

國立交通大學

資訊科學與工程研究所

碩 士 論 文

以共同行為為基礎之三階式Android惡意程式偵  
測與分類



Three-phase Detection and Classification for Android Malware

Based on Common Behaviors

研 究 生：張育妮

指導教授：林盈達 教授

中 華 民 國 一 百 零 二 年 六 月

以共同行為為基礎之三階段 Android 惡意程式偵測與分類  
Three-phase Detection and Classification for Android Malware based on  
Common Behaviors

研究生：張育妮

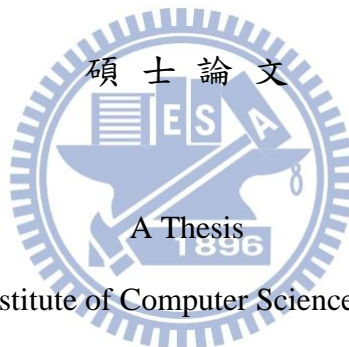
Student : Yu-Ni Chang

指導教授：林盈達

Advisor : Dr. Ying-Dar Lin

國立交通大學

資訊科學與工程研究所



Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

June 2013

Hsinchu, Taiwan

中華民國一百零二年六月

# 以共同行為為基礎之三階式 Android 惡意程式偵測與分類

學生：張育妮

指導教授：林盈達

國立交通大學資訊科學與工程研究所

## 摘要

Android 是目前行動裝置上最受歡迎的作業系統之一。其普及性也使得它常常成為攻擊者攻擊的目標。為了偵測和分類惡意程式，我們提出一個高偵測效能和高準確率之三階段行為分析法，前兩階段用於偵測惡意程式，最後階段用於分類惡意程式。較快的第一階段中，我們利用應用程式要求的權限與貝氏定理快速濾掉應用程式，以減少到較慢的第二階段分析的樣本數量。第二階段中，我們利用最長共同子字串和 N 元產生的系統呼叫序列偵測惡意程式。最後，我們利用行為或權限向量的餘弦相似度將惡意程式分類成已知類型或未知類型。本文顯示在偵測率方面，兩階段比一階段更準確，若第二階採用最長共同子字串產生系統呼叫序列，其偵測率與誤判率分別為 97% 和 3%；若採用權限向量分類，我們能正確辨識 98% 已知類型的惡意程式或新類型的惡意程式。

**關鍵字：**Android, 惡意程式, 行為分析, 權限, 系統呼叫, 貝氏定理, 最長共同子字串, N 元, 餘弦相似度

# Three-phase Detection and Classification for Android Malware Based on Common Behaviors

Student: Yu-Ni Chang

Advisor: Dr. Ying-Dar Lin

Department of Computer and Information Science

National Chiao Tung University

## Abstract

Android is one of the most popular operating systems adopted in mobile devices. The popularity also turns it an attractive target for attackers. To detect and classify malicious Android applications, we propose an efficient and accurate behavior-based solution with three phases. The first two phases detect malicious applications and the last phase classifies the detected malware. The “faster” first phase quickly *filters* out applications with their requested *permissions* judged by the Bayes model and therefore reduces the number of samples passed to the “slower” second phase which detects malicious applications with their *system call sequences* matched by the longest common substring (LCS) or N-gram algorithm. Finally, we classify a malware into known or unknown type based on *cosine similarity* of behavior or permission *vectors*. Our experiments show that the two-phase detection approach works more accurately than a single phase approach. It has a TP rate and a FP rate of 97% and 3%, respectively, with LCS in the second phase. More than 98% of samples can be classified correctly into known or new types based on permission vectors.

**Keywords:** Android, malware, behavior analysis, permissions, system call, Bayes, longest common subsequence, N-gram, cosine similarity

## 致謝

剛進入交大就讀研究所時，因為對環境感到陌生不適應，碩一時，總是有種撐不下去的感覺。因為家人們的鼓勵，使我熬過這難過的關頭，時間久了也漸漸適應了。

在碩士論文研究中，需要找與自己論文題目相關的研究，因為以前不曾讀過如此大量的英文論文，所以剛開始總是很吃力，但長時間看英文論文也提升了自己看英文的速度。從以前就自認為英文在聽力方面，比較差勁，但來到高速網路實驗室後，因為我們實驗室有外籍生，老師們總會用英文與外籍生溝通，我也利用此機會增進自己的英文聽力。

不論在修課期間或是論文研究期間，皆受到諸位師長、同學、朋友與家人的協助與鼓勵，實有說不盡的感謝話語。

首先感謝林盈達教授、賴源正教授和黃俊穎教授的費心指導。在每次討論過程中，他們總是耐心地指引做研究的嚴謹態度與撰寫論文所需的知識與技巧。當遇到瓶頸時，總是提出有建設性的建議使我突破難關。對於感到疑惑不解的地方，老師們總是不厭其煩地解釋或是舉出較易理解的例子，使我們更容易了解且記住。萬分感謝黃俊穎教授花費極大心力在我的論文上，不辭辛勞地指導與修訂論文的撰寫方向，使我最終能夠完成此篇論文。在此感謝老師們的教誨與包容。

感謝高速網路實驗室的同學們，從你們身上我學到了許多，像是重灌、組裝、拆解電腦、待人處事等。謝謝你們總是在我需要幫助時伸出援手，也讓我能夠在歡樂的氣氛下，度過這兩年碩士生涯。

最後感謝我的家人，能夠在我遭遇到挫折時給予支持和鼓勵，讓我能夠適應研究所生活並且無憂無慮的完成學業。

張育妮 謹致於

2013.06.06

# Contents

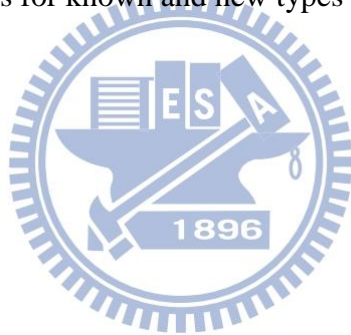
List of Figures .....	V
List of Tables .....	VI
Chapter 1. Introduction .....	1
Chapter 2. Background .....	4
2.1 Differences between PC Malware and Android Malware .....	4
2.2 Related Works.....	6
Chapter 3. Problem Statement .....	8
3.1 Notations .....	8
3.2 Problem Description .....	8
Chapter 4. Three-phase Behavioral Detection and Classification .....	10
4.1 Overview of Three-phase Behavioral Detection and Classification.....	10
4.2 Permission-based Detection (PBD) Phase .....	11
4.3 System Call-based Detection (SBD) Phase .....	12
4.4 Behavior-based Classification (BBC) Phase .....	14
4.5 Implementation .....	15
Chapter 5. Evaluation.....	17
5.1 Experiment Environment .....	17
5.2 Experimental Results .....	20
Chapter 6. Conclusions and Future Works .....	30
References.....	32

## List of Figures

Figure 1. Overview of the proposed solution.....	11
Figure 2. Procedure flowchart of SBD .....	13
Figure 3. Procedure of system call sequence comparator.....	14
Figure 4. Procedure of type classifier .....	15
Figure 5. Procedure of permissions analyzer.....	16
Figure 6. Procedure of system call recorder .....	16
Figure 7. Detailed modification of the init.rc file .....	16
Figure 8. Experiment environment .....	18
Figure 9. Distribution of the permissions requested by benign and malicious applications	21
Figure 10. The permissions requested by benign applications and malware.....	22
Figure 11. Performance for PBD .....	22
Figure 12. Detection performances for system call sequences with various N .....	24
Figure 13. Distribution for the number of malicious behaviors.....	24
Figure 14. Accuracy comparison for one-phase and two-phase detectors.....	26

## List of Tables

Table 1. Related works to detect malicious Android applications.....	6
Table 2. Definition of notations .....	9
Table 3. Number of training samples and detecting samples .....	19
Table 4. List of malware types.....	19
Table 5. Detection performance vs. time consumption .....	25
Table 6. Time consumption, FNR, and FPR.....	26
Table 7. The detail of type vectors.....	27
Table 8. Classification results for known types of malware .....	28
Table 9. Classification results for known and new types of malware .....	29





# Chapter 1 Introduction

Mobile devices were used solely for making phone calls and handling short messages in the past. However, the rapid growth on computing powers and wireless bandwidth turns mobile devices universal devices in digital life. Activities such as watching videos, playing games, checking e-mails, and online shopping now can be done anywhere and anytime with an Internet-connected mobile device. As a result, more users *migrate* from PCs to mobile devices and the number of mobile devices hence grows exponentially.

Due to its openness, Android [1] is one of the most popular operating systems adopted by modern mobile devices. Statistics collected in 2012 [2] show that there has been more than 500 million devices running the Android operating system. The popularity of Android also makes it an attractive target for attackers. From the perspective of attackers, a compromised mobile device not only can be used for launching traditional Internet attacks, but is also capable of conducting monetary attacks such as collecting sensitive personal data, sending short messages, or making phone calls. Consequently, solving security issues on mobile devices becomes important and emergent.

## Mobile Application Security

Mobile application security issues are much more critical than those on traditional PCs. Traditional PC-based malware tried their best to *spread* themselves and compromise as many hosts as possible. However, in addition to the behaviors of PC-based malware, mobile malware also attempts to steal *sensitive* data and conducts *monetary* attacks. It could read the location of a user via the built-in GPS receiver, inspect short messages, or steal contact lists. Furthermore, it is able to send short

message, making phone calls, or relaying phone calls to earn economic benefits. Compared to traditional PCs, mobile devices could be much more attractive to attackers.

We classify solutions to mobile application security issues into two categories, i.e., the *client* solutions and the *server* solutions. Client solutions provide hints and software to prevent users from being compromised by attackers. For example, the list of permission requirements and the anti-virus software [3, 4, 5] are client solutions. In contrast, server solutions are deployed on the server. Server solution can be used to check each application before an application is published online or even if an application is already online. For example, an application is available on the Apple App Store if and only if it has passed security checks done by that market. Similarly, Google has its Bouncer [6] service to search for malicious applications hidden in its market.

### **Observations and Solutions for Mobile Malware**

A number of researches have provided observations and solutions for mobile malware. In general, they can be classified into *external* based and *internal* based solutions. A lot of external based solutions focus on the uses of Android *permissions*. Statistics provided by the Stowaway project [7] showed that one-third out of 940 applications were *over-privileged*. Ryan et al. [8] also showed that most developers over-requested permissions that caused security threats. PUMA [9] used machine-learning techniques to detect malware based on permissions. Although it has a high detection rate, its false positive rate is high as well. Zhou et al. [10] *manually* analyzed essential permissions for 10 different malware families. Although it provides good understandings for the analyzed malware, it *cannot scale* efficiently for handling the explosively growing number of malware.

On the other hand, internal based solutions attempt to identify malicious behaviors by monitoring and capturing *system states* like registers and system calls. AAsandbox [11] observed suspicious applications based on *system call counts*. Crowdroid [12] classified applications into benign and malicious by *system call clusters*. Isohara et al. [13] defined signatures to detect malware by creating regular expressions for *system call names* and *file paths*. Lin et al. [14] detected five types of repackaged malware by using *system call sequences*. The above researches are able to detect or analyzed *known* malware. However, to our knowledge, none of them have been utilized to detect *unknown* malware.

In this work, we propose a *hybrid* solution that detects malicious Android applications based on both *external* observations (the requested *permissions*) and *internal* observations (the system call *sequences*). By combining the two types of behaviors, the proposed solution is able to detect *unknown* malware in an efficient manner. Our detector works in *two-phase*. In the “faster” first phase, we use the permission information to quickly identify *suspicious* applications. In the “slower” second phase, we analyze whether system call sequences generated from a suspicious application from the first phase are malicious. Furthermore, to determine whether an identified malware is a *known* or *new* type, we propose to establish *behavior vectors* from trained malware samples and then determine new types based on *similarity* between an inspected malware and the behavior vector.

The rest of this thesis is organized as follow. In Chapter 2, we give a brief survey of related works. In Chapter 3 and Chapter 4, we give the precise problem statement and describe the details of the proposed mechanism including the processing of permissions and system call sequences, respectively. Chapter 5 presents the experiment results. Finally, some concluding remarks and future work are given in Chapter 6.

## Chapter 2 Background

In this chapter, we discuss the differences between PC malware and Android malware. We also introduce several related works that have inspired the design of our proposed solution.

### 2.1 Differences between PC Malware and Android Malware

We discuss the differences of malware between PC and Android in four aspects: intention, strategies to spread, activation, and Android-specific properties.

#### Intention

In this subsection, we describe the destructive activities for both PC and Android malware. The intention of attackers changes from time to time. In the past, attackers like to show their ability to compromise a large number of computers. As a result, PC malware attempts to reduce the system performance and modify or delete system files. In general, it conducts to *paralyze* the system. In contrast, Android malware focuses on *privacy* and *monetary* attacks. Android malware attempts to *steal* personal information such as user location and even *eavesdrop* the phone calls by sound recording. In addition, it could launch monetary attacks by exploiting *paid* services such as sending short messages, making phone calls, and visiting on-line advertisements.

#### Strategy to Spread

Modern attackers disseminate PC malware via Internet Web, E-mail, as well as peer-to-peer file sharing. A user's computer could be infected if he or she browses a malicious webpage that attackers had injected malicious codes or opens an application downloaded from web, peer-to-peer network, or E-mail. Attackers make the use of

*various* Internet media to spread malware rapidly. In contrast, Android malware is spread through *application markets*. For example, the *official* Android market developed by Google is a digital application distribution platform that permits a user to download or upload applications. However, it lacks a rigorous detection mechanism for malicious applications. Therefore, attackers are able to exploit the weakness to spread malware by *repackaging* popular applications with malicious code to trap users. Similar cases also happen on *third-party* markets.

### **Activation of Malware**

A PC user and an Android user could be trapped into launching a malicious application directly. However, attackers have much more *choices* to activate malicious applications on PC. An attacker is able to request a user to run browser *add-ons* when the user visits a compromised website. Alternatively, modern PC malware also utilizes vulnerabilities of *in-browser* applications such as interpreters, virtual machines, flash players, and document viewers. By injecting itself into in-browser applications, a PC malware can be activated immediately when a vulnerable in-browser application is activated.

### **Android-specific Properties**

In addition to the above-mentioned differences, Android malware has some specific properties. On Android, a user may exploit permissions to determine whether an application is malicious. For instance, if phone-call permission is requested by a game application, the user may refuse to install that application. Another property is that Android malware is usually *embedded* into well-known or popular applications. There are not too many *standalone* malware. Finally, most Android malware is *passively* downloaded by a user instead of actively intruding into a user's device.

## 2.2 Related Works

Table 1 shows several existing solutions to detect Android malware. Kirin [15] used permission security rules to mitigate malware by voice, location, or short messages. They utilized a set of security rules to judge whether an application requests some dangerous *combinations* of permissions. PUMA [9] adopted machine-learning approaches including simple logistic, naïve Bayes, J48, and random tree to classify applications into benign or malicious applications based on permissions. The above two solutions are simple and efficient because they only analyzed the manifest file of an application. However, a malicious application is possible to *evade* the detection of Kirin and PUMA and consequently they have high *false positive* rates.

Zhou et al. [10] obtained the essential permissions and behaviors by *manually* analyzing 10 different malware families. They chose the permissions to filter out benign applications quickly and detected remaining applications by behavioral footprint *matching*. However, it is not scalable because the solution cannot be *automated*.

Table 1. Related works to detect malicious Android applications

Category	Solution	Behaviors	Training	Detection	Cons
Static	<i>Kirin</i> [15]	Permissions	Security Rules	Matching	High FP
	<i>PUMA</i> [9]	Permissions	Machine-learning	Classification	High FP
	<i>Zhou et al.</i> [10]	Permissions Bytecode Structural Layout	Essential Permissions Behavioral Footprint	Matching	Require manual analysis
Dynamic	<i>Crowdroid</i> [12]	System Call Count	Vectors	Clustering	Require a lot of user experience
	<i>Ishara</i> [13]	System Call Name and Parameter	Regular Expression	Matching	Limited types of malware
	<i>Lin et al.</i> [14]	System Call Sequences	Sequences	Matching	Can be easily evaded / Not efficient
Mix	<i>The Proposed Solution</i>	Permissions System Call Sequences	Probability Sequences	Threshold Matching	

Crowdroid [12] monitored system calls invoked by an application and utilized a

*clustering* algorithm to judge whether the application is benign or malicious. However, it has to collect a lot of the user experiences for the same application. Otherwise, it could make a lot false positives. The solution only detects *anomalous* behaviors of analyzed applications. Isohara et al. [13] defined three categories of threats including information leaking, jail-breaking, and destructive application detection. They generated signatures by applying a set of regular expression rules to the name of system calls or file paths. A malicious activity in these three categories is then detected by *signature matching*. However, the system cannot detect the malicious activities other than the three categories of threats. Lin et al. [14] extracted *longest common substrings* (LCS) of system calls for malicious applications of the same type and discriminated malicious behaviors from benign ones based on probabilities derived from the Bayes model. They then detect repackaged malware with the obtained LCS. However, the solution is not efficient because it has to run all applications on emulators or real devices. In addition, the proposed *Layering Multi-Thread Comparison* mechanism provides a door for malware to evade the detection.

To achieve high detection performance and accuracy, we propose a three-phase behavior-based solution, where the first two phases detect malicious applications and the last phase classifies into known or new types of malware.

## Chapter 3 Problem Statement

Here we first describe the notations and then give the problem description.

### 3.1 Notations

Table 2 defines the notations used in our approach. We collect a set of benign applications  $BP = \{bp_i, 1 \leq i \leq A_B\}$  and a set of malware  $MP = \{mp_j, 1 \leq j \leq A_M\}$  for training. For *permission* based detection, we have a set  $P$ ,  $P = \{p_l, 1 \leq l \leq N\}$ , containing all built-in Android permissions. We calculate the probability of being malicious for each of the  $N$  permissions and store in another set  $PP$ . For *system call* based detection, we obtain a set of benign behaviors  $BBS = \{bb_n, 1 \leq n \leq S_B\}$ , a set of suspicious behaviors  $SBS = \{sb_o, 1 \leq o \leq S_S\}$ , and a set of malicious behaviors  $MBS = \{mb_p, 1 \leq p \leq S_M\}$ . For further classification, a *type vector* is used to denote what malicious behaviors a malware has. We collect type vectors of  $V$  trained malware samples to build a set of known type vectors  $TV = \{tv_q, 1 \leq q \leq V\}$  where each  $tv_q$  is a bit-vector  $\langle mb_1, mb_2, \dots, mb_{S_M} \rangle$  to indicate the existence of malicious behavior for sample  $q$ . The proposed solution is evaluated with a given set of inspected applications  $IP = \{ip_k, 1 \leq k \leq A_I\}$ , whose behaviors are denoted as  $IBS = \{ib_r, 1 \leq r \leq S_I\}$ .

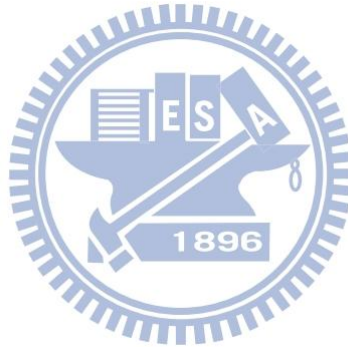
### 3.2 Problem Description

Given a set of benign applications  $BP$ , a set of malware  $MP$ , and a set of applications to be inspected  $IP$ , design an approach to detect if an inspected application is malicious or not based on a set of permissions probability  $PP$  and a set of malicious behaviors  $MBS$ .  $MBS$  is obtained from  $BP$  and  $MP$ . Finally, we classify a detected malicious application into a known type or a new type based on type vectors  $TV$  defined from  $MP$ .



Table 2. Definition of notations

Categories	Notations	Descriptions
Application	$BP$	A set of $A_B$ benign applications for training
	$MP$	A set of $A_M$ malware for training
	$IP$	A set of $A_I$ applications to be inspected
Permission	$P$	A set of $N$ permissions
	$PP$	A set of $N$ permissions' probability of being malicious
Behavior	$BBS$	A set of $S_B$ benign behaviors
	$SBS$	A set of $S_S$ suspicious behaviors
	$MBS$	A set of $S_M$ malicious behaviors
	$IBS$	A set of $S_I$ inspected behaviors
Type	$TV$	A set of $V$ type vectors



## Chapter 4 Three-phase Behavioral Detection and Classification

In this chapter, we describe the process of three-phase behavioral detection and classification (TPBDC) based on permissions and system call sequences. In Section 4.1, we give an overview of permissions and system call sequences. The details of our two detection phases and one classification phase are introduced in Section 4.2, Section 4.3, and Section 4.4, respectively. We discuss implementation issues in Section 4.5.

### 4.1 Overview of Three-phase Behavioral Detection and Classification

To achieve high detection performance and accuracy, we propose a three-phase approach. We choose to check permissions in the first phase so that the *number* of applications passed to the second phase can be reduced. To have better accuracy, we check the system call sequence to reduce *false positive* rates in the second phase.

Figure 1 shows the overview of our three phases: the permission-based detection (PBD) phase, the system call-based detection (SBD) phase, and the behavior-based classification (BBC) phase. In the PBD phase, we extract permissions from *BP*, *MP*, and *IP*. *PP* is obtained from *BP* and *MP* and we then utilize *PP* to judge whether *ip<sub>k</sub>* is *suspicious* and only a suspicious application is passed to the next phase.

In the SBD phase, we record system calls of *BP* and *MP* for training. We train a set of system call sequences from all the applications and then utilize the trained system call sequences to obtain *MBS*. For detection, we record system calls of *ip<sub>k</sub>* and then *match* with *MBS* to detect whether *ip<sub>k</sub>* is malicious. Note that only applications *not* filtered out in the previous phase are processed by this phase. In the BBC phase, we exploit the behaviors of malware to train *TV* and then utilize *TV* to classify *ip<sub>k</sub>* into a known type or a new type depending on whether its behaviors are in *TV* or not.

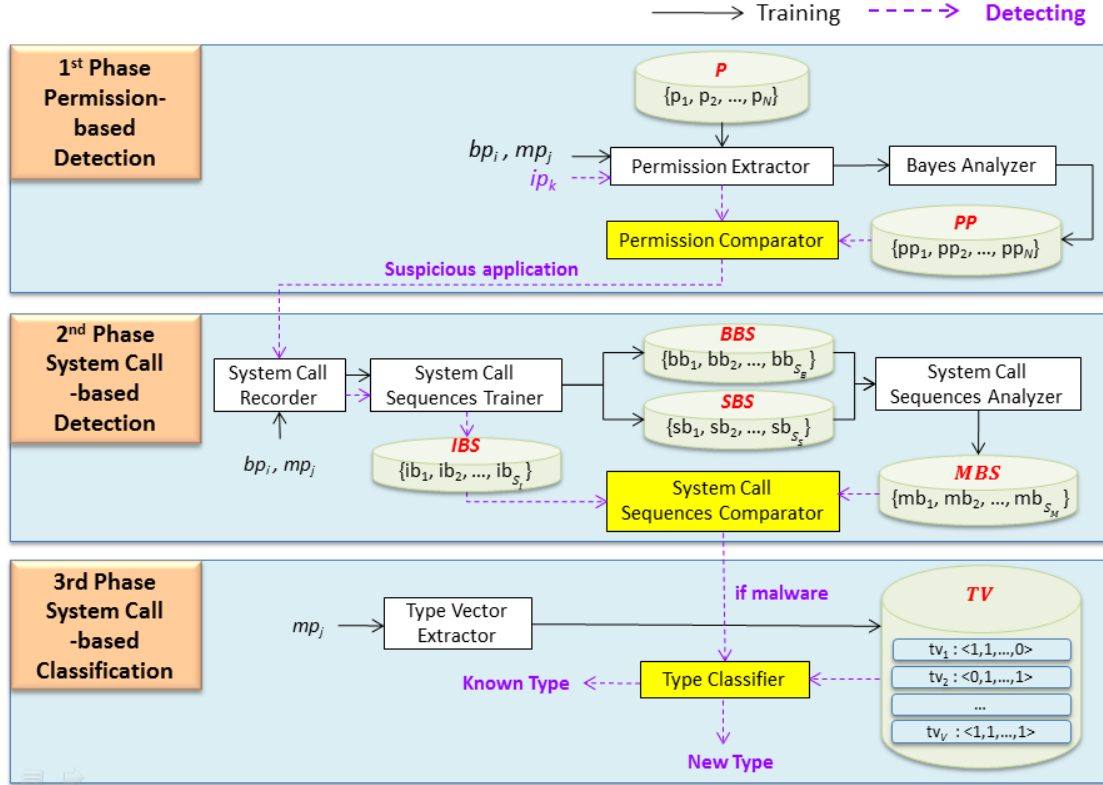


Figure 1. Overview of the proposed solution

## 4.2 Permission-based Detection (PBD) Phase

In this section, we introduce the PBD phase, which is composed of a permission extractor, a Bayes analyzer, and a permission comparator.

### Permission Extractor

The permission extractor is used to retrieve built-in permissions requested by each application. Android has 139 built-in permissions. We extract permissions from  $BP$  and  $MP$  for training and then extract permissions from  $IP$  for detecting.

### Bayes Analyzer

We evaluate the probability of being malicious for each built-in permission. Requested permissions are retrieved from both  $BP$  and  $MP$  and then probabilities are obtained using the Bayes theorem. To simplify the evaluation, we only count Android's built-in permissions. The formula to evaluate the probabilities is

$$P(M|p_l) = \frac{P(p_l|M) * P(M)}{P(p_l|M) * P(M) + P(p_l|B) * P(B)} , \quad (1)$$

where  $p_l$  is one of 139 built-in permissions to be evaluated.  $P(B)$  denotes the ratio of  $BP$  while  $P(M)$  denotes the ratio for  $MP$ .  $P(p_l|B)$  and  $P(p_l|M)$  represent the probability that  $p_l$  is requested by  $BP$  and  $MP$ , respectively. We then get the probability  $P(M|p_l)$ , which indicates the probability to be malicious on the condition that  $ip_k$  requested permission  $p_l$ . The permission probability set  $PP$  is obtained by using formula (1) for all the 139 built-in permissions.

### Permission Comparator

We also extract the requested permissions of  $ip_k$  and calculate the *product* of probabilities of all requested permissions using probabilities from  $PP$ . If the product is larger than the upper bound,  $ip_k$  will be judged as malware. If the product is lower than the lower bound,  $ip_k$  will be judged as benign. If the product of probabilities of  $ip_k$  is between the upper bound and the lower bound, it is marked as a suspicious application and passed to the next phase.

### 4.3 System Call-based Detection (SBD) Phase

The SBD phase is composed of four components. They are system call recorder, system call sequence trainer, system call sequence analyzer, and system call sequence comparator, as shown in Figure 2.

#### System Call Recorder

The system call recorder records the system calls triggered by applications. First, we install  $bp_i$ ,  $mp_j$ , or  $ip_k$  into the Android 2.1 emulator and launch the application. After it has been launched, we emulate several system events such as rebooting, receiving short messages, and answering phone calls. We record system calls of the application for a period of time.

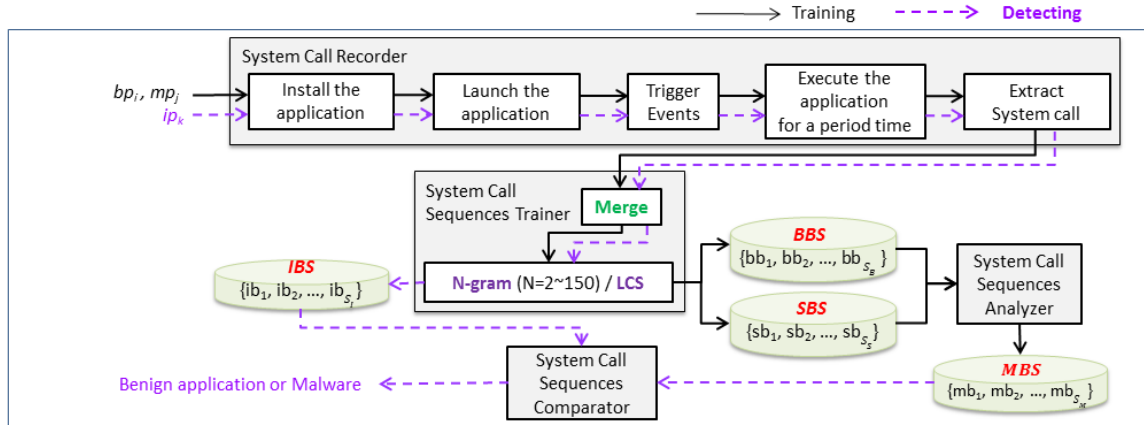


Figure 2. Procedure flowchart of SBD

### System Call Sequence Trainer

The goal of the system call sequence trainer is to generate *BBS*, *SBS*, and *IBS* using the N-gram and the LCS algorithm. We consolidate successive system calls before computing system call sequences because a system call could be issued repeatedly in loops. For instance, a raw system calls sequence of “open, read, read, read, close” would become “open, read, close”

After consolidating successive system calls, *BBS*, *SBS*, and *IBS* are generated by either the N-gram or the LCS algorithm. The purpose of system call sequence trainer is to find out *common* sub-sequences. Since a common malicious behavior is the great resemblance of malware, the system call sequences recorded from the malware should share the system call subsequences considerably. The system call sequences for *BP* are stored in *BBS*, the common system call subsequences for *MP* are stored in *SBS*, and the system call sequences for *IP* are stored in *IBS*.

### System Call Sequence Analyzer

The system call sequence analyzer finds out malicious system call sequences. We obtain *MBS* from *SBS* and *BBS* in this module. We *filter out* a system call sequence if it appears in *both* *SBS* and *BBS*. After the filtering, the malicious behavior

set ( $MBS$ ), which contains only system call sequences appeared in  $MP$ , is obtained.

### System Call Sequence Comparator

To inspect  $ip_k$ , the system call recorder and the system call sequence trainer are used to record system calls and generate system call sub-sequences. Figure 3 shows how the system call sequence comparator compares the system call sequences  $ib_r$  of  $ip_k$  against all malicious behaviors listed in  $MBS$ . The  $ip_k$  is classified as malicious if a malicious behavior is matched.

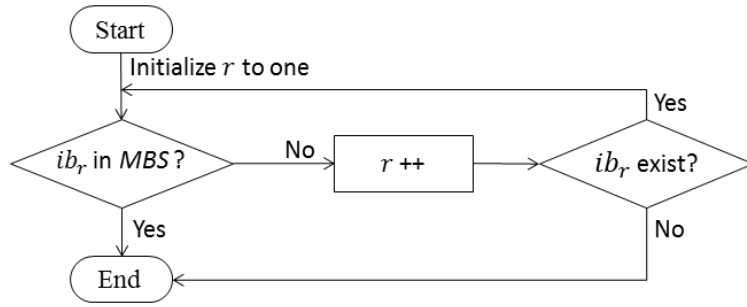


Figure 3. Procedure of system call sequence comparator

## 4.4 Behavior-based Classification (BBC) Phase

If a malicious application is detected, we propose another technique to classify the detected malware into a known type or a new type of malware. In this subsection, we explain the detailed design of BBC, which is composed of a type vector extractor and a type classifier.

### Type Vector Extractor

We utilize a *bit vector* to denote what behaviors malware has. Suppose all identified behaviors are indexed from 1 to 500 and malware has the first, the third, and the 499<sup>th</sup> behaviors. The corresponding bit vector  $tv_I$  would be  $\{1, 0, 1, 0, \dots, 1, 0\}$ . We build bit vectors for  $MP$  and use the bit vectors to detect whether  $ip_k$  is a known type or a new type. All the obtained bit vectors for  $MP$  are stored in a set  $TV$ .

## Type Classifier

Figure 4 shows the procedure of the type classifier. We construct a bit vector  $ipv$  for  $ip_k$  and calculate the similarity between  $ipv$  and all bit vectors of  $MP$  by *cosine similarity* [16]. The cosine similarity is obtained by

$$\frac{\sum_{i=1}^{SM} tv_i * ipv_i}{\sqrt{\sum_{i=1}^{SM} (tv_i)^2} * \sqrt{\sum_{i=1}^{SM} (ipv_i)^2}}, \quad (2)$$

where  $tv$  is one of available  $MP$  bit vectors in  $TV$  and  $ipv$  is the bit vector of  $ip_k$ . We define a threshold as the *lower* bound for the cosine similarity to classify  $ipv$  into a known type or a new type. If the similarity is greater than the threshold, we classify  $ipv$  into the same type of the bit vector having the *maximum* cosine similarity value.

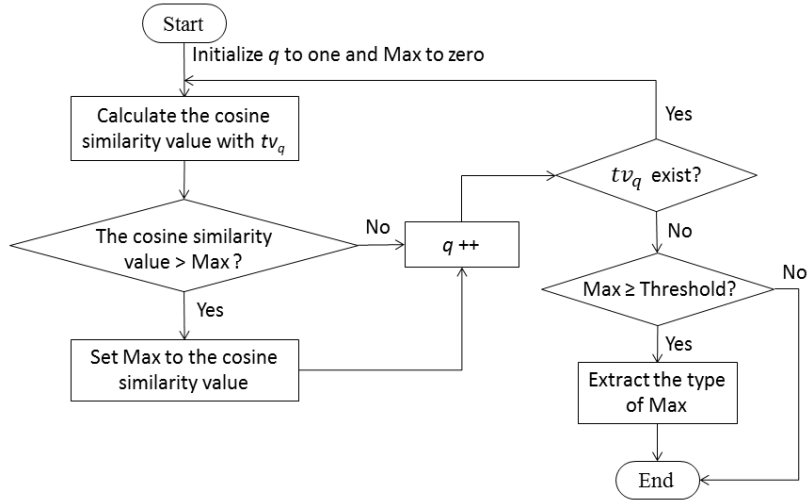


Figure 4. Procedure of type classifier

## 4.5 Implementation

We have developed tools to automatically retrieve permissions and system call sequences of Android applications.

### Permission Analyzer

Figure 5 shows the procedure of the permission analyzer. Because an APK file is basically a ZIP archive file with an APK file extension, we decompress an application

to retrieve permissions of the application by apktool [17]. After decompressing, we get assets, resources, application’s source codes (via disassemble), and the manifest file. To retrieve permissions, we only parse the manifest file because a developer declares requested permissions in this file.



Figure 5. Procedure of permissions analyzer

### System Call Recorder

Figure 6 shows the procedure of the system call recorder. In order to record the system calls of an application, we need to modify the system ramdisk.img and install strace [18] into the emulator. First, we decompress the ramdisk.img, install the strace tool into the system, and modify the init.rc file to launch the strace tool. The strace tool is placed in the /data directory. For the init.rc file, we insert the strace command “/data/strace -F -ff -tt -o /data/tracefile/zygote” into this file as shown in Figure 7. With the above modifications, strace is launched to record system calls right after the emulator boots. The output of the strace tool is placed in /data/tracefile/zygote file.

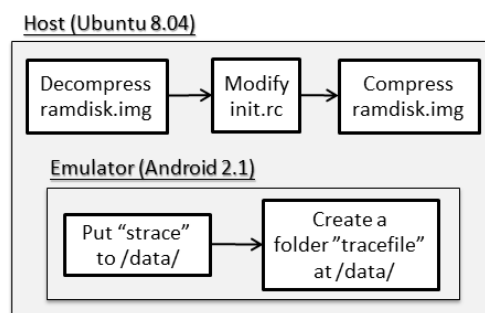


Figure 6. Procedure of system call recorder

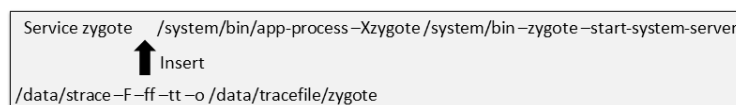


Figure 7. Detailed modification of the init.rc file



## Chapter 5 Evaluation

To evaluate the effectiveness of the proposed solution, we conduct experiments with diverse types of repackaged applications. We describe the experiment environment and the number of trained and inspected applications in Section 5.1. We then present the effectiveness in detection and classification based on permissions and system call sequences in Section 5.2.

### 5.1 Experiment Environment

We introduce the experiment environment from two aspects: The training and the detection and then summarize the samples used in the experiments.

#### Training

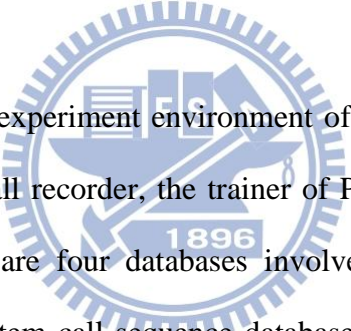


Figure 8 illustrates the experiment environment of training which includes four programs, i.e., the system call recorder, the trainer of PBD, the trainer of SBD, and the trainer of BBC. There are four databases involved, including the permission probability database, the system call sequence database, the malicious behaviors set database, and the type vector database.

To obtain the permissions probability, we parse all benign applications and malware first and calculate the permission probabilities using formula (1). We utilize the system call recorder implemented in the emulator to record system call sequences of training applications for 3 minutes, as suggested by [14], and then use the SBD trainer to acquire the malicious behaviors set. Finally, the BBC obtains the type vectors from behaviors of training malware.

#### Detection

Figure 8 also illustrates the experiment environment of detection with the same four programs. There are three involved databases including the permission

probability database, the malicious behavior set database, and the type vector database.

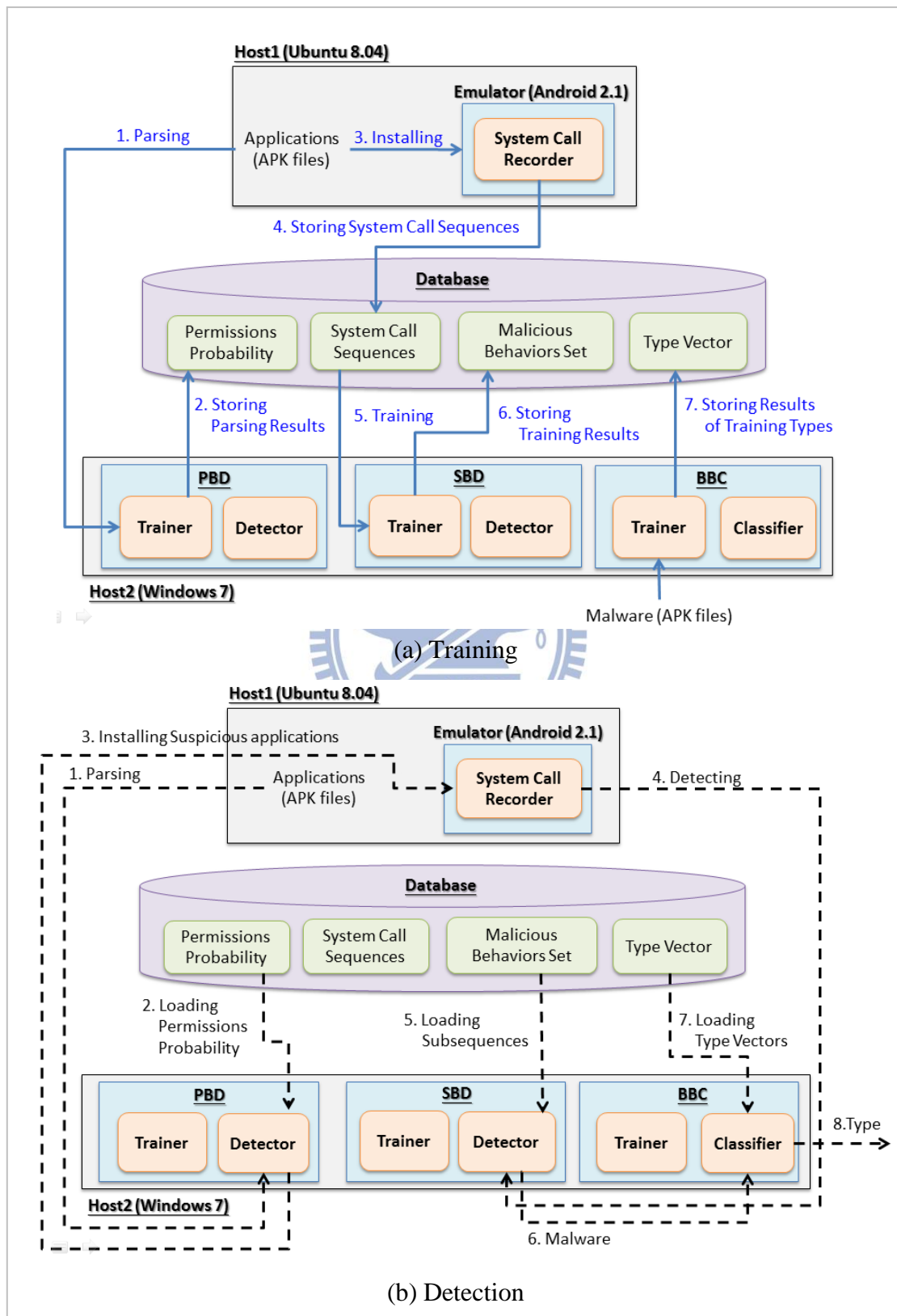


Figure 8. Experiment environment

In order to judge whether an inspected application is benign or suspicious, we parse permissions of the inspected application first and then load previously evaluated permission probabilities for the detector of PBD. If PBD alerts, the inspected application is installed to the emulator to further record system call sequences also for 3 minutes by the system call recorder. The SBD will alert if the system call sequences of the inspected application matches one of available malicious behaviors. We classify the inspected application detected by SBD into a known type or a new type based on the type vector.

### Samples

To conduct the experiments, we prepare 1198 sample applications. Table 3 shows the 1198 applications composed of 933 benign applications and 265 malware. We use 863 applications (700 benign applications and 163 malware) for training and 335 applications (233 benign applications and 102 malware) for detection. Table 4 shows that we adopt the malware types for known types and new types.

The benign applications are collected from third-party markets and malware is collected from Zhou et al. [19] which uses manual or automated crawling from a variety of Android Markets. We utilize several anti-virus tools [20] to ensure the benign applications are virus-free.

Table 3. Number of training samples and detecting samples

Category	Benign Samples	Malicious Samples
Training	700	163
Detection	233	102

Table 4. List of malware types

Category	Malware Type
Known Type	Adrd, AnserverBot, Asroot, BaseBridge, Bgserv, Geimini, GingerMaster, GoneSixty, jSMShider, Kmin, Lightdd, Plankton, RogueSPPush, SndApps, YZHC, zHash, Zsone
New Type	DroidKungFu, GGTracker, GoldDream, PJAPPS.G, Smspacem

## 5.2 Experimental Results

We discuss the performance numbers from seven issues: impact on permissions, impact of the value N for N-gram on system call sequences, impact of the number of malicious behaviors for LCS on system call sequences, detection performance vs. time consumption, effectiveness on the order of applied detection phases, performance comparison for one-phase and two-phase detectors, and evaluation for type vector based classification.

### Impact on Permissions

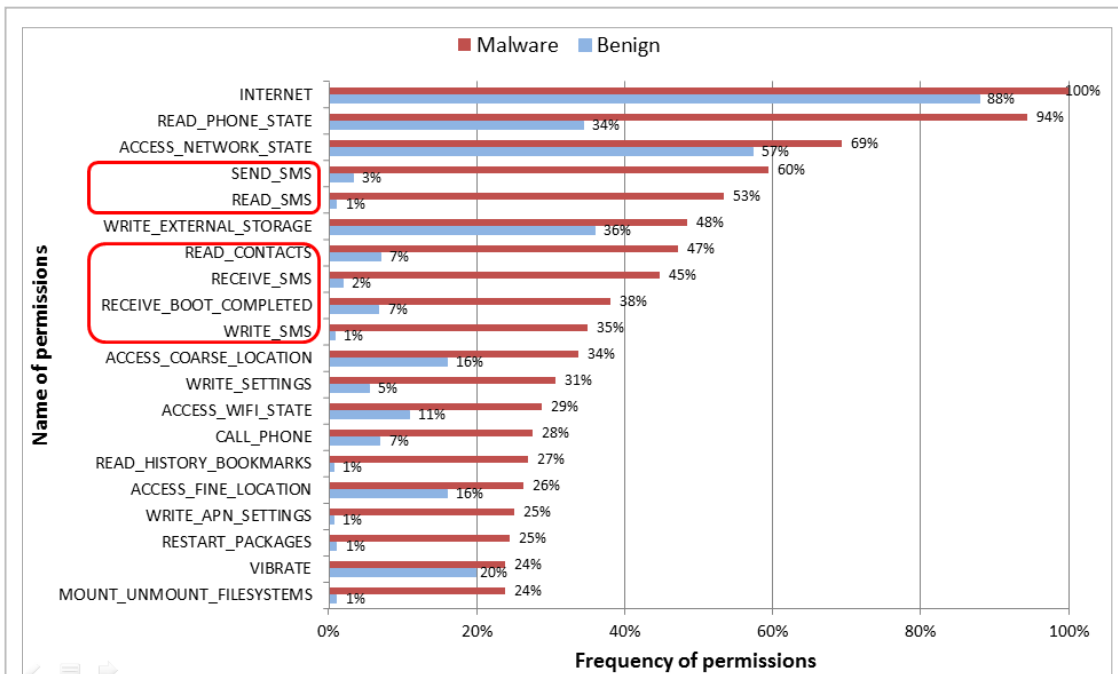
We first evaluate the PBD. First we made preliminary statistics on the number of permissions appearance in benign applications and malware. Figure 9(a) and 9(b) show the distribution of permissions for benign and malicious applications, respectively. We can find that the popular permissions requested by benign and malicious applications are different. In Figure 9 (a), we also observe that some permissions such as SEND\_SMS, READ\_SMS, READ\_CONTACTS, RECEIVE\_SMS, WRITE\_SMS, and RECEIVE\_BOOT\_COMPLETED has much higher frequency being requested by malware than benign applications. Figure 10(a) and 10(b) show the permissions requested by benign applications and malware, respectively.

We calculate the probabilities of 139 permissions by the Bayes theorem. To judge the inspected application, we calculate the product of permission probabilities. If the product is in the predefined ranges, the application is judged as suspicious. Figure 11 shows the accuracy of different thresholds. There are more benign applications and malware filtered out if the threshold is increased. Since we use this phase to reduce the number of applications, the upper bound of 0.9 and the lower bound of 0.1 would be a good choice based on our experiments.

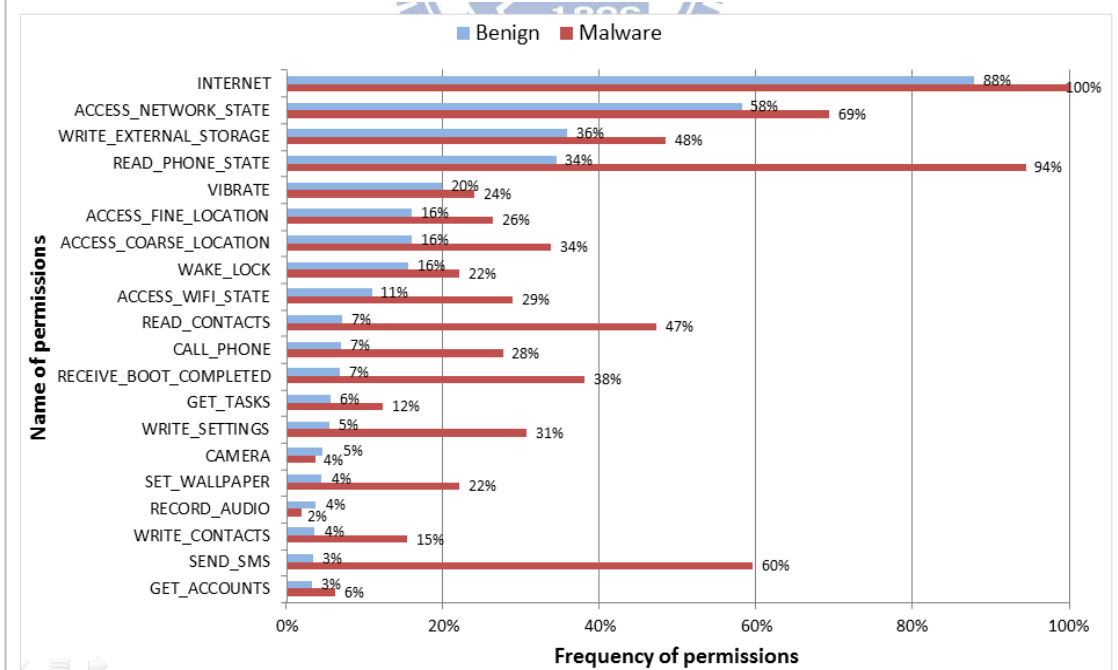
## Impact of the value N for N-gram on System Call Sequences

We evaluate SBD with the N-gram and the LCS algorithm in this experiment.

The length of system call sequences means how many system calls every system call



(a) The top 20 of requested permissions by malware



(b) The top 20 of requested permissions by benign applications

Figure 9. Distribution of the permissions requested by benign and malicious applications

sequence contains. For LCS, the length of system call sub-sequences is so dynamic that we do not have to predefine the length. However, the length of system call sequences for N-gram can be configured so we vary the value N to see its effectiveness.

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest android:versionCode="1" android:versionName="1.0" android:installLocation="auto"
3   xmlns:android="http://schemas.android.com/apk/res/android">
4   <uses-sdk android:minSdkVersion="7" />
5   <uses-permission android:name="android.permission.INTERNET" />
6   <uses-permission android:name="android.permission.READ_PHONE_STATE" />
7   <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
8   <application android:label="@string/app_name" android:icon="@drawable/icon" android:na
9     <activity android:theme="@android:style/Theme.NoTitleBar" android:label="@string/a
10      <intent-filter>
11        <action android:name="android.intent.action.MAIN" />
12        <category android:name="android.intent.category.LAUNCHER" />
13      </intent-filter>
14    </activity>
15    <activity android:theme="@android:style/Theme.NoTitleBar" android:label="Favorites
16    <activity android:theme="@android:style/Theme.NoTitleBar" android:label="USA Newsp
17    <activity android:theme="@android:style/Theme.Translucent.NoTitleBar" android:name
18    <activity android:theme="@android:style/Theme.NoTitleBar" android:label="Favoritos
19    <activity android:theme="@android:style/Theme.NoTitleBar.Fullscreen" android:name=

```

(a) The requested permissions by benign applications

```

16 </receiver>
17 <receiver android:name="com.xxx.yyy.MyAlarmReceiver">
18   <intent-filter>
19     <action android:name="com.lz.myservicestart" />
20   </intent-filter>
21 </receiver>
22 <service android:name="com.xxx.yyy.MyService" android:enabled="true" />
23 </application>
24 <uses-sdk android:minSdkVersion="3" />
25 <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
26 <uses-permission android:name="android.permission.INTERNET" />
27 <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
28 <uses-permission android:name="android.permission.READ_PHONE_STATE" />
29 <uses-permission android:name="android.permission.SEND_SMS" />
30 <uses-permission android:name="android.permission.RECEIVE_SMS" />
31 <uses-permission android:name="android.permission.READ_SMS" />
32 <uses-permission android:name="android.permission.WRITE_SMS" />
33 </manifest>
34

```

(b) The requested permissions by malware

Figure 10. The permissions requested by benign applications and malware

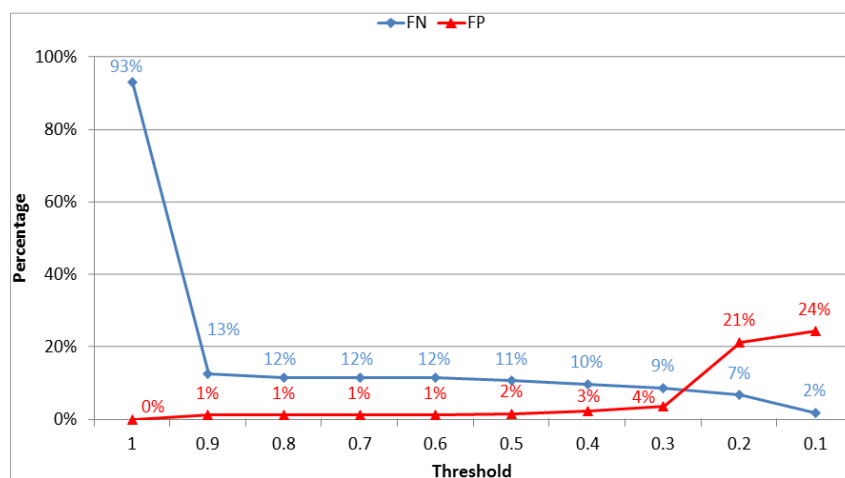


Figure 11. Performance for PBD

The value  $N$  for  $N$ -gram means the unit length of system call sequence retrieved from all system call sequences. Different lengths could lead to different performance. A small  $N$  would filter out malicious system call sequences. If a system provides only 200 different system calls, a value  $N$  of two would have only 19900 combinations of sequences and therefore it can be easily filtered out by a relatively large number of benign system call sequences. In contrast, a large  $N$  would preserve too many benign behaviors within a malicious system sequence. Consequently, it is important to choose a good value for  $N$ .

Figure 12 shows the detection performance with various  $N$ . We divide Figure 12 into three areas by  $N$ . In the first area ( $N$  ranges from 2 to 4), the FN curve is descending but the FP curve is ascending. In the second area ( $N$  ranges from 5 to 15), the FN curve reaches 0% and the FP curve is kept smooth. In the third area ( $N$  ranges from 20 to 150), the FP curve is descending but the FN curve is ascending.

The higher FN rate in the first area is because  $N$  is so small that significant system call sequences are filtered out. In contrast, the higher FN rates in the third area are because  $N$  is too large so that the system call sequences are mixed with benign system call sequence. If we want a lower FN rate, a good value of  $N$  would be in the second area. Based on the experiment, we choose a value of 15 for  $N$ . Although we get a higher FP rate when  $N$  is 15, we can reduce the FP rate with the help of PBO.

### **Impact of the number of malicious behaviors for LCS on System Call Sequences**

Figure 13 presents the percentage of samples versus the number of malicious behaviors. We can observe that some malicious behaviors are also performed by benign applications and there are less FPs if the number of malicious behaviors gets increasing. If LCS-based SBD works with PBD, the number of malicious behaviors with 1 would be a good choice. If LCS-based SBD works alone, the number of

malicious behaviors with 2 would be much suitable.

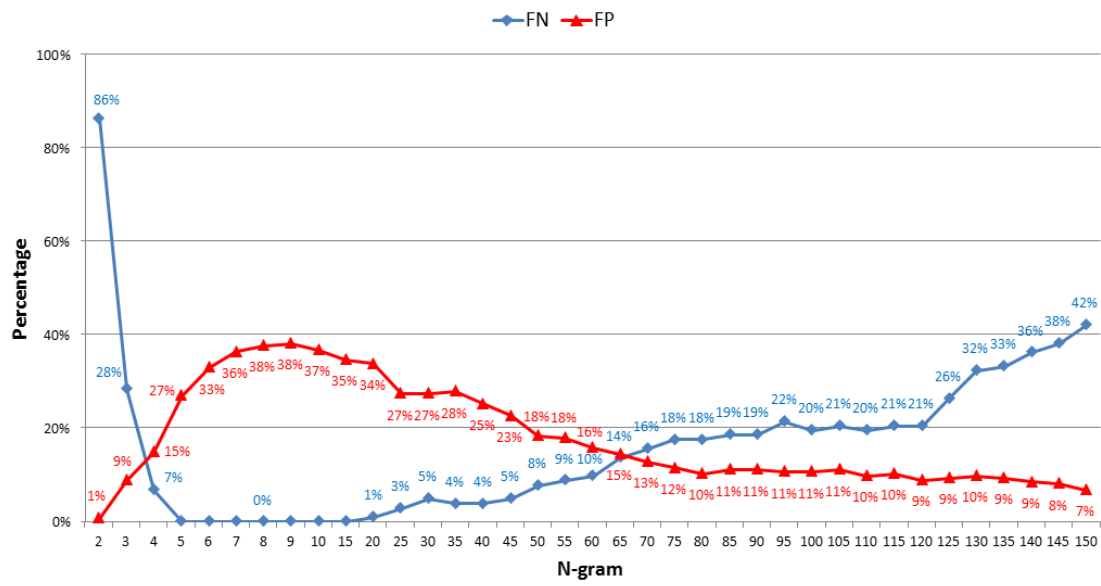


Figure 12. Detection performances for system call sequences with various N

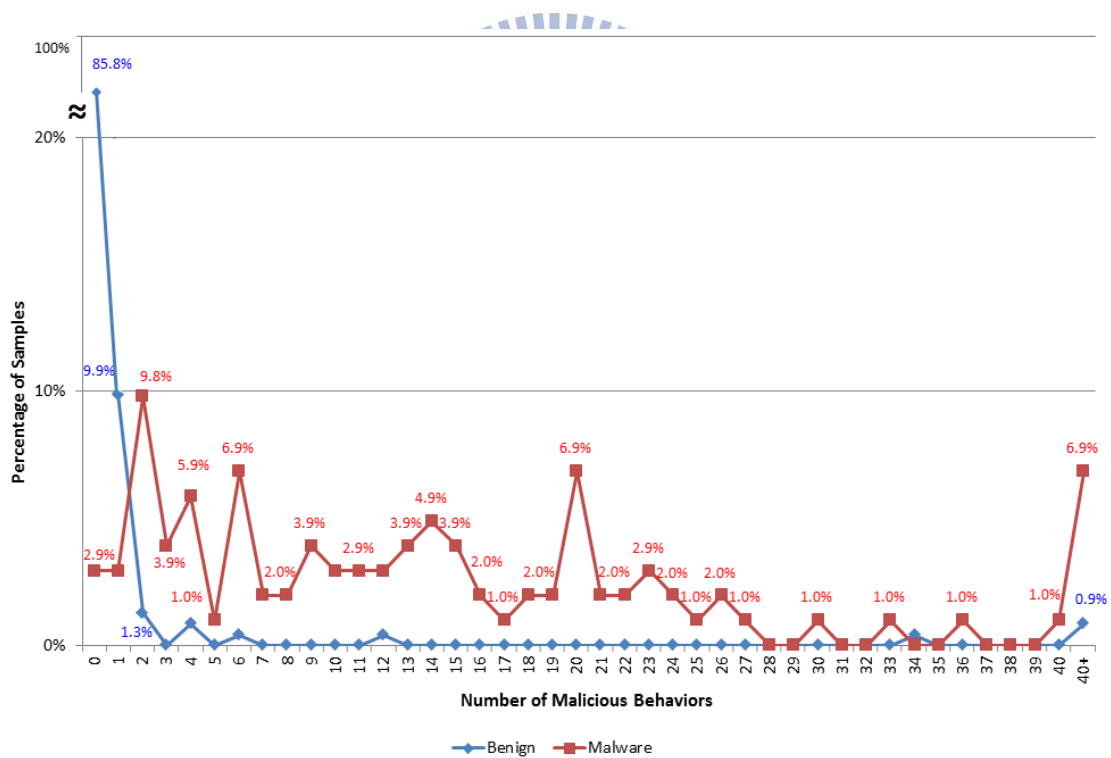


Figure 13. Distribution for the number of malicious behaviors

### Detection Performance vs. Time Consumption

Malware detection mechanisms can be categorized into static analysis or dynamic analysis techniques. Static analysis is simple and efficient but dynamic



analysis is complex and time-consuming. In our approach, PBD is static and SBD is dynamic. Here we discuss the tradeoff between accuracy and time requirement. All the experiments are conducted with a machine equipped with an Intel Core i3 3.1GHz CPU and 16GB of RAM running on a 64-bit Windows 7 operating system. The summary of the experiment results is shown in Table 5.

We first compare accuracy and time for LCS-based and N-gram-based SBD. Although it requires more time, LCS-based detector gets a better accuracy. We also compare accuracy and time between PBD and LCS-based SBD. Although PBD runs faster, it gets a poor accuracy.

Table 5. Detection performance vs. time consumption

Category	Algorithm	FN	FP	Accuracy	Time Consumption (sec/application)
PBD	Bayes Probability	2%	24%	87%	2.57
SBD	LCS	3%	14%	91.5%	601.38
	N-gram	0%	35%	82.5%	600.87

### Effectiveness on the Order of Applied Detection Phases

We also consider the order of applying different detection phases, i.e. PBD and SBD. The measured detection time and detection performance is shown in Table 6.

For the time consumption, if the PBD is applied first, it takes 599 seconds and 262 seconds to analyze permissions for all benign applications and malware, and then the LCS-based SBD spends 32,228 seconds and 6,747 seconds to analyze system call sequences for the remaining 23% of benign applications and 11% of malware. In contrast, if we swap the order, the LCS-based SBD takes 140,122 seconds and 61,341 seconds to analyze system call sequences for all benign applications and malware, and then the PBD spends 84 seconds and 8 seconds to analyze permissions for the remaining 14% of benign applications and 3% of malware. If we want a lower FNR and a lower FPR, running the PBD first would be a better choice as PBD consumes

much less time than SBD. Based on the experiments, we choose PBD as the first phase detector and SBD as the second phase detector.

Table 6. Time consumption, FNR, and FPR

Strategy		Procedure		Time Consumptions			FNR for 1 <sup>st</sup> Phase	FPR for 1 <sup>st</sup> Phase
		1 <sup>st</sup> Phase	2 <sup>nd</sup> Phase	1 <sup>st</sup> Phase	2 <sup>nd</sup> Phase	Total		
PBD → SBD	Benign	100%	23%	599s	32,228s	32,827s	-	1%
	Malware	100%	11%	262s	6,747s	7,009s	2%	-
SBD → PBD	Benign	100%	14%	140,122s	84s	140,206s	-	14%
	Malware	100%	3%	61,341s	8s	61,349s	3%	-

### Performance Comparison for One-phase and Two-phase Detectors

From the above experiment, we know that PBD is able to filter out 76% of benign applications with the lower bound of 0.1 and filter out 87% of malware with the upper bound of 0.9 and LCS-based SBD has a better accuracy than PBD. Figure 14 compares the accuracy of one-phase detectors and two-phase detectors (PBD first and then SBD). The two-phase detectors have a lower FPR than one-phase detectors. We can see that although one-phase detectors could have poor performance, the combined detectors always have a good performance. This also shows that PBD and SBD complement each other.

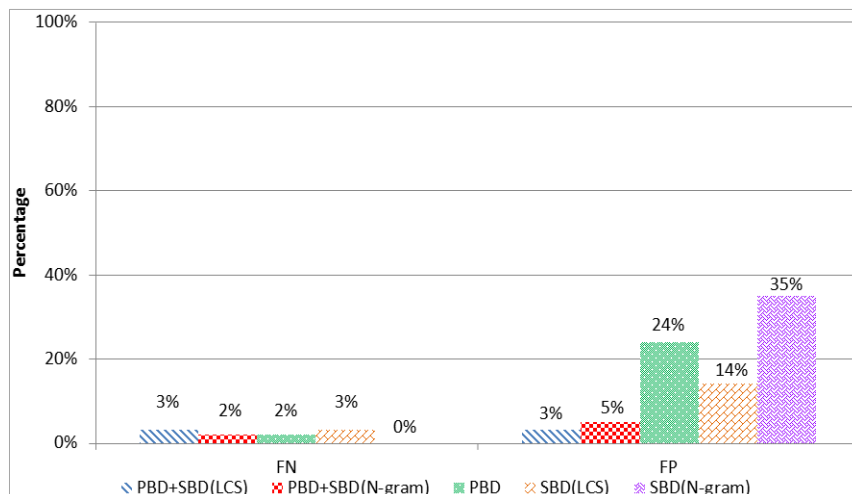


Figure 14. Accuracy comparison for one-phase and two-phase detectors

Now we examine the reasons which cause false negatives for two-phase detectors. In PBD, we filtered out 2% of malware because some malicious

applications request only a few non-critical permissions. In SBD, we missed 1% of malware because some malicious behaviors would be triggered only after we agree the update option. We also discuss the reasons that cause false positives for two-phase detectors. Since most Android malware are repackaged applications, the recorded system call sequences are *mixed* with both benign and malicious *behaviors*. It may cause false positives if we do not completely filter out benign behaviors. In addition, the *noise* from Dalvik VM that runs the applications on Android devices may incur both false positives and false negatives. This is because the system call sequences originated from Dalvik VM itself are recorded as well and it is not able to tell the real origin of system calls.

### Evaluation for Type Vector Based Classification

Table 7 shows the details of type vectors. For system call sequences, we get 149 system call sequence vectors to denote 17 types, and the length of system call sequence vectors is 1460. For permissions, we get 68 permission vectors to denote 17 types, and the length of permission vectors is 139. If we mix system call sequences and permissions, we get 156 mix vectors to denote 17 types, and the length of mix vectors is 1599.

Table 7. The detail of type vectors

Category	Number of Types	Number of Type Vectors	Length of Type Vectors
System Call Sequence Vectors	17	149	1460
Permission Vectors		68	139
Mix Vectors		156	1599

Finally, we evaluate the performance of BBC which recognizes the type (known or new) of a detected malware. We classify a detected malware based on permissions, system call sequences, or mix. To show that type vector is good at classifying type of malware, we attempt to classify all identified malicious applications into a *malware*

*type*. We use the PBD detector followed by the LCS-based SBD detector. The classification result for known types using LCS-based type vectors, permission-based type vectors, and mix-based type vectors is shown in Table 8. We show that 93%, 99%, and 96% of malicious applications can be classified into a correct type based on LCS-based type vectors, permissions-based type vectors, and mix-based type vectors, respectively. It shows that permission-based type vectors can be a better choice than LCS-based type vectors or mix vectors.

Table 8. Classification results for known types of malware

Malware Type	Category	Number of Malware	Classification Result	Percentage
Known Type	System Call Sequence Vectors	99	Correct	93%
			Incorrect	7%
	Permission Vectors	99	Correct	99%
			Incorrect	1%
	Mix Vectors	99	Correct	96%
			Incorrect	4%

We also attempt to classify known and new types of malware with LCS-based type vectors, permission-based type vectors, and mix-based type vectors. To show that type vectors can be used to identify new types of malware, we prepare both known types and new types of malware, as shown in Table 4. We use a threshold of 0.5 for cosine similarities with LCS-based type vectors, a threshold of 0.8 for cosine similarities with permission-based type vectors, and a threshold of 0.65 for cosine similarities with mix-based type vectors. Table 9 shows the classification results. With LCS-based type vectors, although the correct classification rate is decreased by 10% for known types, more than 81% of new type of malware can be classified correctly. It is worth noting that with permission-based type vectors, the correct classification rate is only decreased by 1% for known types and the correct classification rate is more than 98%. With mix-based type vectors, more than 99% of new type of malware can be classified correctly but the correct classification rate is decreased by 3% for known types. We conclude that permission-based type vector

classifier performs better on classifying malware types.

Table 9. Classification results for known and new types of malware

Category	Malware Type	Number of Malware	Classification Result	Percentage
System Call Sequence Vectors	Known Type	99	Correct	83%
			Incorrect	17%
	New Type	42	Correct	81%
			Incorrect	19%
Permission Vectors	Known Type	99	Correct	98%
			Incorrect	2%
	New Type	42	Correct	98%
			Incorrect	2%
Mix Vectors	Known Type	99	Correct	93%
			Incorrect	7%
	New Type	42	Correct	99%
			Incorrect	1%



## Chapter 6 Conclusions and Future Works

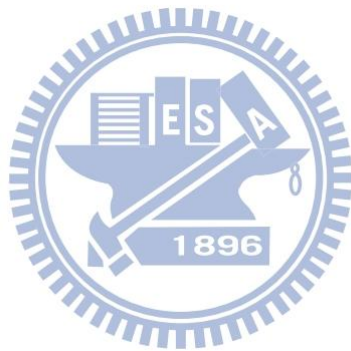
To achieve high detection performance and accuracy, we propose a three-phase behavior-based approach, where the first two phases act as detection mechanisms and the last phase acts as a classification mechanism. We observe application behaviors from two aspects, i.e., permissions and system call sequences. We consider time consumption, false negative rate, and false positive rate to determine the order of the two detection phases. We also evaluate the accuracy of malware type classification.

We adopt various techniques in the design of the proposed detection and classification phases. We use the Bayes theorem to evaluate permission probabilities of being malicious; we use the N-gram and the LCS algorithm to define malicious behaviors from recorded system call sequences. Finally, with LCS-based and permission-based type vectors, we adopt cosine similarity to classify malware into known and new types.

To evaluate effectiveness and efficiency of our approach, we conduct several experiments. The required time for processing a sample with permission based and system call sequence based detector is 2.57 seconds and approximately 600 seconds, respectively. It achieves a good performance of more than 97% true positive rates and less than 3% false positive rates. For malware type classification, with permission-based type vectors, more than 98% of detected malicious applications can be correctly classified into both known and new types.

Although the proposed solution already performs well in several aspects, we think it could be further improved by considering the following information. For system call sequences, we did not consider the parameters of system calls. In addition, currently we only trigger three system events including system rebooting, SMS receiving, and phone calls. More system events could be considered. Furthermore, we

did not interpret the malicious behaviors from system call sequences. If exact malicious behaviors can be interpreted, we believe it could help to better classify the type of malware.



## References

- [1] Android. [online], available from World Wide Web;  
<http://www.android.com/>
- [2] Android (operating system). [online], available from World Wide Web;  
<http://developer.android.com/about/index.html>
- [3] Trendmicro. [online], available from World Wide Web;  
<http://tw.trendmicro.com>
- [4] Kaspersky. [online], available from World Wide Web;  
<http://www.kaspersky.com>
- [5] Lookout. [online], available from World Wide Web;  
<https://www.lookout.com/>
- [6] Bouncer. [online], available from World Wide Web;  
<http://googlemobile.blogspot.tw/2012/02/android-and-security.html>
- [7] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. “Android Permissions Demystified,” *Proceedings of the 18th ACM Conference on Computer and Communications Security*, Chicago, Illinois, USA, pp. 627-638, October, 2011.
- [8] J. Ryan, W. Zhaohui, G. Corey, and S. Angelos, “Analysis of Android Applications' Permissions,” *Proceedings of the 6th IEEE International Conference on Software Security and Reliability Companion (SERE-C)*, pp.45-46, June, 2012.
- [9] B. Sanz, I. Santos, P. Galán-García, C. Laorden, X. Ugarte-Pedrero, P.G. Bringas, and G. Alvarez, “PUMA: Permission usage to detect malware in android,” *Proceedings of the 5th International Conference on Computational Intelligence in Security for Information Systems*, Ostrava (Czech Republic), pp. 5-7, September, 2012.
- [10] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, “Hey, You, Get off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets,” *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2012
- [11] T. Blasing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak, “An android application sandbox system for suspicious software detection,” *Proceedings of the 5th International Conference on Malicious and Unwanted Software*, Nancy, France, pp. 55–62, 2010.
- [12] I. Burguera, U. Zurutuza, and N. T. Simin, “Crowdroid: Behavior-based malware detection system for Android,” *Proceedings of the 1<sup>st</sup> ACM workshop on Security and privacy in smartphones and mobile devices*, Chicago, IL, USA, pp. 15–25, October 2011.
- [13] T. Isohara, K. Takemori, and A. Kubota, “Kernel-based behavior analysis for android malware detection,” *Proceedings of the 7<sup>th</sup> International Conference on Computational Intelligence and Security*, Sanya, Hainan, China, pp. 1011–1015, December 2011.
- [14] Y. D. Lin, Y. C. Lai, C. H. Chen, and H. C. Tsai, “Identifying Android Malicious Repackaged



- Applications by Thread-grained System Call Sequences,” *Computers & Security*, in revision.
- [15] W. Enck, M. Ongtang, and P. McDaniel, “On lightweight mobile phone application certification,” *Proceedings of the 16th ACM conference on Computer and communications security*, Chicago, Illinois, USA, pp. 235-245, November 2009.
- [16] C. Manning, P. Raghavan, and H. Schütze. “Introduction to Information Retrieval,” Cambridge Univ Press, 2008.
- [17] apktool. [online], available from World Wide Web;  
<http://code.google.com/p/android-apktool/>
- [18] strace. [online], available from World Wide Web;  
<http://sourceforge.net/projects/strace>
- [19] Y. Zhou and X. Jiang, “Dissecting Android malware: Characterization and evolution,” *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, Oakland, CA, U.S.A, pp. 95–109, May 2012
- [20] VirusTotal. [online], available from World Wide Web;  
<http://www.virustotal.com/>

