive and scalable android malware detection through online learning. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 2484–2491, July 2016.

[56] Nir Nissim, Robert Moskovitch, Oren BarAd, Lior Rokach, and Yuval Elovici. Aldroid: efficient update of android anti-virus software using designated active learning methods. *Knowledge and Information Systems*, 49(3):795–833, 2016.

[57] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[58] Orlando R. E. Pereira and Joel J. P. C. Rodrigues. Survey and analysis of current mobile learning applications and technologies. *ACM Comput. Surv.*, 46(2):27:1–27:35, December 2013.

[59] L. R. Rabiner and B. H. Juang. An introduction to hidden markov models. *IEEE ASSP Magazine*, pages 4–15, January 1986.

[60] Lawrence Rabiner. First hand: The hidden markov model. *IEEE Global History Network*, pages 4–15, October 2013.

[61] Bahman Rashidi and Carol Fung. Disincentivizing malicious users in recdroid using bayesian game model. *Journal of Internet Services and Information Security (JISIS)*, 5(2):33–46, May 2015.

[62] Bahman Rashidi and Carol Fung. A game-theoretic model for defending against malicious users in recdroid. In *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on (IM'15), Ottawa, Canada*, pages 1339–1344, May 2015.

[63] Bahman Rashidi and Carol Fung. A survey of android security threats and defenses. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, 6(3):3–35, September 2015.

[64] Bahman Rashidi, Carol Fung, and Tam Vu. Recdroid: A resource access permission control portal and recommendation service for smartphone users. In *Proceedings of the ACM MobiCom Workshop on Security and Privacy in Mobile Environments*, SPME '14, pages 13–18, New York, NY, USA, 2014. ACM.

[65] Bahman Rashidi, Carol Fung, and Tam Vu. Recdroid: A resource access permission control portal and recommendation service for smartphone users. In *Proc. of the ACM MobiCom Workshop on Security and Privacy in Mobile Environments (SPME '14), Maui, Hawaii, USA*, pages 13–18. ACM, September 2014.

[66] Bahman Rashidi, Carol Fung, and Tam Vu. Dude, ask the experts!: Android resource access permission recommendation with recdroid. In *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*, pages 296–304, May 2015.

[67] Francesco Ricci, Lior Rokach, and Bracha Shapira. Introduction to recommender systems handbook. In Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B. Kantor, editors, *Recommender Systems Handbook*, pages 1–35. Springer US, 2011.

[68] Wilson Rothman. Smart phone malware: The six worst offenders. `http://www.nbcnews.com/tech/mobile/smart-phone-malware-six-worst-offenders-f125248`.

[69] M. Rychetsky. *Algorithms and Architectures for Machine Learning Based on Regularized Neural Networks and Support Vector Approaches*. Shaker Verlag GmbH, Germany, December 2001.

[70] J. Sahs and L. Khan. A machine learning approach to android malware detection. In *2012 European Intelligence and Security Informatics Conference*, pages 141–147, Aug 2012.

[71] Florian Schaub, Rebecca Balebako, Adam L. Durity, and Lorrie Faith Cranor. A design space for effective privacy notices. In *Eleventh Symposium On Usable Privacy and Security (SOUPS 2015)*, pages 1–17, Ottawa, 2015. USENIX Association.

[72] Burr Settles. Active learning literature survey. *Computer Sciences Technical Report*, 1648, 2010.

[73] Burr Settles and Mark Craven. An analysis of active learning strategies for sequence labeling tasks. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '08, pages 1070–1079, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics.

[74] Pooja Singh, Pankaj Tiwari, and Santosh Singh. Analysis of malicious behavior of android apps. *Procedia Computer Science*, 79:215 – 220, 2016. Proceedings of International Conference on Communication, Computing and Virtualization (ICCCV) 2016.

[75] The Statistics Portal Statista. Number of apps available in leading app stores as of june 2016. http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.

[76] Guillermo Suarez-Tangil, Mauro Conti, Juan E. Tapiador, and Pedro Peris-Lopez. *Computer Security - ESORICS 2014: 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part I*, chapter Detecting Targeted Smartphone Malware with Behavior-Triggering Stochastic Models, pages 183–201. Springer International Publishing, Cham, 2014.

[77] Steven Tadelis. Game theory: An introduction. Princeton University Press, January 6, 2013.

[78] A. J. Viterbi. A personal history of the viterbi algorithm. *IEEE Signal Processing Magazine*, 23(4):120–142, July 2006.

[79] Kevin Walsh and Emin Gün Sirer. Fighting peer-to-peer spam and decoys with object reputation. In *Proceedings of the 2005 ACM SIGCOMM Workshop on Economics of Peer-to-peer Systems*, P2PECON '05, pages 138–143, Philadelphia, Pennsylvania, USA, 2005.

[80] Songyang Wu, Pan Wang, Xun Li, and Yong Zhang. Effective detection of android malware based on the usage of data flow {APIs} and machine learning. *Information and Software Technology*, 75:17 – 25, 2016.

[81] Cui Xiaolin, Tan Xiaobin, Zahang Yong, and Xi Hongsheng. A markov game theory-based risk assessment model for network information system. volume 3, pages 1057–1061.

[82] Liang Xie, Xinwen Zhang, Jean-Pierre Seifert, and Sencun Zhu. pbmds: A behavior-based malware detection system for cellphone devices. In *Proceedings of the Third ACM Conference on Wireless Network Security*, WiSec '10, pages 37–48, New York, NY, USA, 2010. ACM.

[83] Rui Xu and II Wunsch, D. Survey of clustering algorithms. *Neural Networks, IEEE Transactions on*, 16(3):645–678, May 2005.

[84] Liu Yang, Nader Boushehrinejadmoradi, Pallab Roy, Vinod Ganapathy, and Liviu Iftode. Short paper: Enhancing users' comprehension of android permissions. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 21–26, New York, NY, USA, 2012. ACM.

[85] Xia You and Zhang Shiyong. A kind of network security behavior model based on game theory. pages 950–954.

[86] Jerzy Zabczyk. *Mathematical control theory: an introduction*. Springer Science & Business Media, 2009.

[87] Quanyan Zhu, Carol Fung, Raouf Boutaba, and Tamer Basar. A game-theoretical approach to incentive design in collaborative intrusion detection networks. In *Game Theory for Networks, 2009. GameNets' 09. International Conference on*, pages 384–392. IEEE, 2009.

[88] Quanyan Zhu, Carol Fung, Raouf Boutaba, and Tamer Basar. Guidex: A game-theoretic incentive-based mechanism for intrusion detection networks. *Selected Areas in Communications, IEEE Journal on*, 30(11):2220–2230, 2012.

VITA

Bahman Rashidi is currently pursuing the Ph.D. degree in computer science with the Virginia Commonwealth University. His research interests include distributed systems, mobile systems and privacy issues in smartphone devices. He received the master's degree in computer engineering from the Iran University of Science and Technology (IUST), Tehran, Iran, in 2014. He is the recipient of the Outstanding *Early-Career Student Researcher Award* and *Outstanding Research Paper Award* from the VCU computer science department in 2015 and 2018 respectively.