5-2014

Tapjacking Threats and Mitigation Techniques for Android Applications

Vanessa Cooper
Kennesaw State University, vcooper3@students.kennesaw.edu

Tapjacking Threats and Mitigation Techniques for Android Applications

Master's Thesis

by

Vanessa N. Cooper
Kennesaw State University
Bachelor of Science
Computer Science
Kennesaw State University
2011

Submitted in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

May 2014

Tapjacking Threats and Mitigation Techniques for Android Applications

This thesis is approved for recommendation to the Graduate Council.

[illegible]
Hisham M. Haddad
Thesis Co-Advisor

[illegible]
Hossain Shahriar
Thesis Co-Advisor

[illegible]
Ken Hoganson
Department Chair, Committee Member

[illegible]
Ying Xie
MSCS Director, Committee Member

[illegible]
Jose Garrido
Assistant Chair, Committee Member

DEDICATION

To my mother: Ona Cooper

To my siblings: Jeff, O.T., and Courtney

The loving memory of my father: Ronnie Cooper

The loving memory of my grandparents:

Ivy and Ruby Polk & Otis and Viola Cooper

ACKNOWLEDGEMENTS

I want to thank my thesis advisors,

Drs. Hisham Haddad and Hossain Shahriar.

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

With the increased dependency on web applications through mobile devices, malicious attack techniques have now shifted from traditional web applications running on desktop or laptop (allowing mouse click-based interactions) to mobile applications running on mobile devices (allowing touch-based interactions). Clickjacking is a type of malicious attack originating in web applications, where victims are lured to click on seemingly benign objects in web pages. However, when clicked, unintended actions are performed without the user's knowledge. In particular, it is shown that users are lured to touch an object of an application triggering unintended actions not actually intended by victims. This new form of clickjacking on mobile devices is called *tapjacking*. Much of the research work has focused on developing mitigation techniques on web application level clickjacking issue. However, none of the research has thoroughly investigated attacks and mitigation techniques due to tapjacking in mobile devices. In this thesis, we identify coding practices that can be helpful for software practitioners to avoid malicious attacks and define a detection techniques to prevent the consequence of malicious attacks for the end users. We first find out where tapjacking attack type falls within the broader literature of malware, in particular for Android malware. In this direction, we propose a classification of Android malware. Then, we propose a novel technique based on Kullback-Leibler Divergence (KLD) to identify possible tapjacking behavior in applications. We validate the approach with a set of benign and malicious android applications. We also implemented a prototype tool for detecting tapjacking attack symptom using the KLD based measurement. The evaluation results show that tapjacking can be detected effectively with KLD. This thesis is organized in the following format: a classification of Android malware, a survey of mitigation techniques, a discussion of our proposed KLD-Based approach, and an application implementation.

TABLE OF CONTENTS

CHAPTER 1

Motivation, Problem Statement, and Contribution

1.1 Overview

With mobile applications, the user's actions are always passed back to an activity. An activity is a representation of a screen or view. Tapjacking takes advantage of this process by initiating methods when a user gestured user interface (UI) elements in the activity. These can cause damage in a variety of ways. Some methods simply aim to be a nuisance by changing the user's mobile phone background. Other methods can be much more detrimental by changing the user's mobile phone lock password and taking over control of a mobile application or device. Figure 1 shows three UI elements that could trigger hidden malicious code: the *Sign in* button and the two editable text fields *Email* and *Password*.

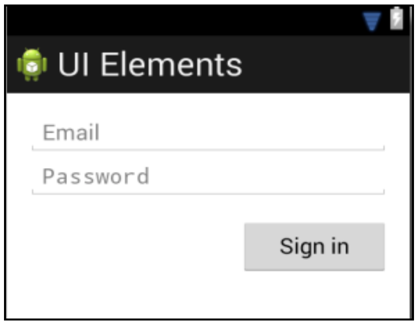


Figure 1: Screenshot of Android application that shows three UI elements

1.2 Motivation

Little work has been done on understanding the scope and extent of tapjacking attacks within mobile devices. Moreover, end users do not have any protection to reduce unwanted consequences caused by tapjacking. The most affected individuals are those who are not aware of the characteristics of possible malware. Malicious code triggers activities which could be as simple as copying user input from a text field to infinitely running a program in the foreground without user knowledge. Most mobile application developers are oblivious to the importance of security within their mobile applications. There are many potential losses when taking into account that many mobile device users access their bank accounts, school information, and daily schedules using applications.

Most mobile applications require access to very sensitive user information, such as birthdates, physical and mailing addresses, and other uniquely identifiable information (such as the device International Mobile Station Equipment Identity (IMEI)). The IMEI is the "social security number" of the mobile device. When considering a mobile application such as Facebook, tapjacking attacks would provide access to a user's most personal information and photos, as well as a list of the user's family and closest friends. Within LinkedIn, a user's business contact information, current employer, and professional profile are heavily exposed. For example, a malicious attack could post unflattering information on a LinkedIn user's profile. These are just few examples of tapjacking threats to Android mobile applications.

1.3 Problem Statement

Tapjacking allows malicious developers to completely hijack a mobile device or to simply perform malicious acts. In addition, if malicious mobile applications have unnecessary permissions to the mobile device, then they can perform even more malicious activities. Fortunately, most security experts are able to scan for unneeded permissions and prevent applications from being published into their respective App Stores. However, if mobile users decide to download applications from unknown sources, and enable permissions, then they open themselves up to vulnerabilities. In this thesis, detection and mitigation techniques for tapjacking malware are explored so that mobile users have a chance to check mobile applications before installing to their devices. By detecting malicious code before installation, mobile users will have a peace of mind in the safety of the personal information and their mobile devices.

This research is intended to answer the following question:

Given that we have an access to both legitimate and malicious applications performing a specific functionality, how do we distinguish a good behavior (or functionality) from a bad behavior?

1.4 Research Methodology

The research methodology involved an intensive literature review of over 30 papers involving malware detection in Android mobile applications and the overview of the Android operating system. We identified the primary detection techniques such as sandboxing, machine learning, and permission analysis based methods. In identifying the advantages and disadvantages of each technique, we were able to determine appropriate measures to detect malware with the least disadvantages. Our literature review concluded with the KLD-based approach and its newfound popularity in security mechanisms. We also performed an experiment to evaluate our proposed KLD-Based approach with a metric-based approach.

Our analysis included the identification of the required source code and permissions that would allow us to perform some very popular malicious actions. By linking the source code and permissions, we were able to determine the intention of source code by checking the permissions in the application's Android Manifest file and reviewing the results of the static code analysis. Each application's functionality is tied to a permission and set of elements. We identified the set of elements and permissions required for our SMS case study. The next step in our process was to devise a way to decompile and analyze mobile applications in a secure environment before computing the occurrence probability. We then implemented a java application that allows the user to select an Android application, decompile the application to readable source code, and analyze the application using our developed prototype class. Our application implementation produces a CSV file that includes the data from our KLD-Based approach. Using the data, we are able to compute the KLD value for each of the evaluated Android applications in relation to the known good Android application. Using this method, a user is able to accurately determine the malicious nature of an Android application with the least error.

1.5 Contribution

This work addresses the stated research question by performing an in-depth study of tapjacking attacks and applications that are responsible for these attacks. In particular, emulating tapjacking attacks with a mobile application and understand the application code elements including API call patterns and permissions causing tapjacking attacks. This work also identifies a new detection technique based on Kullback-Leibler Divergence metric in an effort to help not only mobile application developers, but also end users who may not have any technical knowledge on tapjacking attack. More specifically, the contributions of this work include the following:

a) An overview of the Android Operating System (OS) and a classification of Android malware, understanding the code level and permission level features (Cooper *et al.* 2014) that are responsible for malicious activities (Chapter 2)
b) A survey of literature work intended to mitigate malware activities during application development and deployment stages, discuss the importance of mobile device security and user information, outline common vulnerabilities in mobile applications (Chapter 3)
c) A KLD-Based approach to differentiate between malware and good applications based on source code level features and apply the concept to detect suspected malware (Chapter 4)
d) An explanation of the application implementation for our KLD-Based approach, outlining the required steps and intended results in our experiment (Chapter 5)

CHAPTER 2

Technology Overview

2.1 Technology Overview

Android has become the leading smartphone OS in the world with staggering sales figure of 60 million phones in the third quarter of 2011, 50% market share (Aaron, 2011). A recent study shows that more than 50% of Android mobile have unpatched vulnerabilities, opening them up to malicious applications (malware) and attacks. A compromised smartphone can inflict severe damage to both users and the cellular service provider. Malware applications can make the phone partially or fully unusable, cause unwanted billing, steal private information, or infect every name in a user's phonebook (Reza & Mazumder, 2012).

Recently, a malware affected more than 100,000 Android devices in China (known as *MMarketPay*). This malware is a hidden application that appeared to be legitimate and is designed to purchase applications and contents without the consent of the device users (victims). As a result, victims saw a staggering amount of bills (Baldwin, 2012). The incident prompted Google, the developer of the Android OS, to introduce stricter rules for applications on Android such as naming of applications and banning applications that disclose personal information without user permission. An Android Short Message Service (SMS) malware firm was fined £50,000 by the UK premium phone services regulator *PhonepayPlus ("PhonePay Plus", 2013)*. An SMS is a text messaging service available on most mobile devices and is a very popular choice of communication.

Possible attack targets into smart phones include Cellular networks, Internet connections (via Wi-Fi, General Packet Radio Service / Enhanced Data rates for Global Evolution (GPRS/EDGE) or 3G network, Universal Serial Bus (USB) and other peripherals (Shabtai, Fledel, & Elovici, 2010). Given all these, it is important to study malicious Android applications and their characteristics. A solid understanding of the characteristics of malware is the beginning step to prevent much of the unwanted consequences. This chapter is intended to overview the Android OS, its architecture, and security threats posed by Android malware. In particular, we focus on the characteristics commonly found in malware applications and understand the code level features that may lead to the detection of the malicious signatures for prevention. We also discuss some common defense techniques to mitigate the impact of malware applications.

2.2 Android OS

Android is an open source OS based on the Linux first launched in 2007 and intended for mobile phones (Rehm, 2012). Between the two major variants of smartphone (Android and iOS), Android is the most popular one. As of October 2013, the latest version of Android OS is 4.4 (commonly known as KitKat supporting API level 19). Being developed and supported by Google, all Android devices allow users to synchronize access to storage and communication services provided by Google. For example, users can login to Google Gmail to check email and access contact list, calendar, and other free applications automatically. The default desktop of Android has five screens that can be switched by tapping. A user can move any icon to any place on the desktop by tapping and hovering. Android devices allow users to download and install new applications for legitimate purposes that may include game, business, communication, photography, and service. The common place to find applications is the Google Play Store ("Google Play", 2013).

The Android Developer manual recommends some common practices for programmers for developing applications ("Android Design", 2013). These include the guidelines for developing applications that are visually appealing to users. A developer can reuse standard theme that control visual properties of the elements for user interface of an application such as color, height, padding, and font size. Recommended guidelines for color and illumination of icons are provided to represent different state of an icon (e.g., a gray colored icon means static, illuminated icon means "pressed", 50% illumination means "focused", 30% of illumination means "disable"). Developers can choose different color styles and text font sizes. The guidelines recommend using *textColorPrimaryInverse* and *textColorSecondaryInverse* for light themes. Also to maintain consistency of look and feel in the same UI, it is recommended to use scale-independent pixels (sp) wherever possible.

Legitimate applications support well-known gestures to allow users interacting with applications based on the screen objects. Table 1 shows the core gesture set that is supported in Android. Unlike desktop or laptop computers, activities and operations can be performed on Android devices based on touching (also known as tapping). Note that a "tap" is a brief touch followed by the release of touch on a certain entity of Android screen. Usually, “tap” is considered as a single event for smartphone device and applicable for a visible icon. Most legitimate applications are developed in a way so that useful operations are performed based on user-initiated gestures. Nevertheless, some legitimate applications may not need gestures to perform operations (*e.g.*, an application that is intended to clear cache data periodically upon installation). For this thesis, we can fairly assume that most good applications have a visible Graphical User Interface (GUI) or UI elements to enable tapping, and the actions preformed are expected by users. In contrast, for malware, a visible GUI may trigger different or hidden actions without the user's knowledge.

Table 1: A List of Gesture Types Supported in Android
(“Android Design”, 2013)

Type	**Description**	**Action**
Touch (tap)	Triggers the default functionality for a given item.	Press, lift
Long press	Enters data selection mode. Allows a user to select one or more items in a view and act upon the data using a contextual action bar.	Press, wait, lift
Swipe	Scrolls overflowing content or navigates between views in the same hierarchy.	Press, move, lift
Drag	Rearranges data within a view, or moves data into a container (e.g. folders on Home Screen).	Long press, move, lift
Double touch	Zooms into content. Also used as a secondary gesture for text selection.	Two touches in quick succession
Pinch open	Zooms into content.	2-finger press, move outwards, lift
Pinch close	Zooms out of content.	2-finger press, move inwards, lift

2.3 Android Architecture

The Android OS framework has a number of layers to facilitate the execution of applications (Shabtai *et al*. 2010). Table 2 shows an overview of the OS framework (“Android Design”, 2013). The bottom layer has the Linux kernel. On top of the kernel, a set of native libraries (C/C++) and the Android virtual machine (Dalvik, which is the Android-specific implementation of the Java virtual machine) reside. The Dalvik VM relies on the underlying Linux kernel to handle low-level functionalities such as process and memory management. The Dalvik VM executes .dex files (Dalvik executable), which can be created by transforming Java classes using the SDK tools (“Memory Management in Android”, 2010). The next layer is the Application Framework encompassing the Java core libraries, which rely on the native libraries. The topmost layer contains the Java-based applications that are created using the Application Framework layer. Java Applications communicate with the Android Framework through a variety of key applications, such as Messaging, Gallery, and the Camera (Shabtai *et al*. 2010).

Table 2: Architectural Overview of Android OS
(“Android Design”, 2013)

APPLICATIONS layer				
Home	Contacts	Phone	Browser	
APPLICATION FRAMEWORK layer				
Activity Manager	Window Manager	Content Providers	View System	
Package Manager	Telephony Manager	Resource Manager	Location Manager	Notification Manager
LIBRARIES			**ANDROID RUNTIME**	
Surface Manager	Media Framework	SQLite	Core Libraries	
OpenGL | ES	FreeType	WebKit	Dalvik Virtual Machine	
SGL	SSL	libc		
LINUX KERNEL layer				
Display Driver	Camera Driver	Flash Memory Driver		Binder (IPC) Driver
Keypad Driver	WiFi Driver	Power Management		Audio Drivers

2.4 Security Features

Android has a number of built-in security features to protect the data and memory that belong to processes or applications running on the device (“Security Tips”, 2013). We discuss core security features including *sandbox*, *permission-based access control*, *secure Inter Process Communication (IPC)*, *safe memory management*, and *data encryption*.

Sandbox:

In Android, each application runs on a sandbox (i.e., each process has its own copy of the virtual machine). As a result, an application cannot access the data and code of another Android application. Sandboxes are regularly used for scanning programs and applications that contain unverified developer certificates. Because sandboxing isolates each application, it provides a more stable environment and prevents other applications from being infected from a malicious application.

Permission-based access control:

User-granted permissions for each application are the basis to grant or restrict access to system features and user data. During installation of an application, the permissions required to operate different peripherals are declared and a user is prompted whether or not he/she intends to grant/deny the permission. If a user does not grant any of the needed permission, the application is not installed.

Secure IPC:

An application cannot directly access other application's memory spaces (containing data). Thus, the Inter Process Communication (IPC) mechanism plays a key feature in accessing data from one application to another application. A developer needs to implement IPC based on the following three steps ("Android IDL Example with Code Description – IPC", 2013): implementation of Android Interface Definition Language (AIDL) interface, implementation of remote service, and exposing the remote service to local clients.

Safe memory management:

Each Android application runs in a separate process within its own Dalvik instance. Dalvik is a register-based virtual machine optimized to ensure that a device can run multiple instances efficiently. Dalvik is responsible for memory and process management during run time and can stop and kill processes as necessary. Memory management related vulnerabilities such as buffer overflow, memory leak, and uninitialized pointer usage are eliminated by incorporating some of the well-known technologies like Address Space Layout Randomization (to prevent code injection attack), NX (non-executable stack due to buffer overflow), and ProPolice (return address space corruption prevention).

Data encryption:

Android allows users to encrypt their data and other profile information. It is possible to encrypt accounts, downloaded applications, media files, and settings. An encrypted device can be decrypted based on a user chosen password (during each time the device is powered on). The encryption process is costly both in terms of processing power (device needs to be plugged with power) and time (can take more than an hour) (Brinkmann, 2012).

2.5 Android Malware

Malware or "malicious software" is implemented with malicious intention. Malware is often installed without the victim's knowledge of the capability of unintended actions that can be performed. More specifically, victims usually overlook the list of permissions needed to run the malware and voluntarily grant the permission without understanding the effect of malicious actions. Under the broad definition of malware, several categories are well-known including virus (a malicious program that can copy itself in an infected computer), worms (similar to virus, except having the ability of propagation in new machines), and Trojan horses (a program that installs a backdoor in an infected computer to communicate with hacker-controlled computer) ("What is Malware?", 2013).

Table 3: A List of Malicious Actions Performed by Android Malware
(Felt *et al.* 2011)

Malware Type	Example Action	Required Permissions
Changing Wallpaper Setting (M1)	Novelty and amusement by change the default wallpaper without user's permission (personal).	*SET_WALLPAPER*
Accessing User Credentials (M2)	Secretly accessing user information stored on the Android device.	*GET_ACCOUNTS*
SMS Message and Premium Rate Calls (M3)	Bills victim by arbitrarily initiating phone calls to premium numbers or sending text messages to premium numbers.	*SEND_SMS* *CALL_PHONE* *CALL_PRIVILEGED*
Phone Ransom (M4)	Locking a client's phone by changing default setting on password or other profile information.	*DISABLE_KEYGUARD* *WRITE_SETTINGS* *WRITE_SECURE_SETTINGS*
Hacking Social Networks (M5)	Secretly accessing and updating user profile information on a social network (device).	*READ_SOCIAL_STREAM* *WRITE_SOCIAL_STREAM*

Tapjacking is another form of malware. The act of tapjacking occurs when a user unknowingly triggers a malicious code by clicking a button or a view. There are several ways to initiate tapjacking attacks, and this thesis explores five different types of malicious tapjacking actions. Table 3 shows a classification of tapjacking malware that are capable of performing specific operations in the Android platform. Tapjacking malware includes the changing of the desktop setting by installing wallpaper without user knowledge (M1), accessing device and personal profile information and sending it over the Internet to unwanted third parties (M2), launching phone calls and sending messages to premium numbers (M3), asking for ransom by locking the phone and suggesting to pay for unlocking (M4), and hacking social network accounts (M5). Each of these malware types requires one or more permission changes for the malware to take its course.

Note that some of the malware applications are known as spyware. Spywares are programs developed to monitor and log activities performed on a computer (e.g., Keylogger). Spyware not only collects sensitive personal information (e.g., websites visited, typed password), but also steals information, and in the worst case can send them to others for further damages ("Difference between Adware and Spyware", 2005).

Adware is another malware application type. Adware displays advertisements and marketing contents automatically after the installation. Advertisements are displayed in a small section of the interface or as a pop-up window. It is used for legitimate reason such as generating revenues for companies who intend to sell products. An example of adware is the popular e-mail program named Eudora ("Eudora", 2014). It can be purchased in sponsored mode, when Eudora displays an advertisement window containing toolbar links. We do not consider such adware as malicious.

2.6 Classification of Android Malware

In this section, we show code level examples of tapjacking malware that can represent the five types of tapjacking malware discussed in Table 3. Tapjacking is the root cause of the five mentioned malware types because of its similar deceptive, malicious acts. Figure 2 shows how a tapjacking attack could occur when a user clicks a submit button in a mobile application. The submit button could be as simple as sending an SMS or even updating your Facebook profile. Each time the user clicks the submit button, the *onClick()* method is called and the *openNextUIView()* and *startMaliciousCode()* methods are executed. As the user views the next UI view, malicious code is being executed without their knowledge.

```
//user clicks a submit button on the screen
Button submitButton = findViewById(R.id.clickButton);

submitButton.setOnClickListener( new OnClickListener() {
      @Override
      public void onClick(View v) {
            openNextUIView(); //show the next UI screen
            startMalicousCode();    //display malicious code
      }});
```

Figure 2: Tapjacking attack triggered by button click

We discuss the key part of Java code and the list of permissions that appear in *AndroidManifest.xml* file for the reader’s convenience. It is important to note that both sections of code, Java code and permissions, are necessary to perform the listed malware actions. All java source files and interactive user views (activities) must be listed. This is a requirement for all mobile applications. In all malicious actions, a user is first required to agree to the permission list when downloading and installing the mobile applications. Therefore, it is very important for a user to remain vigilant about requested permissions in mobile applications.

2.7 Changing Wallpaper (M1)

Earlier, we discussed how malicious code could be used to change the mobile phone’s wallpaper. Though this may seem like a fairly trivial act, one must realize that changing the wallpaper is accessed through the mobile device settings. If malicious developers can gain access to your mobile device settings, then they can do almost anything that they desire on your mobile device. In Figure 3, we examine the source code responsible for executing the malicious action of changing the wallpaper without the user’s specification. In this case, the required permission is *SET_WALLPAPER*, shown in Figure 4. Without this line of code, the malicious code would be ineffective. During development, permissions are automatically added when a developer invokes

Android classes directly linked to that permission. However, the system is not able to determine if the developer is attempting to use the permission in a malicious way.

```
//Retrieve instance of the application
WallpaperManager myWallpaperManager =
    WallpaperManager.getInstance(getApplicationContext());

//R.drawable.five presents a stored image
myWallpaperManager.setResource(R.drawable.five);
```

Figure 3: Required source code to change wallpaper
Source: "Set Wallpaper using WallpaperManager", 2011

```
<uses-permission
      android:name="android.permission.SET_WALLPAPER" />
```

Figure 4: Required permission for changing wallpaper
Source: "Set Wallpaper using WallpaperManager", 2011

In Figure 5, we examine a code snippet on how to change the sound settings on the mobile device. A malicious application can access the *AudioManager* and set the ringer volume to zero. As a result, a victim will not be altered or notified for related activities such as incoming phone call or SMS messages. On the contrary, their phone's ringtone could sound very loudly during an important business meeting.

```
//Access system settings for the audio
AudioManager audio =
(AudioManager)getSystemService(Context.AUDIO_SERVICE);

//Change Ringer to Silent
audio.setRingerMode(0);
```

Figure 5: Silence the sound settings on an Android device
Source: "How to make android phone silent in java", 2012

2.8 Accessing User Credentials (M2)

As stated above, the mobile device settings are the key to the control of the mobile device. However, it is also equally important to secure personal information on the device. Most mobile applications have the ability to run continuously in the foreground. These mobile applications could be anything from Gmail, Facebook, or Instagram. In order to gain access to the accounts linked to your Facebook or Instagram, you only need a username and password. The username and password are both treated as a string of characters. If your mobile application is running in the foreground in an open session, it is possible to retrieve the data associated with that mobile application.

```
Pattern emailPattern = Patterns.EMAIL_ADDRESS;

// Functionality is available for API level 8+
Account[] accounts = AccountManager.get(context).getAccounts();

for (Account account : accounts) {
      if (emailPattern.matcher(account.name).matches()) {
            String possibleEmail = account.name;
      }}
```

Figure 6: Required source code to access user account information
Source: "How to get the Android device's Primary Email Address", 2010

Figure 6 shows how a malicious mobile application can access and retrieve a user's email address. It's important to note that this source code applies to devices with an API level of 8 or greater. First, the malicious code seeks to retrieve the email address. Then, the code searches the device for all accounts, denoted by *getAccounts(),* associated with that email address. Most times, we use the same email address for our social networking accounts, school accounts, and personal accounts. Lastly, all of the accounts are iterated over in order to find the user's login, or *account.name*.

The *GET_ACCOUNTS* permission, shown in Figure 7, is the only required permission for retrieving user accounts. However, if a developer wanted to make changes to the user account information, they would be required to list permissions for editing the user account. This means that the mobile application would seek to acquire read and write access for user account information. However, this case only seeks to retrieve or get the user's accounts.

```
<uses-permission
      android:name="android.permission.GET_ACCOUNTS" />
```

Figure 7:Required permission for retrieving user account information
Source: "How to get the Android device's Primary Email Address", 2010

2.9 SMS Message and Premium Rate Call (M3)

An SMS is the primary choice of communication for most people today. Unfortunately, SMS message sending is also one of the most popular types of malicious activities. SMS messages can be easily sent, received, and read while at work, in meetings, etc. In addition, most people carry their mobile devices everywhere; this makes SMS a very efficient portal of communication. Figure 8 shows the only permission, *SEND_SMS*, required to send an SMS message. However, sending an SMS message is also a motivating case because there are two ways to send a message.

```
<uses-permission
      android:name="android.permission.SEND_SMS"/>
```

Figure 8: Required permission to send SMS message
Source: "Send SMS in Android", 2013

The first option is shown in Figure 9. It outlines a hidden attempt to send an SMS message. Here, *SmsManager.getDefault()* returns the default SMS engine. The *sendTextMessage()* method is called to send a message. This way of sending a message can easily be included in any method or loop without the user's knowledge. Since the action is hidden and does not require user input, it can be flagged as suspicious or malicious activity. However, this method could also be used to send the SMS message after retrieving the user input from the UI elements. Therefore, scanning for this method signature could also lead to a false positive warning. The key indicator to determining if it's being using maliciously is to look for hard-coded values that are not passed back from the user's input into the UI.

```
//Retrieve the default SMS engine
SmsManager sms = SmsManager.getDefault();

//Send a text message using desired text
sms.sendTextMessage(phoneNumber, null, message, null, null);
```

Figure 9: Hidden method to send SMS message
Source: "Send SMS in Android", 2013

Note that among the five parameters, the first is used to supply a phone number (variable or hard coded), the second is the service center address but is not required because the default will be used, the third is for the message contents, the fourth broadcasts when the message is sent (if this parameter is not null), and the fifth broadcasts when the message is delivered (if this parameter is not null).

Figure 10 shows the second option of sending an SMS message using Intent object creation followed by launching an activity running on the background (*startActivity()* method call). Note that during the Intent object creation, a *Uri.parse()* method is invoked to indicate the sending of SMS message to a phone number. Such SMS sending operation also does not require any interaction with a user, hence, can be considered as potentially malicious. Note that the destination phone number and the desired message are retrieved directly from the UI elements and sent to the next view, or activity.

```
//Send a text message using text from user’s screen
startActivity(new Intent(Intent.ACTION_VIEW, Uri.parse("sms:"
+ phoneNumber)));
```

Figure 10: Visible method to send SMS message
Source: “Send SMS in Android”, 2013

In Figure 11, we show how a mobile application can initiate a phone call. In this case, a phone call is initiated using the Intent object creation (specifying appropriate flag of *Intent.ACTION_CALL*). Note that the dialer is never used here, as a result a user will not notice that a phone call is initiated. Moreover, the supplied phone number (*number*) can be a fixed hard-coded premium number is called without the user’s knowledge. This can lead to expensive phone bill, especially if the mobile application is left running overnight while the user is away from the device. In order to perform this action, a malicious developer would include the permissions listed in Figure 12.

```
//Initiate a phone call using desired phone number
String number = “1-900-444-8821”;

Intent callIntent = new Intent(Intent.ACTION_CALL, Uri.parse(number));

startActivity(callIntent);
```

Figure 11: Initiating a phone call without using phone dialer
Source: "How to make a phone call in Android", 2011

```
<uses-permission
      android:name="android.permission.CALL_PHONE"/>

<uses-permission
      android:name="android.permission.CALL_PRIVILEGED"/>
```

Figure 12: Required permissions to make phone call without phone dialer
Source: "How to make a phone call in Android", 2011

2.10 Phone Ransom (M4)

Phone ransom is a fairly new occurrence in mobile malware. By gaining access to the user's settings, a malicious developer can change the mobile device password and lock the mobile user out of their own device. Normally, a message is then displayed on the wallpaper or lock screen, which prompts the user to either pay to unlock the phone or to simply taunt the user for being breached.

Figure 13 shows how to lock the screen of a mobile device. The *KeyguardManager* is accessed which further accesses the *KeyguardLock* for enabling or disabling the default lock. One objective of malware is to disable the lock for the purpose of ransom. A message is later displayed prompting the user to pay a fee in order to unlock the device and continue unharmed. However, this is often just a ploy in order to retrieve funds from a very desperate person. Figure 14 shows the required permissions to edit phone settings and save them accordingly.

```
//Access system settings for the keyguard
KeyguardManager mgr =
      (KeyguardManager)getSystemService(Activity.KEYGUARD_SERVICE);

// Lock the device
KeyguardLock lock = mgr.newKeyguardLock(KEYGUARD_SERVICE);

lock.disableKeyguard(); //Disable the keyguard from showing
```

Figure 13: Lock an Android device and disable keyguard
Source: "Lock and Android phone", 2012

```
<uses-permission
      android:name="android.permission.DISABLE_KEYGUARD "/>

<uses-permission
      android:name="android.permission.WRITE_SETTINGS "/>

<uses-permission
      android:name="android.permission.WRITE_SECURE_SETTINGS"/>
```

Figure 14: Required permissions to disable keyguard
Source: "Lock and Android phone", 2012

By listing *WRITE_SETTINGS* and *WRITE_SECURE_SETTINGS*, we are able to cover more circumstances. The first simply allows a malicious developer to make changes to all of the device settings. The second is used for mobile applications signed by the operating system. Together, this is a very strong combination for having complete access to alter a mobile device according to the malicious developer's desires.

2.11 Hacking Social Networks (M5)

Malicious activities have escalated even higher with Android's added ability to synch mobile application with social networks in API Level 15. Now, a user can update his/her status on Facebook, Twitter, and other social networks directly from the mobile device. With this added implementation, many security threats have emerged and malicious attacks can be mounted. In Figure 15, we examine how a malicious mobile application can easily gain access to a user account and send fraudulent status updates to the user profile.

```
//Create status update to post on user profile
ContentValues values = new ContentValues();
values.put(StreamItems.RAW_CONTACT_ID, rawContactId);       //destination
values.put(StreamItems.TEXT, "Lunch at 3.00 PM");           //message
values.put(StreamItems.TIMESTAMP, timestamp);               //timestamp
values.put(StreamItems.COMMENTS, "Family and Friends");     //comments

//Specify where content will be posted and send request to post content
Uri.Builder builder = StreamItems.CONTENT_URI.buildUpon();
builder.appendQueryParameter(RawContacts.ACCOUNT_NAME, accountName);
builder.appendQueryParameter(RawContacts.ACCOUNT_TYPE, accountType);
Uri streamItemUri = getContentResolver().insert(builder.build(), values);
long streamItemId = ContentUris.parseId(streamItemUri);
```

Figure 15: Code snippet for updating social network account
Source: "Get Social Updates of your contact list using Ice cream sandwich", 2012

In the first section of Figure 15, the code fills in required format and the desired contents to be posted on the account. Then, the code acquires access to that user account. After gaining access to the user profile, a malicious activity can then gather the user's interests, friend's list, and a multitude of other details. Since individuals tend to post birthday pictures, pet names, and other private information, they are vulnerable to identity theft. As shown in the last section of the figure, the request is sent in a readable format to the destination address, and the user's account is updated with the fraudulent information. Figure 16 outlines the required permissions for accessing and updating a user profile on a social network.

```
<uses-permission
      android:name="android.permission.READ_SOCIAL_STREAM "/>

<uses-permission
      android:name="android.permission.WRITE_SOCIAL_STREAM"/>
```

Figure 16: Required permissions to update social network profile
Source: "Get Social Updates of your contact list using Ice cream sandwich", 2012

CHAPTER 3

Literature Review

3.1 Overview

This section presents a literature review of recent work on Android malware and the various techniques for mitigation of Android malware applications. Many detection techniques have been proposed in the literature to enhance the security of Android platforms and deployed applications. We chose three detection techniques that closely relate to our proposed KLD-based detection technique. These techniques include sandboxing systems for Android applications (Blasing *et al.* 2010), machine learning to extract static features of Android applications (Shabtai *et al.* 2010), decompiler-based static analysis (Enck *et al.* 2011), and permission-based detection techniques (Barrera *et al.* 2011). These techniques were compiled during a literature study of malware in mobile applications; we briefly explain the advantages and disadvantages of these related works.

3.2 Sandboxing Detection

A sandbox (Blasing *et al.* 2010) provides a realistic execution environment, but in an isolated manner. As a result, the effect of a potential malicious application does not affect the outside environment. It is useful not only for signature identification, but also for disinfecting a malware. The sandbox has two steps: *static* and *dynamic* analysis.

An Android application is shipped as a compressed (apk extension) installation file. In static analysis, the sandbox decompresses installation files and disassembles executable files to identify malicious code fragments. When decompressed, the content is saved into three main parts: *AndroidManifest.xml* (an XML file having the meta-information of the application including its description and security permissions), *classes.dex* (a file having the Java bytecode that can be interpreted by Dalvik Virtual Machine), and *res* (a special folder having files that define the layout, language, and so on).

The manifest file contains the main "launchable activity" information. The byte code (from *classes.dex*) of the application is converted to human readable format having a folder hierarchy containing files with parsable pseudo-code. The code is then scanned for suspicious patterns. A list of static code patterns that are commonly considered as Android malware (Blasing *et al*. 2010) are as follows: the usage of the Java Native Interface, the usage of *getRuntime, the* usage of Java reflection, the usage of services and IPC provision, and the usage of android permissions.

The dynamic analysis phase of the sandbox system is intended to monitor system and library calls with arguments. In general, system calls are function invocations made from user space into the kernel to request services or resources from the operating system (Hyatt, 2013). A Loadable Kernel Module (LKM) is implemented and placed in the Android emulator environment. The modified kernel keeps logging the function calls invoked by applications and their arguments for later analysis. This gives a low-level system call sequence responsible for malicious activities.

Advantages: The sandbox reduces the generation of signatures based on system level call tracing. It has been shown that on average it takes 48 days to come up with the signatures of a new malware, which leaves the window of damaging opportunity by malware wide (Oberheide, Cooke, & Jahanian, 2008).

Disadvantages: As the lowest level of system calls are intercepted and logged, implementation of a loadable kernel module (LKM) is daunting and error prone task. Special attention is needed as emulator tends are very unstable if low-level changes are performed.

3.3 Machine Learning Detection

Machine Learning algorithms originated as heuristic-based detection methods that could easily evaluate software in search of malware. Since machine learning is automated, malicious features are predetermined and normally classified by their distinct code patterns. In addition, machine learning can process static code and determine its malicious capability. Static analysis uses significantly less time and resources. More importantly, it does not require the mobile application to be executed as in dynamic analysis. Shabtai *et al.* (2010) apply the machine learning technique to differentiate the characteristic of applications between two categories: tools and games. They extracted features from the byte-code (dex files) and XML (permission). The learned features were used to identify the general type of the application, which can be used as an indicator for potential malicious activities.

The machine learning process has two phases: training and testing. First, a classification model is derived from a group of predetermined vectors and labels that represent the learning algorithm. This model is referred to as the training set. For accuracy and inclusion, the training set should include a wide variety of malicious applications. However, it's equally important that learning algorithm is able to properly identify the varying code patterns the malicious mobile applications. Then, a testing set of APKs is parsed according to its identifier, or its obvious malicious features. Each of the malicious actions exists within a representative vector and can be used to predict the origin of the malicious activity. If a malicious feature is flagged in the testing phase, the learning algorithm is able to determine which class files are affected.

There are three main problems with the extraction of malicious features: misleading the learning algorithm with inaccurate features, over-fitting or crowding with the amount of features to be evaluated, and creating a model complexity which exceeds the power of the learning algorithm (Shabtai *et al*. 2010). For accuracy and efficiency, filters are used to prevent the occurrence of the three difficulties above. These filters are responsible for ranking and scoring the features and determining which features are selected for the classification model.

Advantages: The approach is automated and can enable the static detection of malware applications. This proves to be extremely beneficial in cases where executing a possibly malicious application would cause harm to the evaluator's machine.

Disadvantages: Depending on the type of classification algorithms, performance will vary. Also, the accuracy of training is important. A good initial dataset representing all types of applications are needed. If an application fits into overlapping category (e.g., a game application need to send information over the internet to store score of a user online which may be of similar to an application intended for browsing on the web), then machine learning is prone to false positive warning for benign application.

3.4 Static Analysis Detection

Enck *et al.* (2011) analyzed a large set of android applications collected from market to identify a set of dataflow, structure, and semantic patterns. It is also very important to evaluate the development background and run-time environment compilation of an Android application, such as the application structure, register architecture, and the instruction set. The dataflow patterns identify whether any sensitive data information piece should not be sent to outside (*e.g.*, IMEI, IMSI, ICC-ID). The structural analysis logs any API usage for retrieving sensitive information such as device ID or telephone manager. The semantic analysis performs the arguments of parameter method calls. For example, when a text message is being sent, it is checked if it is being used either to a constant or a dynamic number. The earlier might represent a malicious application activity. Their observation from seemingly benign applications can be considered as features to develop signatures.

Their *ded decompiler* (Enck *et al*. 2011) can recover the original application source code. The source code is then scanned and analyzed to uncover possible security threats. Though Enck *et al.* did not focus malware analysis in their study, the decompiler uncovered misuse of phone metadata. The analysis of the application source code revealed 27 findings of data misuse and improper coding practices. Some of those findings include "Phone identifiers are frequently leaked through plaintext requests", "Phone identifiers are sent to advertisement and analytics servers", "Some developer toolkits probe for permissions", and "Few applications unsafely delegate actions".

Batyuk *et al.* (2011) proposed not only the detection of malicious application's signature but also proposed a flexible mitigation approach. They performed static analysis on binary code of android applications (after decompressing APK and decoding Java bytecode into Smali assembly language). They looked for the presence of APIs that may be relevant to reading sensitive information (e.g., IMEI or device identifier, IMSI or subscriber identifier, phone number, ICC-ID or SIM card serial number, writing information to output stream) as well as any functionality for third party usage related to "Ads" and "Analytics". The mitigation approach can accommodate users' needs, which could be to either deny the installation of application based on the generated report or apply patching to mitigate potential security risks.

Yang *et al.* (2012) detected money-stealing malware. They examine the manifest file of android applications to see if a billing permission is present. Then they looked for specific method calls or APIs that perform SMS messaging or calls to premium phone numbers. Finally, they check for the presence of notification suppressor (*i.e.*, extending *SmsReceiver* or *BroadcastReciever* classes and overriding *onReceive* or *abortBroadCast* methods, respectively to suppress message sent notification supplied from the corresponding ISPs) that prevents victims from knowing that messages are being sent or calls have been made without their consent.

Seo *et al.* (2012) developed a framework to automatically decompile the package of android applications from both official websites (e.g., Google's Android Market, Apple's AppStore) and third party (or black marketers). Then analyzed the decompressed source files to obtain the API calls present in methods and applied known information about risky API calls to classify applications as malware or benign. In particular, they label method calls obtaining sensitive information. For example, getSimSerialNumber() for getting SIM card serial number, sendDataMessage() for sending data, reading local file with File(), changing background image with WallpaperManager.setResource(), downloading files from Internet with openStream(), and getting latitude and longitude with getLatitude(), getLongitude()) calls. They checked the execution of the APIs using a virtual machine.

Schmidt *et al*. (2008) developed an anomaly detection approach for mobile devices. In particular, they collected feature data from mobile devices running the Symbian operating system. Examples of features range from simple (user inactivity, free RAM), medium (process count), and complex (CPU usage, and outbox SMS message count). By relying on native APIs supported by the OS, simple features can be collected. While relying on multiple APIs and heuristics, specific algorithms can also collected the medium and complex features. The features can detect anomaly activities due to malware. For example, if a malware sends SMS message due to a keystroke, then the number of processes increases (for sending each of the message), the amount of free RAM decreases, and the number of message count in the outbox increases.

Advantages: There are a wide variety of possible threats identified by this method and could be used to set a new standard of proper coding practices. Though the findings were not malware, they illustrate how easily a mobile application can be infiltrated due to poor coding practices or suspicious activity.

Disadvantages: It is very difficult to uncover malicious applications using this method because many poorly written mobile applications would be flagged as malicious causing false positive warnings. Therefore, it is important to note that this detection technique is more useful for determining potential risks and allow developers to close possible loopholes beforehand.

3.5 Permission Analysis Detection

Barrera *et al*. (2010) applied a self-organizing map-based learning algorithm to cluster different permission sets. Although the study relies on a set of general Android applications, it cannot be applied for detecting malware due to the observation that both malicious and benign applications may have similar types of permissions. Similarly, Porter *et al*. (2011) compared the permission system between Google Chrome and Google Android, and performed a subjective analysis for improving permission model in general for security and user level awareness. Nevertheless, a detection technique is still needed to identify malicious behaviors of malware, and our approach is complementary to these earlier efforts.

Schimidt *et al*. (2009) detected malware running on iOS platform. They analyzed executable code and performed machine learning (leveraging clustering algorithms) to identify features common in malicious applications. In particular, the features target the low level network and file system operations such as file copying and getting the host address.

Enck *et al.* (2009) developed a rule-based certification technique named Kirin that can check the presence of undesirable properties in applications suspected as malware. The approach starts from general functionality requirements and then analyzed whether required permissions can create conflicting operations that are used in malware operations. For example, an application should not have both RECEIVE_SMS and WRITE_SMS permission. The success of the certification process relies on the types of rules specified by the system and required.

Advantages: Because permissions are displayed to the user at install time, mobile users can determine whether an application's permissions relate to the purpose of the mobile application. Unknown or unused permissions are a great indicator of potential malicious activity. The permission-based detection technique is also intelligent enough to discern whether a mobile application's settings and properties align with its stated intention.

Disadvantages: Permissions can be maliciously inserted into an AndroidManifest.xml file after the user has installed the mobile application to the device Chin *et al*. (2011). Therefore, it is not ideal to rely on permission-based analysis as the sole detection technique.

3.6 Other Work

Nicolaou *et al*. (2013) explore the exponential rise of web browsing since 1999. With the rise of mobile devices, web browsing on mobile application devices will soon dominate web traffic. The authors also explore how companies and mobile developers will need to begin making the transition to mobile websites or mobile applications. More importantly, with the transition of web applications onto your mobile device, mobile users are susceptible to the many issues of web traffic and HTTP connections. In addition, network connectivity is not as stable in mobile applications as it is in desktop and laptop browsers. Therefore, users could experience many dropped requests. Furthermore, users would be required to keep an updated mobile device so that their machines can still efficiently process data from the applications.

Rastogi *et al*. (2013) developed a systematic framework named *DroidChameleon* for evaluation purposes. In the ten popular commercial anti-malware applications used, none of the applications was able to thwart attacks from modified malware. It appears that malware authors frequently use that polymorphism as an obfuscation technique to avoid detection by transforming the malware into different forms. Metamorphism is also used because it mutates the code so that it is removed but still executes the same behavior.

The author's findings were as follows:

- All the studied anti-malware products are vulnerable to common transformations.
- At least 43% signatures are not based on code- level artifacts.

- 0% of signatures do not require static analysis of bytecode. Only one of ten anti-malware tools was found to be using static analysis.
- Anti-malware tools have evolved towards content-based signatures over the past year (or since 2012).

Chin *et al*. (2011) analyzed 20 mobile applications; 60% of them contained exploitable security vulnerabilities. The authors used the *ComDroid* tool for analyzing the apps. Message passing vulnerabilities are dangerous because they leave the user susceptible to stolen passwords, emails, banking information, etc. Android's message passing system can be very vulnerable for non-savvy developers and unsuspecting end-users. Their findings are shown below:

- **Broadcast theft** – silently reading (or eavesdropping) the contents of a broadcast intent without actually interrupting or stealing the broadcast
- **Activity hijacking** – malicious activities are launched instead of the actual activity Service hijacking – malicious services intercept an intent that was meant to be sent to a legitimate service.
- **Special intents** – Intent uses a Uniform Resource Identifier (URI) reference and is able to add permissions for that intent without the end-users' knowledge.
- **Malicious broadcast injection** – malicious intents can propagate throughout the application by using commands in a broadcast intent
- **Malicious activity launch** – launching malicious activities implicitly or explicitly through the use of the Intent
- **Malicious service launch** – any application can start and bind to unprotected services

Chin *et al*. (2011) also explore "Intents", which can be used for intra-application and inter-application communication. There are four main components for the Intents: activities, services, broadcast receivers, and content providers. Intents can use message passing for three of the components: activities, services, and broadcast receivers. From a permissions level, services and activities must be declared in the *AndroidManifest.xml* in order to receive other intents. The message passing system uses the same "Intents" for transmitting data outside of the application to third party by the use of links or APIs or by passing information between views of a mobile application. The main red flag is the use of an explicit Intent that calls a developer specified recipient. Using the default Android platform, one would simply allow the Android application to use the correct calls to communicate with the appropriate intra-application Intent.

CHAPTER 4

Proposed KLD-Based Malware Detection

4.1 Overview

Instead of using heuristic-based approaches, such as Euclidean Distance or other measures, to compare an application with known sample applications, this work uses a formal method based on probabilistic models. It is assumed that all benign applications are generated by a hidden probabilistic model (say *M_benign*); and each malicious application is generated from a hidden probabilistic model (say *M_malicious*). The hypothesis is that the divergence between the models *M_benign* and *M_malicious* should be detectable. Then, Kullback-Leibler Divergence (KLD) is used to evaluate the divergence between the *M_benign* and *M_malicious* models.

Since the hidden probabilistic models are unknown, observable features generated from either model are used to approximate the model. For this purpose, features (f_1 to f_{10}) are extracted. It is further assumed that each application is generated by randomly sampling (f_1 to f_{10}) from the hidden model. Since the observed population is very limited, a smoothing technique is needed to avoid zero probability of feature observation.

The KLD computes the divergence between two given probability distributions. Let us assume that P and Q represent two probability distributions,

where $P = \{p_1, ..., p_n\}$ and $Q = \{q_1, ..., q_n\}$.

Then, the KLD is defined as follows (Cover & Thomas, 2006):

$$KLD\,(P, Q) = \sum_{i} pi * log_2\,(p_i / q_i) \quad \text{……… Equation } i$$

Here, the following two constraints are satisfied:

$$\sum_{i} pi = 1 \text{ ………………………………. Equation } ii$$

$$\sum_{i} qi = 1 \text{ ………………………………. Equation } iii$$

Cooper (2014a) starts with a hypothesis that the KLD between benign and malicious application for performing a specific operation should be relatively high. On the other hand, the KLD among benign applications performing the same operation should be relatively low. This approach uses different features to detect malicious applications. We define feature elements from the source code that relate to the primary purpose of the application's functionality. Using this information, we are able to determine suspicious malware applications. Our prototype implementation analyzes the source code of a suspected malware application in a secure environment without running the malware application on a mobile device.

4.2 Related Work

Our work is motivated by a number of works that apply the concept of Kullback-Leibler Divergence (KLD) as a measure to solve a number of problems from various domains including document's author identification (Bigi, 2003), masquerade attack detection for network security (Tapiador & Clark, 2010), outlier data value detection in wireless sensor network (Li & Wang, 2012), quality of non-object oriented software modularization (Sarkar, Rama, & Kak, 2007), and risk analysis in the domain of fuel cell study (Fukui, Sato, Mizusaki, & Numao, 2010).

Bigi (2003) applied KLD to identify authorship of documents. The approach first builds a model of each document author by aggregating documents generated by that author. It first develops a set of candidate models. Then, for a given document of unknown author, the approach finds the smallest KLD between a known model and the document. The model that is closest to the document is selected as the author. Similar to this work, we apply constant back-off smoothing technique to address the missing elements (or tokens derived from Java code of the malware).

Specifically, we compare the KLD between the code level features captured by population elements of an application and the expected population obtained from benign applications. The deviation, if exceeds a given threshold value, provides an indication of the presence of malware operation in an application.

Tapiador *et al.* (2010) detected masquerade attacks based on an anomaly-based technique that compares a given request with known normal request using KLD measure. In a masquerade attack, an attacker steals credentials of legitimate users and performs further malicious actions using the credentials. The KLD enables the detection of padding in command sequences independent of the length and position in a block of request. In contrast, we apply KLD to detect malware activities based on code level features.

Li *et al.* (2012) applied differential KLD to detect anomalous data value in wireless sensor networks. The network is divided into clusters. In each cluster, the sensors remain physically close to each other and sense similar values. The outlier values are detected using KLD. Sarkar *et al.* (2007) applied information theoretic measure including KLD to measure the quality of modularization in non-object oriented software systems. Fukui *et al.* (2010) measured the similarity of events based on KLD and applied it in the domain of fuel-cell study.

4.3 KLD-Based Approach

KLD is not a distance; it is a divergence between two probability distributions that are asymmetric in nature. All of the literature work that we studied employs KLD to detect anomaly or security issues; none has compared the KLD value with any distance metrics, such as Euclidean or cosine. We consider SMS message sending as a case study for this work. For a given SMS functionality, we identify the source code responsible for invoking it along with source of inputs. The malicious applications typically do not accept inputs from users and mostly supplies static values during the invocation of method calls. On the other hand, the legitimate applications, while performing the same functionality, rely on user-supplied inputs. This makes a difference between the behavior of a malicious and a legitimate application. KLD can be a suitable measure to understand it as an automated process; hence, it can be used to detect malicious applications.

To compute the KLD between two population sets (or probability distributions), we need to define a set of elements relevant to the specific SMS operations and obtain a collection of legitimate application samples to build *P* set. Now, given that we have a new application (*Q*), we can then find how divergent is the new application compared to the *P* set with respect to SMS operation to label the new application as malware or good application.

However, the challenge here is computing the term $p_i * log_2 (p_i/q_i)$. It can be rewritten as subtraction of two terms: $p_i * log_2 (p_i) - p_i * log_2 (q_i)$. While we compute KLD (*P*, *Q*), if either p_i or q_i is zero (no occurrence of probability is observed from applications), then the term becomes infinite, which results in KLD (*P*, *Q*) to be zero. To address this issue, we propose to apply a well-known smoothing technique known as constant back-off (Bigi *et al*. 2003). Here, all zero probability values in both *P* and *Q* are substituted with a very negligible constant value and all the non-zero values are equally subtracted with the same constant amount proportionally so that Equations *(ii)* and *(iii)* are still satisfied. This simple step results in two smoothed probability distributions denoted as *P′* (derived from *P*) and *Q′* (derived from *Q*). So, we essentially compute KLD (*P′*, *Q′*) to avoid infinity problem instead of KLD (*P*, *Q*).

4.4 Elements of Population

Table 4 shows the list of 10 elements (f_1-f_{10}) that we consider in building the population of elements and compute their occurrence probabilities from Android applications. Among them, the first five elements are commonly found to be legitimate ways of sending (f_1-f_4) or receiving (f_5) SMS messages (based on extensive survey and reports from related work).

For example, f_1 represents sending SMS message by creating a visual Action window where a user can provide message and destination number for sending a message. At the Java source code level, we then look for the following sequence of method call invocation: *setContentView()* that allows for displaying of an Action window on the screen, one or more call of *getText()* to access the current values of input from GUIs passed as SMS sending operation argument, and the presence of the event handler that invokes the text retrieval operation and SMS sending operation. Good applications send SMS messages using variables as part of their argument of the respective method (*sendTextMessage()* and variable argument) as shown in f_2. An application may rely on creating an Intent object and store SMS messages as part of the method call argument (*putExtra*) followed by launching the Activity (f_3). The *Uri.parse()* method can be invoked as well for sending messages (f_4).

Table 4: A Description of SMS Operational Element for Building Population Set

Type	Name	Description	Signature Sequence
Benign	f_1	SMS message is sent with visual input, through even handler method	*setContentView()*, *getText()*,EventHandler
	f_2	SmsManager object is created, sendTxtMsg is invoked, variable argument is present	*SmsManager* class, *sendTextMessage()*, variable argument
	f_3	Create Intent object, write SMS message, variable argument message, start Activity	*Intent* class, *putExtra()*, variable SMS message or phone number, *startActivity()*
	f_4	Start activity with “smsto:” string in Uri.parse method and variable parameter for SMS message	*startActivity()*,*Uri.parse()*, presence of “smsto:”, variable argument in *Uri.parse()*
	f_5	Message delivery or receiving status is notified	Presence of *Toast.makeToast()* with SMS keyword or presence of exception handling for message sending or receiving error code
Malicious	f_6	SMS message is sent without input from visual interfaces, and in presence or absence of event handler method	*SmsManager*, no *getText()*, no event handler for the SMS sending operation
	f_7	SmsManager object is created, sendTxtMsg is invoked, constant argument present	*SmsManager*, *sendTextMessage()*, constant SMS message or phone number
	f_8	Using intent object, putting SMS body, and constant argument message	*Intent* class, *putExtra()*, constant argument for SMS message or phone number
	f_9	Start activity with “smsto:” string in Uri.parse method and constant parameter representing SMS message	*startActivity()*,*Uri.parse()*, presence of "smsto:" string, constant argument for message or phone number in *Uri.parse()*
	f_{10}	Message delivery or receiving status is not notified	No presence of *Toast.makeToast()*, and no exception handling for message sending or receiving error code

Finally, a legitimate application notifies users about the receiving of any incoming message that could be due to the failure of sending an earlier message from a phone, or receiving a message from new source. In this case, we check the presence of viewable Activity window and explicit code for handling the status (f_5). More specifically, we look for the presence of the *Toast.makeToast()*method invocation with short message containing the keyword "SMS" and exception handling code that does not suppress the SMS sending error message or receiving information. Similarly, the last five elements (f_6 - f_{10}) represent malicious ways of sending (f_6 - f_9) or receiving (f_{10}) SMS messages. For example, one way of sending SMS would be not to display any Activity window (no *setContentView()* call), no extraction of inputs (no *getText()* call), and no event handler method invocation where SMS sending is taking place. Similarly, we look for the sequence of the absence of other API sequence to identify these elements.

4.5 Back-off Smoothing

For a given set of legitimate Android applications, we compute the *P* set containing the occurrence of f_1 - f_{10} and the probability distribution. Then, given a new Android application we identify the *Q* set containing the occurrence probability of f_1 - f_{10} and see how distant the two sets are to understand the closeness. The less divergence we find, the closer the two sets, hence *Q* is identified to be good application with respect to the specific SMS operation. On the other hand, if the divergence is very high, then we label *Q* as malware. As one or more elements from *P* and *Q* may not have any occurrence (zero probability), they need to be smoothed (already discussed in Section 4.3).

4.6 Evaluation using Data Set

We evaluated our approach as follows: first we gather a set of legitimate Android applications downloaded randomly from the web, where each of the applications contains Java code for performing SMS functionalities. To ensure diversity in the test applications, selected applications rely on different known techniques of sending or receiving SMS messages (*SmsManager*, *putExtra* for *Intent*, *Uri.parse*). We have total 17 applications in our data set to construct the *P* set. The *P* set applications are shown in table 5 along with the occurrence (frequency) of their population elements. The last row of Table 5 shows the combined frequency of all population elements (the *P* set).

For the *Q* set, we use one application that we are comparing with the *P* set. Table 6 shows the KLD between *P* and each of the malware (*Q*). We show the results in terms of *P′* and *Q′* (after smoothing the sets). The value ranges between 12.47 and 17.25, which provides a basis of threshold values for consideration to detect new malware samples for their benign or maliciousness.

Table 5: Occurrence of Elements in the *P* Set

Application	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}
SMS_Android-Build-In-SMS-Application-Example	0	0	0	0	0	0	0	1	0	0
SMS_Android-Send-SMS-Example	1	1	0	0	1	0	0	0	0	0
SMS_AndroidSMSExample_1	1	1	0	0	1	0	0	0	0	0
SMS_AndroidSMSExample_2	0	0	0	0	1	0	0	0	0	0
SMS_apriorit_SecureMessages	0	0	0	0	0	0	0	0	0	0
SMS_Cloud SMS	0	1	0	0	0	0	0	0	0	0
SMS_Free SMS India	0	1	0	0	7	0	2	0	0	0
SMS_GO SMS Pro	0	2	0	0	9	0	1	0	1	0
SMS_Handcent SMS	0	0	0	0	0	0	0	0	0	0
SMS_javacodegeeks_AndroidSMSExample_1	1	1	0	0	1	0	0	0	0	0
SMS_MightyText.src	0	4	0	0	4	1	0	0	0	0
SMS_mkyong-Android-Send-SMS-Example	1	1	0	0	1	0	0	0	0	0
SMS_mkyoung-Android-Build-In-SMS-Application-Example	0	0	0	0	0	0	0	1	0	0
SMS_msatpathy_SMSTest	1	2	0	0	3	0	0	1	0	0
SMS_Ninja SMS	0	0	0	0	0	1	0	0	0	0
SMS_SecureMessages	0	0	0	0	0	0	0	0	0	0
SMS_SMSTest	1	2	0	0	3	0	0	1	0	0
Total	**6**	**16**	**0**	**0**	**31**	**2**	**3**	**4**	**1**	**0**

Table 6: KLD Between Good (*P′*) and Malware (*Q′*) Applications

Malware Application Name (*Q′*)	**KLD (*P′*, *Q′*)**
AndroidDogwar	16.93
DroidDeluxe	17.25
DroidDreamlight2	17.25
DroidKungFu2A	12.47
DroidSlasher_1_1.0.1(GoldDreamA)	12.47
HippoSMS	12.47
Lovetrap	12.47
Spitmo	16.38
Zitmo	17.25
zj_NinjaChicken_other	12.47

To further complement our evaluation, we randomly computed the KLD between the trained samples (*P*) and another new set of good applications performing SMS operations. Table 7 shows a snapshot of the obtained KLD values showing the divergence between good and good applications ranges between 5.12 and 17.25. Our experiment led to one false-positive warning. Considering the threshold values obtained from malware analysis in Table 2 (12.47-17.25), we find that Virtual Table Tennis 3D application is labeled as malware. The other nine applications are considered as benign. Thus, KLD can be a suitable measure to identify malware and benign applications for SMS operations if the threshold of divergence is considered carefully.

Table 7: KLD Between Good (*P′*) and Good (*Q′*) Applications

Good Application Name (*Q′*)	**KLD (*P′*, *Q′*)**
Barcode Scanner	10.81
FxCamera	9.97
Huffington Post	11.82
My Currency – Converter	8.77
Skype	7.23
To-Do Calendar Planner	5.12
Viber	9.42
Virtual Table Tennis 3D	17.25
WhatsApp	12.32
YouTube	8.65

4.7 Discussion

Here, we will demonstrate how another metric-based approach will give less accurate results when compared to applying our KLD-based approach.

The metric-based approach is defined as follows:

Malicious: $Sum(f_6\text{-}f_{10}) > Sum(f_1\text{-}f_5)$ ……… Equation *iv*

Benign: $Sum(f_1\text{-}f_5) \geq Sum(f_6\text{-}f_{10})$ ……… Equation *v*

Table 8 compares the sum of the benign, $Sum(f_1\text{-}f_5)$, elements with the sum of the malicious, $Sum(f_6\text{-}f_{10})$, elements. We see that this metric-based approach does show that the total sum for all benign elements is greater than all of the malicious elements.

Table 8: Sum of Elements in the *P* Set

Application	$Sum(f_1\text{-}f_5)$	$Sum(f_6\text{-}f_{10})$
SMS_Android-Build-In-SMS-Application-Example	0	1
SMS_Android-Send-SMS-Example	3	0
SMS_AndroidSMSExample_1	3	0
SMS_AndroidSMSExample_2	1	0
SMS_apriorit_SecureMessages	0	0
SMS_Cloud SMS	1	0
SMS_Free SMS India	8	2
SMS_GO SMS Pro	11	2
SMS_Handcent SMS	0	0
SMS_javacodegeeks_AndroidSMSExample_1	3	0
SMS_MightyText.src	8	1
SMS_mkyong-Android-Send-SMS-Example	3	0
SMS_mkyoung-Android-Build-In-SMS-Application-Example	0	1
SMS_msatpathy_SMSTest	6	1
SMS_Ninja SMS	0	1
SMS_SecureMessages	0	0
SMS_SMSTest	6	1
Total	**53**	**10**

When we compare the sums for each of the applications in the *P* set, we also see that most of the applications have a higher *Sum(f_1-f_5)* value that indicates the application is harmless. However, we also see in Table 9 that *Sum(f_1-f_5)* is not always greater than *Sum(f_6-f_{10})*. Three of the applications had a *Sum(f_1-f_5)* value that was less than the *Sum(f_6-f_{10})*. Our KLD-Based approach shows that all of the applications in the *P* set were within the benign threshold of values. Therefore, our approach gives more accurate results.

Table 9: Accuracy of Metric-Based Approach for the *P* Set

P set	Correct	14/17
	Incorrect	3/17

Next, we tested the metric-based approach on the suspected malicious applications in the *Q* set. In Table 10, we see the occurrence of elements in our first *Q* set that represent the suspected malicious applications. Table 11 compares the sum of the benign, *Sum(f_1-f_5),* elements with the sum of the malicious, *Sum(f_6-f_{10}),* elements. In Table 12, we see that the accuracy of the metric-based approach continues to decrease even though it still holds true to our hypothesis. As shown in Table 6, our KLD-Based approach shows that all of the applications in the malicious *Q* set fall within the threshold of values.

Table 10: Occurrence of Elements in the Malicious *Q* Set

Application	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}
AndroidDogwar	0	0	0	0	0	0	2	0	0	0
DroidDeluxe	0	0	0	0	0	1	0	0	0	0
DroidDreamlight2	0	0	0	0	0	1	0	0	0	0
DroidKungFu2A	0	0	0	0	0	0	0	0	0	0
DroidSlasher_1_1.0.1(GoldDreamA)	0	1	0	0	0	1	0	0	0	0
HippoSMS	0	0	0	0	0	1	0	0	0	0
Lovetrap	0	1	0	0	0	1	0	0	0	0
Spitmo	0	0	0	0	0	1	1	0	0	0
Zitmo	0	0	0	0	0	1	0	0	0	0
zj_NinjaChicken_other	0	1	0	0	0	1	0	0	0	0

Table 11: Sum of Elements in the Malicious *Q* Set

Application	$Sum(f_1\text{-}f_5)$	$Sum(f_6\text{-}f_{10})$
AndroidDogwar	0	2
DroidDeluxe	0	1
DroidDreamlight2	0	1
DroidKungFu2A	0	0
DroidSlasher_1_1.0.1(GoldDreamA)	1	1
HippoSMS	0	1
Lovetrap	1	1
Spitmo	0	2
Zitmo	0	1
zj_NinjaChicken_other	1	1

Table 12: Accuracy of Metric-Based Approach for the Malicious *Q* Set

Malicious *Q* set	Correct	6/10
	Incorrect	4/10

Lastly, we tested the metric-based approach on the suspected benign applications in the other *Q* set. In Table 13, we see the occurrence of elements in our second *Q* set that represent the suspected benign applications. Table 14 compares the sum of the benign, $Sum(f_1\text{-}f_5)$, elements with the sum of the malicious, $Sum(f_6\text{-}f_{10})$, elements. In Table 15, we see that the accuracy of the metric-based approach is poor in comparison to our KLD-Based approach. We received only one false-positive warning for the Virtual Table Tennis 3D application.

Table 13: Occurrence of Elements in the Benign *Q* Set

Application	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}
Barcode Scanner	0	0	0	0	0	1	0	0	0	0
FxCamera	0	0	0	0	0	0	0	0	0	0
Huffington Post	0	0	0	0	0	0	0	0	0	0
My Currency – Converter	0	0	0	0	0	0	0	0	0	0
Skype	0	0	0	0	0	0	0	0	0	0
To-Do Calendar Planner	1	0	0	0	0	0	0	0	0	0
Viber	0	1	0	0	0	0	0	0	0	0
Virtual Table Tennis 3D	0	0	0	0	0	1	0	0	0	0
WhatsApp	0	0	0	0	0	0	0	0	0	0
YouTube	0	0	0	0	0	1	0	0	0	0

Table 14: Sum of Elements in the Benign *Q* Set

Application	***Sum(f_1-f_5)***	***Sum(f_6-f_{10})***
Barcode Scanner	0	1
FxCamera	0	0
Huffington Post	0	0
My Currency – Converter	0	0
Skype	0	0
To-Do Calendar Planner	1	0
Viber	1	0
Virtual Table Tennis 3D	0	1
WhatsApp	0	0
YouTube	0	1

Table 15: Accuracy of Metric-Based Approach for the Benign *Q* Set

Benign *Q* set	Correct	7/10
	Incorrect	3/10

CHAPTER 5

Application Implementation

5.1 Overview

We implemented a prototype application to demonstrate the functionality of our proposed KLD-Based approach. There are three stages:

(i) decompiling the APK file into readable source code,
(ii) analyzing the source code using our prototype Java class, and
(iii) determining the status of a mobile application as good or bad by reviewing the data set.

Our approach is partially automated, and the *P* set is calculated beforehand from a set of known good samples. In this section, we apply Kullback-Leibler Divergence (KLD) to differentiate malware and legitimate application behavior for SMS message functionality. We also explain the decompiling process in detail using screenshots from our GUI. Note that this application implementation is in progress. This chapter presents the work completed as of March 17, 2014.

5.2 Decompiling the APK

As mentioned above, we first convert the APK file into readable source code. The prototype application can automate the decompiling process of the APK file before computing the occurrence probability, which is the second stage of our KLD-based approach. Figure 17 displays the GUI of the decompiling of the APK file. Here, we have three steps: (a) to choose the APK file, (b) to convert the APK file to a jar file, and (c) to extract the source code from the jar file. The white space will serve as a logger to update the user on their selections and the decompiling process. Using the file browser, Figure 18 shows how to browse to the desired location and select the APK file after clicking the *Choose File* button (step (a)).

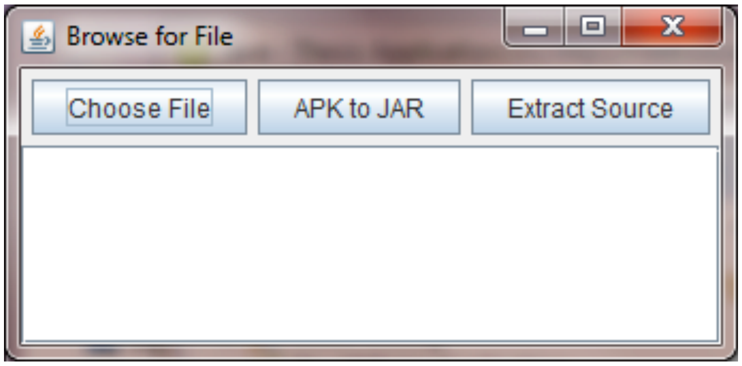


Figure 17: GUI of application that decompiles the APK

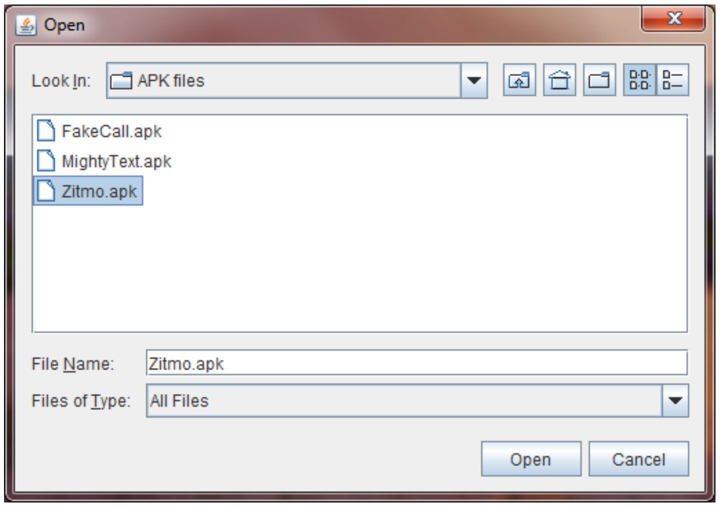
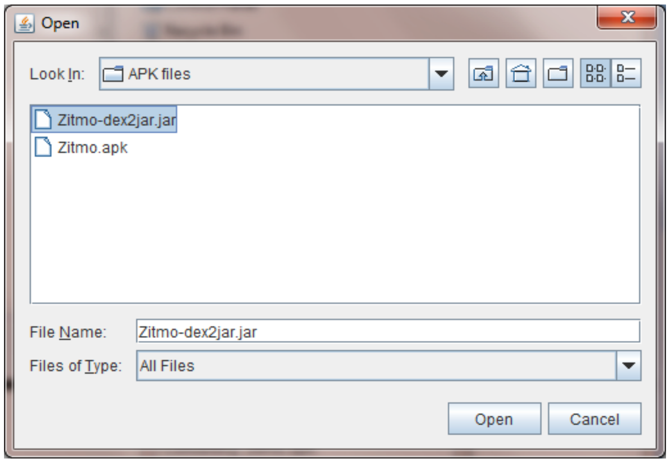


Figure 18: Demonstration of selecting an APK to decompile

Now, we convert the APK file to a jar file. To do that, we select the *APK to JAR* button (step (b)). In order to decompile the APK file, we use the command lines from the open source *dex2jar* utility ("dex2jar", 2013). *Dex2jar* is a very useful tool for extracting source code of mobile applications. It is also capable of maintaining the integrity of the folder structure. Once the process is complete, we can go back to the file browser and see that *Zitmo-dex2jar.jar* has been created, as shown in Figure 19.

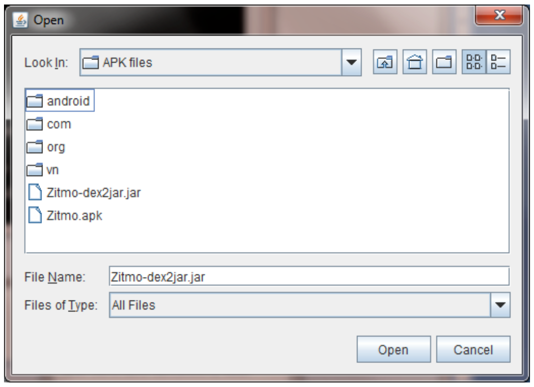


Zitmo-dex2jar.jar
Zitmo.apk

File Name: Zitmo-dex2jar.jar
Files of Type: All Files

Figure 19: Verification that APK file was converted to jar file

Now, we initiate the *Extract Source* button (step (c)). The contents of *Zitmo-dex2jar.jar* are extracted, and the Java class files are generated. Figure 20 shows a screenshot where the jar file is converted to Java class files. The next step is to convert all *.class* files into *.java* files using the JD-GUI ("Java Decompiler", 2013).



android
com
org
vn
Zitmo-dex2jar.jar
Zitmo.apk

File Name: Zitmo-dex2jar.jar
Files of Type: All Files

Figure 20: Verification of readable source code

5.3 Analysis of the Source Code

We implemented a prototype Java class, *TestAndroidKLD.java* that analyzes the decompiled APK files at the Java code level and can compute the occurrence probability of elements of interest (f_1 - f_{10}) from java source files (See Appendix A for *TestAndroidKLD.java* source code).

TestAndroidKLD.java has a method, *scanJavaFile(File file)*, that checks the main *Zitmo* directory and each of its subdirectories for java files. Each of those java files is then scanned and checked for the occurrence of the elements of population. For example, f_1, explained in Table 4, refers to an SMS message being sent with visual input. *scanJavaFile(File file)* checks for the presence of *setContentView()*; if it is present, that is an indication of a benign action.

Table 16: Output of Method Call Occurrence for the *P* Set (Part 1)

Application	Activity	View	setContentView()	getText()	EventHandler	SmsManager
SMS_Android-Build-In-SMS-Application-Example	1	0	1	0	1	0
SMS_Android-Send-SMS-Example	1	0	1	1	1	1
SMS_AndroidSMSExample_1	1	0	1	1	1	1
SMS_AndroidSMSExample_2	1	0	1	0	1	0
SMS_apriorit_SecureMessages	1	0	1	0	1	0
SMS_Cloud SMS	15	20	0	2	17	5
SMS_Free SMS India	11	0	0	2	5	5
SMS_GO SMS Pro	20	52	0	12	599	24
SMS_Handcent SMS	4	17	0	11	404	28
SMS_javacodegeeks_AndroidSMSExample_1	1	0	1	1	1	1
SMS_MightyText.src	9	0	0	0	0	16
SMS_mkyong-Android-Send-SMS-Example	1	0	1	1	1	1
SMS_mkyoung-Android-Build-In-SMS-Application-Example	1	0	1	0	1	0
SMS_msatpathy_SMSTest	1	0	1	1	1	6
SMS_Ninja SMS	9	22	0	0	16	0
SMS_SecureMessages	1	0	1	0	1	0
SMS_SMSTest	1	0	1	1	1	6
Total	**79**	**111**	**11**	**33**	**1052**	**94**

Table 17: Output of Method Call Occurrence for the *P* Set (Part 2)

Application	sendTextMessage	Const arg	Var arg	Intent	putExtra (sms_body)
SMS_Android-Build-In-SMS-Application-Example	0	0	0	1	1
SMS_Android-Send-SMS-Example	1	0	1	0	0
SMS_AndroidSMSExample_1	1	0	1	0	0
SMS_AndroidSMSExample_2	0	0	0	1	0
SMS_apriorit_SecureMessages	0	0	0	0	0
SMS_Cloud SMS	1	0	1	0	0
SMS_Free SMS India	3	2	1	0	0
SMS_GO SMS Pro	3	1	2	0	0
SMS_Handcent SMS	0	0	0	0	0
SMS_javacodegeeks_AndroidSMSExample_1	1	0	1	0	0
SMS_MightyText.src	4	0	4	0	0
SMS_mkyong-Android-Send-SMS-Example	1	0	1	0	0
SMS_mkyoung-Android-Build-In-SMS-Application-Example	0	0	0	1	1
SMS_msatpathy_SMSTest	2	0	2	1	1
SMS_Ninja SMS	0	0	0	0	0
SMS_SecureMessages	0	0	0	0	0
SMS_SMSTest	2	0	2	1	1
Total	**19**	**3**	**16**	**5**	**4**

The *P* set computation requires adding up of all the f_i counts from all sample applications. A counter keeps track of each element's occurrence. While *TestAndroidKLD.java* scans the source code, it creates and writes all data to a CSV file. Tables 16 and 17 show the generated outputs that have been saved into the *KLD_Results.csv* file.

5.4 Reviewing the Obtained Results

The first two steps, mentioned in Sections 6.2 and 6.3, are repeated for multiple mobile applications that have the same type of functionality. By evaluating a large number of applications, we are able to prevent KLD values from being skewed too heavily in one direction. Using the values generated in the CSV file, we can compare the known good and malicious KLD values. First, we calculate the final tabulation for each element in the population set, as shown in Table 5. Then, we are able to calculate its KLD value and determine if it has malicious operations.

5.5 Performance

Currently, our KLD-based approach is being executed as a desktop application. The average time to build our *P* set was a total of 0.146 seconds. The average time to build our malicious *Q* set was a total of 0.153 seconds. The average time to build our benign *Q* set was a total of 0.113 seconds. These average times are considered to be fairly efficient since they do not require an excessive amount of time to analyze the chosen applications and generate the CSV file that tracks the occurrence of the population elements. This performance would change once transitioning from an offline desktop application to a running service on a mobile device.

5.6 Deployment

The offline analysis of scanning Android applications does not require an Internet connection. However, as malicious activities continue to evolve, the *P* set would require updating. Our initial intention for the deployment phase was to distribute the approach as a running service on the Android device. After careful consideration, we realized that the large variety of device hardware would affect the consistency of implementation and efficiency. The added constraint of declining battery power and device lifespan would deter users from running our service on their devices. In our future research, we plan to deploy our approach as a service in the cloud environment in order to maximize performance.

CHAPTER 6

Dissemination of Research Findings

Android Malware Detection Using Kullback-Leibler Divergence
Vanessa N. Cooper, Hisham M. Haddad, and Hossain Shahriar.
Work in Progress.

Abstract
Many recent reports suggest that malware applications cause high billing to victims by sending and receiving of hidden SMS messages Given that, there is a need to develop necessary technique to identify malicious SMS operations as well as differentiate between good and bad SMS operations within applications. In this paper, we apply Kullback-Leibler Divergence (KLD) as a distance to identify the difference between good and malicious SMS operations. We develop a set of elements that represent sending or receiving of SMS messages both legitimately and maliciously. Then, we compare the divergence of the trained set of elements. Our evaluation shows that the divergence between good and bad applications remains significantly high, whereas between two applications performing the same SMS operations remain low. We evaluate the proposed KLD-based concept for identifying a set of malware applications. The initial results show that our approach can identify all known malware and has less false positive warning.

Development and Mitigation of Malicious Android Applications
Vanessa N. Cooper, Hossain Shahriar, and Hisham M. Haddad.
Book Chapter. Contribution to the book titled *Handbook of Research on Digital Crime, Cyberspace Security, and Information Assurance*, Edited by Maria Manuela Cruz-Cunha, Polytechnic Institute of Cávado and Ave, Portugal. Published by IGI Global, Spring 2014.

Abstract
As mobile applications are being developed at a faster pace, the security aspect of user information is being neglected. A compromised smartphone can inflict severe damage to both users and the cellular service provider. Malware on a smartphone can make the phone partially or fully unusable; cause unwanted billing; steal private information; or infect every name in a user's phonebook. A solid understanding of the characteristics of malware is the beginning step to prevent much of the unwanted consequences. This chapter is intended to provide an overview of security threats posed by Android malware. In particular, we focus on the characteristics commonly found in malware applications and understand the code level features that allow us to detect the malicious signatures. We also discuss some common defense techniques to mitigate the impact of malware applications.

A Survey of Android Malware Characteristics and Mitigation Techniques
Vanessa N. Cooper, Hossain Shahriar, and Hisham M. Haddad.
Conference Proceedings. *Proceedings of the IEEE International Conference on Information Technology: New Generations (ITNG 2014)*, Las Vegas, Nevada, April 2014.

Abstract
As mobile applications are being developed at a faster pace, the security aspect of is being neglected. A solid understanding of the characteristics of malware is the first step to preventing many unwanted consequences. This paper provides an overview of popular security threats posed by Android malware. In particular, we focus on the characteristics commonly found in malware applications and understand the code level features that can enable detection techniques. We also discuss some common defense techniques to mitigate the impact of malware applications.

Android Malware Detection Based on Kullback-Leibler Divergence
Vanessa N. Cooper.
Invited Student Research Abstract to the ACM SAC 2014 Student Research Competition (SRC) Program. *Proceedings of the ACM-SIGAPP Conference on Applied Computing (SAC 2014)*, Gyeongju, Korea, March 2014, pp. 1695-1696.

Abstract
A recent study shows that more than 50% of mobile devices running Google's Android mobile operating system (OS) have unpatched vulnerabilities, opening them up to malicious applications and malware attacks. The starting point of becoming a potential victim due to malware is to allow the installation of applications without knowing in advance the operations that an application can perform. In particular, many recent reports suggest that malware applications caused unwanted billing by sending SMS messages to premium numbers without the knowledge of the victim [1,2]. Given that, there is a need for techniques to identify malicious behaviors of applications before installing them.

Study of Agility in Mobile Application Development
Vanessa N. Cooper and Hisham M. Haddad.
Conference Proceedings. *Proceedings of the International Conference on Software Engineering Research and Practice (SERP 2013)*, Las Vegas, Nevada, July 2013, pp. 384-390.

Abstract
Not only has Agility infiltrated enterprise and consumer mobile application development, but it has also become an integral part of most IT departments and the standard for younger generation developers. Despite the numerous benefits of Agile development, software developers often find out that there are also several pitfalls to avoid during mobile application development. In this study, we will explore the potential pitfalls of incorporating agility into the development of mobile applications. The motivation behind this work stems from professional and personal experience of the primary author.

CHAPTER 7

Conclusion and Future Work

7.1 Conclusion

This thesis provides an overview of security threats posed by Android malware. We discuss the overall structure of the Android OS and how its security features attempt to prevent malware attacks. We also discuss the details of Android's privacy features and overall architecture. We discuss three different types of malware (grayware, spyware, and malware) and how they affect Android security. In particular, we focus on the characteristics commonly found in malware applications and understand the code level features that allow us to detect the malicious signatures. In addition, our examination of the code level demonstrates the likelihood of an Android application's malicious activities by those specific method signatures.

We also discuss some common defense techniques to mitigate the impact of malware applications. Those defense techniques are as follows: sandboxing, machine learning algorithms, decompiler-based static analysis, and secure software architecture for Android applications. A secure Android operating system and better coding practices will greatly reduce the possibilities of Android malware. These defense techniques enhance the security of the Android platform and deployed applications. We discuss both the advantages and disadvantages of each of these techniques.

In this thesis, we propose to choose the Kullback-Liebler Divergence (KLD) as a measurement to differentiate between legitimate and malicious application behavior at source code level. The methodology builds probability distributions from the available source code of an application performing a specific functionality. We show some highlights of choosing possible elements of interest that can be useful to differentiate between a benign and malicious application behavior. Then, we apply the KLD measure to show that the difference between a legitimate and malicious application is infinite, whereas the difference between two legitimate applications is close to zero.

We also develop a prototype application that can partially automate the decompiling process of the APK file before computing the occurrence probability. We address the detection of malicious SMS operations within malware based on a set of proposed elements that can be used to build population for computing KLD. Furthermore, to address the elements having zero probability, we propose to apply constant back-off smoothing technique. We evaluated our approach using a set of known good applications to build one population set followed by a set of malware applications obtained from the web. The results show that KLD between good and malware applications are high and ranges from 12.47 to 17.25. In addition, we also measured the KLD between the trained applications and another set of good applications, and found that the KLD between good and good applications may range from 5.12 to 17.25. Based on the study of Android malware, we conclude that there should be a pair of threshold values for identifying malware applications using KLD. In our evaluation, only one good application has been labeled as malware (false positive).

We believe that the application of KLD is very practical and simply deduces the elements of population for each functionality type into a threshold of values (which can identify a simple pass/fail). False positives were also investigated to ensure that the range of values is correct for both benign and malignant applications. We conclude that our application implementation of the KLD method accounts for more mitigation techniques. By examining the Android Manifest file (permission analysis), we can determine the intended functionality of each application and automatically generate its elements of population from a predetermined list. Using that information, our static analysis of the source code will yield more accurate results by checking for obfuscated code. Also, this is being done in an isolated environment (sandboxing) and the application is not being dynamically executed which greatly reduces risk of infection.

7.2 Future Work

Our future research includes theoretical and implementation goals. On the theoretical side, our goals are: (i) choosing an appropriate smoothing technique to practically compute KLD, when one of the elements occurrence probability is found to be zero, (ii) finding more elements of population to cover more cases, (iii) documenting all possible known code patterns for performing specific functionality of interests that are common in malware applications, and (iv) validating our hypothesis using a larger collection of sample Android applications consisting of both legitimate and malicious behaviors.

On the implementation side, the conditions that we used to check the occurrence of population elements may not be exhaustive and accurate for all types of malware activities. However, we plan to create an interface where the end user can specify the population elements based on the activity. Our future goal includes automating the process for decompiling the APK file and analyzing the source code. We also plan to research the possibilities of deploying the application as a service in the cloud environment

Appendix A: TestAndroidKLD.java Source Code

```
import java.io.BufferedReader;
import java.io.DataInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.StringTokenizer;


public class TestAndroidKLD {

public static int scanJavaFile (File file){

      int store=0;

      try{

      FileInputStream fstream = new
      FileInputStream(file.getAbsolutePath().toString());

      DataInputStream in = new DataInputStream(fstream);

      BufferedReader br = new BufferedReader(new InputStreamReader(in));

      String str;

      while ((str = br.readLine()) != null) {

      if ((str.indexOf(" Activity") > 0) && str.indexOf(" class ") > 0 ){
            result[0]++;
            store=1;
      }

      if ((str.indexOf(" View") > 0) && str.indexOf(" class ") > 0 ){
            result[1]++;
            store=2;
      }

      if ((str.indexOf("setContentView") > 0) &&
      (str.indexOf("R.layout.main") > 0)){
            result[2]++;
            System.out.println ("setContentView() stmt: " + str);
            store=3;
      }

      if ((str.indexOf("getText().toString()") > 0) && store==5){
            result[3]++;
            System.out.println ("getText() call within event handler stmt:
" + str);
            store=4;
      if ((str.indexOf("public void on") > 0) && (str.indexOf("View ") >
0)
                                    && !str.contains(",")){

            result[4]++;
            System.out.println ("event handler stmt: " + str);
            store=5;
      }

      if (str.indexOf("SmsManager") > 0 && !(str.indexOf("import") >= 0)){
            result [5]++;
            System.out.println ("SmsManager stmt: "+ str);
            store=6;
      }

      if (str.indexOf("sendTextMessage") > 0 ){
            result [6]++;
            System.out.println ("sendTextMessage () stmt: "+ str);
            store=7;
      }

      if ((str.indexOf("sendTextMessage") > 0 ) && str.contains("\"")){
            result [7]++;
            System.out.println ("constant argument in sendTextMessage ():
"+ str);
            store=8;
      }

      if ((str.indexOf("sendTextMessage") > 0 ) && !str.contains("\"")){
            result [8]++;
            System.out.println ("variable argument in sendTextMessage ():
"+ str);
            store=9;
      }

      if (str.indexOf("Intent") > 0 && str.indexOf("new ") > 0 &&
str.indexOf("=") > 0 && str.indexOf("Intent.ACTION_VIEW") > 0 ){

            result [9]++; //intent++;
            System.out.println ("Intent creation stmt: "+ str);
            store=10;
}

if (str.indexOf("sendIntent.putExtra") > 0 && str.indexOf("sms_body") > 0
){
      result [10]++;
      store=11;
      System.out.println ("sms using Intent stmt: "+ str);
      if (!str.contains("\"")){
            System.out.println ("variable sms stmt: "+ str);
            result [11]++;
      }
}
```

```
      if (str.indexOf("sendIntent.putExtra") > 0 &&
str.indexOf("sms_body") > 0 ){
            result [10]++;
            store=11;
            System.out.println ("sms using Intent stmt: "+ str);
            if (!str.contains("\"")){
                        System.out.println ("variable sms stmt: "+ str);
                        result [11]++;
            }
            if (str.contains("\"")){
                        System.out.println ("const sms stmt: "+ str);
                        result [12]++;
            }
      }


      if (str.indexOf("startActivity") > 0 ){
            result [13]++;
            store=12;
            if (str.indexOf("Uri.parse") > 0){
                  result [14]++;

                  System.out.println ("Activity with Uri stmt: "+ str);

                  if (str.indexOf("smsto:") > 0){
                        result [15]++;

                        System.out.println ("Activity with Uri and smsto
stmt: "+ str);

                        String msg =
str.substring(str.indexOf("smsto:")+2, str.length()-1);
                        if (!msg.contains("\"")){
                              result [16]++;
                              System.out.println ("Activity with Uri,
smsto with variable msg: "+ str);
                        }
                         (msg.contains("\"")){
                              result [17]++;
                              System.out.println ("Activity with Uri,
smsto with const msg: "+ str);
                        }
                  }
            }
      }

      if (str.indexOf("Toast.makeText") > 0 && str.indexOf("SMS") >0 ){
            System.out.println ("Toast.makeText stmt: "+ str);
            result [18]++;
      }
```

```
            if (str.indexOf("RESULT") > 0 && str.indexOf("SMS") >0 ){
                  System.out.println ("Result notification for SmsManager
stmt: "+ str);
                  result [19]++;
            }

      }
      in.close();

      }catch (Exception e){ //Catch exception if any
                  System.err.println("Error: " + e.getMessage());
            }
            return store;
      }

      public static void walk( String path ) {

        File root = new File( path );
        File[] list = root.listFiles();

        if (list == null) return;

        for (File f : list) {
            if (f.isDirectory()) {

                walk(f.getAbsolutePath());
            }
            if (f.getName().endsWith("java")){
                  fcount++;
            }

        }
    }

      public static int fcount=0, dcount =0, imgCount=0;
      public static int activityCount=0, viewCount=0;
      public static int obs1_getText=0;
      public static int intent=0, settype_sms=0, uriparse=0;
      public static int startActivityWithContext=0,
startActivityNoContext=0, putExtra=0;
      public static int smsto=0, smsmanager=0, sendtxtmsg=0;
```

```
static String csvFile = "C:\\Users\\TechDev\\Desktop\\KLD_Results.csv";
      static String header[] =
      {"Application", "Activity", "View",
       "setContentView()", "getText()", "EventHandler", //obs1(ben): sms
is sent with visual input and through even handler method
       "SmsManager", "sendTextMessage", "Const arg", //obs2(mal):
SmsManager object is created, sendTxtMsg is invoked, constant arg present
       "Var arg", //obs3 (ben): SmsManager object is created, sendTxtMsg
is invoked, variable arg present
       "Intent", "putExtra(sms_body)", "variable SMS", //obs4 (ben):
using intent object, putting sms body, and variable is used for message
       "Constant SMS", //obs5 (mal): using intent, constant sms message
       "StartActivity", "Uri.parse", "smsto:", "Uri_variable SMS", //obs6
(ben): start activity with smsto uri and variable param (ben)
       "Uri_const SMS", //obs7 (mal): start activity with smsto uri and
const param
       "Toast", "SmsManager.RESULT" //obs8(ben): result is notified
SmsManager based msg delivery

       //"StartActivity_NoContext" //obs7 (mal): start activity with no
context

      };
      static int [] result = new int [20];

      public static void main(String[] args) {
            // TODO Auto-generated method stub

            int appcount=0;

            String path = "C:\\Users\\TechDev\\Desktop\\SMS_sample";
            String appName="";
            writeHeader(header, csvFile);

            File root = new File( path );
            File[] list = root.listFiles();

            if (list == null) return;

            for (File f : list) {
            if ( f.isDirectory() ) {
                  appcount++;
                  System.out.println( "\n****Dir:" + f.getAbsoluteFile()
);
                  String temp = f.getName();
                  StringTokenizer stk = new StringTokenizer (temp, "\\");
                  while (stk.hasMoreTokens()){
                        appName = stk.nextToken();

                        fcount= dcount = imgCount=
activityCount=viewCount =
      intent=settype_sms=0;
                        startActivityWithContext = startActivityNoContext
=                             putExtra = uriparse= 0;
```

```
       walk( f.getAbsolutePath());
                 generateCsvFile(csvFile, appName, result);

                 for (int i =0; i<result.length; i++){
                   result[i] =0;
                 }

             }
         }

         System.out.println( "\nApplication count:" + appcount);

       }
       public static void writeHeader(String [] header, String csvFile){

             try{
                FileWriter writer = new FileWriter(csvFile, true);

                for (int i =0; i< header.length; i++){
                      writer.append(header[i] + ",");

                }
                writer.append ("\n");
                writer.flush();
                writer.close();
             }
             catch(IOException e)
             {
                  e.printStackTrace();
             }
       }

       private static void generateCsvFile(String sFileName, String
appName,    int result[]){
             try{
                 FileWriter writer = new FileWriter(sFileName, true);
                 writer.append(appName+ ",");
                 for (int i =0; i<result.length; i++){
                   writer.append (result[i] + ",");
                 }
                 writer.append('\n');

                 writer.flush();
                 writer.close();
             }
             catch(IOException e){
                  e.printStackTrace();
             }
       }
}
```

References

Aaron, D. B. (2011, November 17). Google android passes 50% of Smartphone Sales. *Bloomberg Businessweek.* Retrieved August 21, 2013, from http://www.businessweek.com/news/2011-11-17/google2android-passes-50-of-smartphone-sales-gartner-says.html

Android Design. *(2013).* Retrieved August 21, 2013, from *http://developer.android.com/design/index.html*

Android IDL Example with Code Description – IPC. (2013, July 20). Retrieved August 21, 2013, from http://techblogon.com/android-aidl-example-with-code-description-ipc

Baldwin, C. (2012, September 17). Android devices vulnerable to security breaches. ComputerWeekly.com. Retrieved August 21, 2013, from http://www.computerweekly.com/news/2240163351/Android-devices-vulnerable-to-security-breaches

Barrera, D., Kayacik, H., Oorchot, P., & Somayaji, A. (2010). A Methodology for Empirical Analysis of Permission-Based Security Models and Its Application to Android. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, 2010, pp. 73-84.

Batyuk, L., Herpich, M., Camtepe, S., Raddatz, K., Schmidt, A. & Albayrak, S. (2011). Using Static Analysis for Automatic Assessment and Mitigation of Unwanted and Malicious Activities within Android Applications. *In Proceedings of 6th International Conference on Malicious and Unwanted Software (MALWARE),* October 2011, pp. 66-72.

Bigi, B. (2003). Using Kullback-Leibler Distance for Text Categorization. *Lecture Notes in Computer Science (LNCS*). Volume 2633, 2003, pp. 305-319.

Blasing, T., Batyuk, L., Schmidt, A., Camtepe, S. & Albayrak, S. (2010). An Android Application Sandbox System for Suspicious Software Detection. In *Proceedings of the Proceedings of the 5th International Conference on IEEE Malicious and Unwanted Software*, 2010, pp. 55-62.

Brinkmann, M. Encrypt all data in Android phone. (2012, October 13). Retrieved August 21, 2013, from http://www.ghacks.net/2012/10/13/encrypt-all-data-on-your-android-phone

Chin, E., Felt, A.P., Greenwood, K., & Wagner, D. (2011). Analyzing inter-application communication in Android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services* (MobiSys '11). ACM, New York, NY, USA, 239-252.

Cooper, V. N. (2014a). Android Malware Detection Based on Kullback-Leibler Divergence", Invited Student Research Abstract to the SAC 2014 Student Research Competition (SRC) program. *Proceedings of the ACM-SIGAPP Conference on Applied Computing (SAC 2014)*, Gyeongju, Korea, March 2014, pp. 1695-1696.

Cooper, V. N. & Haddad, H. M. (2013). Study of Agility in Mobile Application Development. *Proceedings of the International Conference on Software Engineering Research and Practice (SERP 2013),* Las Vegas, Nevada, July 2013, pp. 384-390.

Cooper, V. N., Shahriar, H., & Haddad, H. M. (2014b). A Survey of Android Malware Characteristics and Mitigation Techniques. *Proceedings of the IEEE International Conference on Information Technology: New Generations (ITNG 2014)*, Las Vegas, Nevada, April 2014.

Cooper, V. N., Shahriar, H., & Haddad, H. M. (2014c). Book Chapter titled *Development and Mitigation of Malicious Android Applications*. Contribution to the book titled *Handbook of Research on Digital Crime, Cyberspace Security, and Information Assurance*, Edited by Maria Manuela Cruz-Cunha, Polytechnic Institute of Cávado and Ave, Portugal. Published by IGI Global, Spring 2014.

Cover, T.& Thomas, J. *Elements of Information Theory*, John Wiley and Sons, 2006.

Difference between Adware and Spyware. (2005, July 17). Retrieved August 21, 2013, from http://www.techiwarehouse.com/engine/41cc4355/Difference%20Between%20Adware%20&%20Spyware

dex2jar. (2013). Retrieved August 21, 2013, from https://code.google.com/p/dex2jar/

Enck, W,, Octeau. D., McDaniel, P., & Chaudhuri, S. (2011). A study of android application security. In *Proceedings of the 20th USENIX conference on Security* (SEC 2011). August 2011. USENIX Association, Berkeley, CA, USA, 21-21.

Enck, W., Ongtang, M., & McDaniel, P. (2009). On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conf. Computer and Communications Security (CCS 09),* ACM, 2009, pp. 235-245.

Eudora*. (2014). Eudora.com.* Retrieved August 21, 2013, from http://www.eudora.com

Felt, A. P., Finifter, M., Chin, E., Hanna, S. & Wagner, D. (2011). A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices* (SPSM 2011). ACM, New York, NY, USA, 3-14.

Felt, A. P., Greenwood, K., & Wagner, D. (2011). The Effectiveness of Application Permissions. In *Proceedings. of the 2nd USENIX Conference on Web Application Development (WebApps)*, 2011.

Fukui, K., Sato, K., Mizusaki, J., & Numao, M. (2010). Kullback-Leibler Divergence Based Kernel SOM for Visualization of Damage Process on Fuel Cells. *IEEE International Conference on Tools with Artificial Intelligence*, October 2010, pp. 233-240.

Get Social Updates of your contact list using Ice cream sandwich. (2012). Retrieved August 21, 2013, from http://creativeandroidapps.blogspot.com/2012/07/get-social-updates-of-your-contact-list.html

Google Play*. (2013), Google Play Store.* Retrieved August 21, 2013, from https://play.google.com/store?hl=en

How to get the Android device's Primary Email Address. (2010). Retrieved August 21, 2013, from http://stackoverflow.com/questions/2112965/how-to-get-the-android-devices-primary-e-mail-address

How to make android phone silent in java. (2012). Retrieved August 21, 2013, from://stackoverflow.com/questions/10360815/how-to-make-android-phone-silent-in-java

How to make a phone call in Android. (2011). Retrieved August 21, 2013, from http://stackoverflow.com/questions/1556987/how-to-make-a-phone-call-in-android-and-come-back-to-my-activity-when-the-call-i

Hyatt, E. C. Custom Android Phone, (2013), Retrieved August 21, 2013, from http://sites.google.com/site/edwardcraighyatt/projects/custom-android-phone

Java Decompiler. (2013). Retrieved August 21, 2013, from http://jd.benow.ca/

Li, G. & Wang, Y. (2012). Differential Kullback-Leibler Divergence Based Anomaly Detection Scheme in Sensor Networks. In *Proceedings of 12th IEEE International Conference on Computer and Information Technology (CIT)*, October 2012, pp. 966-970.

Lock and Android phone. (2012). Retrieved August 21, 2013, from http://stackoverflow.com/questions/4793339/lock-an-android-phone

Memory Management in Android. (2010, July 5). Retrieved August 21, 2013, from http://mobworld.wordpress.com/2010/07/05/memory-management-in-android/

Nicolaou, A. (2013). Best Practices on the Move: Building Web Apps for Mobile Devices. *Communications of the ACM*. August 2013, Vol. 56 Issue 8, p45-51.

Oberheide, J., Cooke, E., & Jahanian. F. (2008). Cloudav: Nversion antivirus in the network cloud. In *Proceedings of the 17th USENIX Security Symposium* (Security'08), San Jose, CA, July 2008.

PhonePay Plus. *(2013). Phonepayplus.org.uk.* Retrieved August 21, 2013, from *http://www.phonepayplus.org.uk*

Rastogi, V., Chen, Y., & Jiang, X. (2013). DroidChameleon: evaluating Android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security* (ASIA CCS '13). ACM, New York, NY, USA, 329-334.

Rehm, L. (2012, October 25). A Guide to Android OS. (2012). Retrieved August 21, 2013, from http://connect.dpreview.com/post/8437301608/guide-to-android-os

Reza, H. & Mazumder, N. (2012). A Secure Software Architecture for Mobile Computing. In *Proceedings of the 9th International Conference on Information Technology- New Generations* (ITNG 2012), Las Vegas, NV, pp. 566-571.

Sarkar, S., Rama, G. & Kak, A. (2007). API-Based and Information-Theoretic Metrics for Measuring the Quality of Software Modularization. *IEEE Transactions on Software Engineering*, January 2007, Vol. 33, No. 1, pp. 14-32.

Schmidt, A., Peters, F., Lamour, F. & Albayrak, S. (2008). Monitoring Smartphones for Anomaly Detection. In *Proceedings of the 1st International Conference on Mobile Wireless MiddleWARE,* Operating Systems, and Applications (MOBILEWARE), Article No. 40, Innsbruck, Austria, February 2008.

Schmidt, A.D., Clausen, H., & Camtepe, A. (2009). Detecting Symbian OS Malware through Static Function Call Analysis. In *Proceeding of the 4th International Conference on Malicious and Unwanted Software (Malware 09),* IEEE, 2009, pp. 15-22.

Security Tips. *(2013).* Retrieved August 21, 2013, from *http://developer.android.com/training/articles/security-tips.html*

Send SMS in Android. (2013). Retrieved August 21, 2013, from http://stackoverflow.com/questions/4967448/send-sms-in-android

Seo, S., Lee, D., &Yim, K. (2012). Analysis on Maliciousness for Mobile Applications. In *Proceedings of the 2012 Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing* (IMIS '12), IEEE Computer Society, Washington, DC, USA, 126-129.

Set Wallpaper using WallpaperManager.(2011, March 28). Retrieved August 21, 2013, from http://android-er.blogspot.com/2011/03/set-wallpaper-using-wallpapermanager.html

Shabtai, A., Fledel, Y. & Elovici, Y. (2010). Automated Static Code Analysis for Classifying Android Applications Using Machine Learning. In *Proceedings of the 2010 International Conference on Computational Intelligence and Security* (CIS 2010). 329-333.

Shabtai, A., Fledel, Y., Kanonov, U., Elovici, Y., Dolev, S., & Glezer, C. (2010). Google Android: A Comprehensive Security Assessment*. IEEE Security & Privacy*, March 2010, pp. 35-44.

Tapiador, J. & Clark, J. (2010). Information-Theoretic Detection of Masquerade Mimicry Attacks. In *Proceedings of 4th International Conference on Network and System Security (NSS*), September 2010, pp. 183-190.

What is Malware?. (2013). Retrieved August 21, 2013, from http://www.microsoft.com/security/resources/malware-whatis.aspx

Yang, C., Yegneswaran, V., Porras, P., & Gu, G. (2012). Detecting Money-Stealing Apps in Alternative Android Markets. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS),* October 2012, Raleigh, North Carolina, USA, pp. 1034-1036.